(To appear in CAD special issue on visualization)

Efficient Rendering of Trimmed NURBS Surfaces*

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

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

November 28, 1994

Abstract:

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

Additional Keywords and Phrases: NURBS, trimming curves, Bézier surface, CSG, Boundary Representation, Trimming Curves, Tessellation.

*Supported by Office of Naval Research contract, ONR N00014-94-1-0738, DARPA ISTO Order No. A410, NSF Grant No. MIP-9306208, Junior Faculty Award, University Research Award, NSF Grant CCR-9319957, ARPA Contract DABT63-93-C-0048 and NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, and NSF Prime Contract No. 8920219.

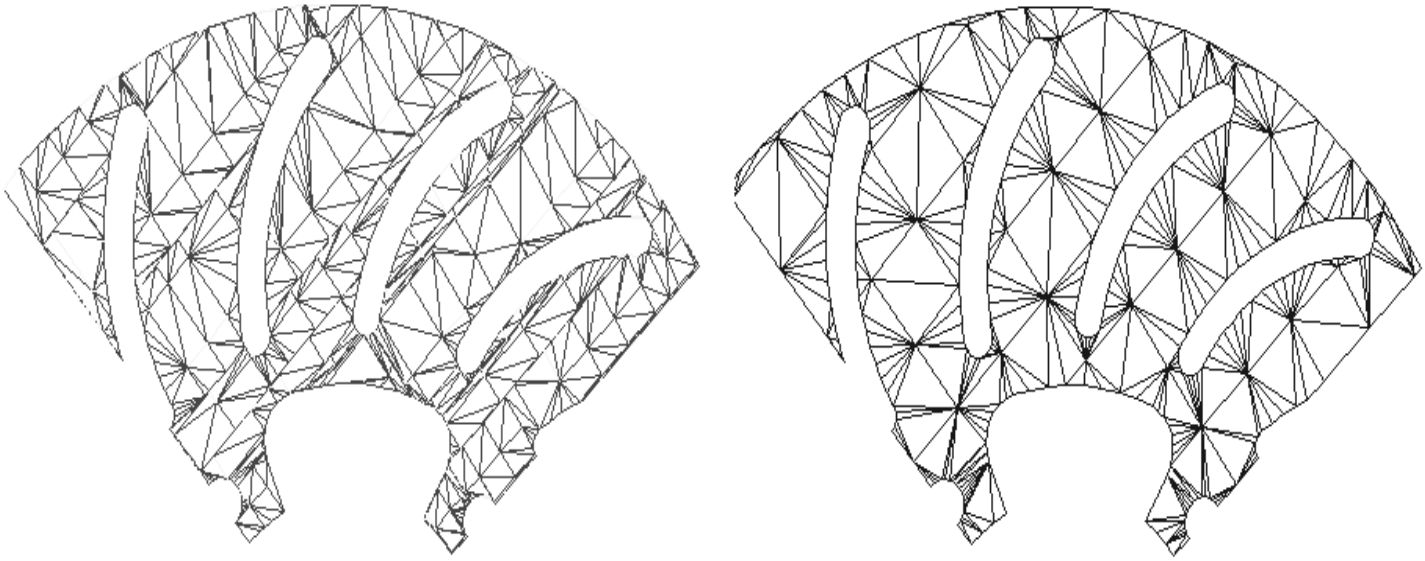
1 Introduction

Many applications involving CAD/CAM, virtual reality, animation and visualization use models described using NURBS (Non-Uniform Rational B-Spline) surfaces. Over the last few years, they have gained a lot of importance in industry and are used to represent shapes of automobiles, airplanes, ships, mechanical parts etc. Recent graphics standards like PHIGS+ and OpenGL have included NURBS surfaces as graphics primitives. Interactive display of models consisting of thousands of such surfaces on current graphics systems is a major challenge. In this paper we focus on trimmed surface models, typically obtained after surface intersection or other boolean operations.

The problem of rendering curved surfaces (both trimmed and untrimmed) has been extensively studied in the literature and main techniques include pixel level surface subdivision, ray tracing, scan-line display and polygonization [Cat74, Kaj82, NSK90, LCWB80]. Techniques based on ray tracing, scan-line display and pixel level display do not make efficient use of the hardware capabilities available on current architectures. In particular, the current graphics systems can render up to millions of transformed, shaded and z-buffered polygons per second [Ake93, Fea89]. As a result, only algorithms based on polygonization come close to real time display. Many different methods of polygonization have been proposed in the literature [AES93, Dea89, LR81, SL87, Roc87, AES91, FMM86]. Broadly speaking they can be classified into uniform and adaptive tessellation of NURBS surfaces. Many of these algorithms focus on trimmed surface models [RHD89, Luk93, LC93, Che93, SC88, Vla90].

We present a fast algorithm for rendering trimmed NURBS models. Our approach shares its principle with the algorithm presented by Rockwood et. al. [RHD89]. This is in terms of converting the NURBS models into Bézier surfaces, using uniform tessellation and computing the untrimmed regions of each cell and triangulating it. This is in contrast with direct rendering of trimmed NURBS surfaces using the B-spline representation [Luk93, LC93]. The rendering algorithm involves the computation of intersections of the trimming curve with the domain cells, visible region determination and triangulation. These operations are relatively simpler and faster to perform on a Bézier representation than on B-splines.

The algorithm in [RHD89] partitions each trimming curve into monotonic segments. This monotonic subdivision followed by special triangulation at the patch boundaries sometimes becomes a bottleneck for the [RHD89] algorithm. We overcome these problems with handling trimming curves and present efficient algorithms for trimmed cell computation and triangulation (Fig 1). We also devise *better bounds* for uniformly tessellating the surface domain into fewer cells and compute the



(a) [RHD89]'s Triangulation (b) Our Triangulation

Figure 1: Comparison of Patch Triangulation (on a patch from the Alpha_1 Rotor)

trimming regions without partitioning them into monotonic regions. In particular, we compute piecewise linear representation of the trimming curves using view dependent bound computation, trace them over the domain cells, partition the cells into trimmed and untrimmed regions and triangulate the trimmed regions. We compare the surface triangulation of the two algorithms in Fig. 1. Fig. 1(a) corresponds to the SGI-GL implementation based on Rockwood et. al.'s algorithm [Nas93, GL]. Our algorithm makes use of *coherence* between successive frames and performs *incremental computation* at each frame. This has a significant impact on the speed of the overall algorithm

The rest of the paper is organized in the following manner. We present our notation and formulate the problem in section 2. In Section 3, we review the algorithm for tessellating the domain into cells as a function of the viewing parameters. Section 4 handles trimming curves and triangulation of the untrimmed regions for each surface. In section 5 we make use of coherence between successive frames and highlight the incremental algorithm. Finally, we discuss implementation in Section 6. In this paper we have demonstrated these techniques on tensor-product surface models only. They can be generalized to triangular patches as well.

2 Problem Definition

Given a trimmed NURBS surface model, we use knot insertion algorithms to decompose it into a series of Bézier patches [Far90]. We also subdivide the NURBS trimming curves at the patch boundaries and transform them into Bézier curves. Piecewise linear trimming curve representations are decomposed at the patch boundaries as well. All these steps are part of the preprocessing phase. Decomposing NURBS patches and trimming curves allow us to derive better bounds on derivatives and curvature using the Bèzier representation. The resulting algorithm for trimmed region computation also becomes much simpler as well.

2.1 Notation

We use the following notation for the rest of the paper. The 3D coordinate system in which the NURBS model is defined is referred to as the *object space*. Viewing transformations, like rotation, translation and perspective, map it onto a viewing plane known as the *image space*. Associated with this transformation are the viewing cone and clipping planes. Finally, *screen space* refers to the 2D coordinate system defined by projecting the image space onto the plane of the screen. In all our pictures of the (u, v) domain, the v axis is horizontal and the u axis is vertical.

An $m \times n$ rational Bézier surface is specified by an $(m+1) \times (n+1)$ mesh of control points, $\{(\mathbf{r}_{ij}, w_{ij}) = (x_{ij}, y_{ij}, z_{ij}, w_{ij})\}$. The surface, also referred to as a patch, is represented as a tensor product parametric equation with parameters $(u,v) \in [0,1] \times [0,1]$:

$\mathbf{F}(u,v) = (\mathtt{x}(u,v), \mathtt{y}(u,v), \mathtt{z}(u,v), \mathtt{w}(u,v)) =$

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

$B_i^m(u), B_j^n(v)$ are the Bernstein polynomials.

Each surface also has a set of trimming curves (which may or may not be loops) associated with it. These can be either piecewise linear or Bézier curves. A piecewise linear curve with k segments is specified by a sequence of $k+1$ points $\mathbf{C}_{pl} = [\mathbf{p}_0 \ldots \mathbf{p}_k]$ A Bézier curve of degree n is specified by a sequence of $n+1$ control points, $\{(\mathbf{p}_i, s_i) = (u_i, v_i, s_i)\}$. The parametric equation of the curve is:

$\mathbf{C}(t) = (\mathtt{u}(t), \mathtt{v}(t), \mathtt{s}(t)) =$

$$(\Sigma_{i=0}^{n} s_i\mathbf{p}_i B_i^n(t), \Sigma_{i=0}^{n} s_i B_i^n(t)).$$

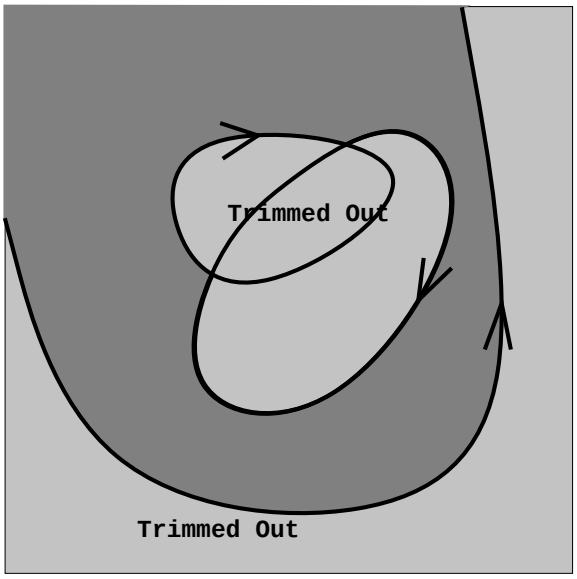


Figure 2: Trimming Rule

2.2 Trimming rule

Every trimming curve is an oriented curve, i.e. it has a starting point, $\mathbf{p}_0$, and an ending point, $\mathbf{p}_{last}$. For loops $\mathbf{p}_0 = \mathbf{p}_{last}$. A point P is on the right of the curve if the segment PQ makes 90° in the clockwise direction from the tangent to the curve at Q, where Q is the point on the curve closest to P. A trimming curve is said to *trim out* the part of the patch that lies on its right. Thus a clockwise loop trims out the enclosed region. If two curves intersect, the region trimmed out is the union of regions trimmed out by each of the curves.

This is called the *handedness rule* of trimming. It does not allow self intersecting trimming curves but two different trimming curves can intersect. Sometimes a *winding rule* is used to defined the exterior of a curve. According to this rule, the region of a surface that is enclosed by an even number of loops is trimmed out. This means that trimming curves must explicitly be loops. Fig. 2 shows a trimmed region, based on the handedness rule. However we would have to redefine the trimming curve if we used the winding rule.

2.3 Problem Formulation

A patch is evaluated at a set of points on a rectangular grid made of n_u equi-spaced lines isoparametric in u and n_v equi-spaced lines isoparametric in v. A curve is evaluated at n_t equi-spaced

points along its parameter t.

Triangles are generated by taking adjacent points on this grid and the trimming curves, three at a time, such that they do not overlap and they do not lie in the trimmed region of the patch.

The problem is to find n_u, n_v and n_t , and to find a method for generating these triangles such that:

- The triangles form a reasonable approximation to the trimmed Bézier surface: they should not deviate from the surface more than a user specified tolerance, TOL_d, in screen space.
- The triangles have a reasonable size: the edges of the triangles should be shorter than a user specified tolerance, TOL_s, in screen space.
- The triangle generation and rendering is done at *interactive* rates.

The first criterion is known as the *deviation criterion* for polygonization. It is a function of the second derivative vector of the given surface representation. The second criterion is referred as the *size criterion*. These two are used to obtain a good polygonization of the surface such that we obtain a smooth image after Gouraud or Phong shading of the polygons [KM94].

3 Tessellation Computation

In this section, we highlight the algorithm for tessellating the surface as a function of the viewing parameters. More details are given in [KM94]. In particular, we dynamically compute the polygonization of the surfaces as the viewing parameters are changing. Polygonization can be computed using uniform or adaptive subdivision for each frame. Uniform tessellation involves choosing constant step sizes along each parameter. Adaptive tessellation uses a recursive approach to subdivision with a stopping condition (normally based on some "flatness" and "surface area" criteria). For large scale models, uniform subdivision methods have been found to be faster in practice [FMM86, KM94]. In practice, large scaled NURBS models typically consist of relatively flat surfaces. This is indeed the case after converting B-spline models into Bézier surfaces. Adaptive subdivision performs well on surfaces with highly varying curvatures and large areas. In such cases uniform tessellation may oversample them. The performance of uniform tessellation algorithms is a direct function of the tessellation step sizes, and these need to be computed carefully. In the context of uniform tessellation, the evaluation of Bézier polynomials can be optimized if we use points that lie on an isoparametric

curve: we can reuse one of the factors of the tensor product or use forward differencing [Roc87]. Another major reason for the choice of uniform tessellation is the relative simplicity and efficiency of handling trimming curves (as compared to adaptive subdivision).

3.1 Patch tessellation

There is considerable literature on computation of bounds on polynomials [LR81, FMM86, Roc87, AES91]. They are base upon the size or the deviation criteria. Sometimes a *normal deviation criterion* is also used. This criterion bounds the deviation of the triangles' normals from the surface normal. While this bound can improve the image quality, it is relatively expensive to evaluate.

We have chosen to use the size criterion to tessellate each patch. The deviation criterion is a function of the second order derivative vector and becomes relatively expensive on rational surfaces. The size criterion by itself may not result in a good approximation on patches with small area and highly varying curvature. A simple technique to account for such cases has been described in [KM94].

The size criterion can be applied in two ways for step size computation:

- Compute the bounds on the surface in the object space as a preprocessing step and map these to the screen space. The step size is computed as a function of these bounds and viewing parameters [LR81, FMM86, AES91, KM94].

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

The advantage of the first method is that it reduces the on-line time for bounds computation. Little computation is required to calculate the desired step size for a patch given the viewing parameters (the transformation matrix). Though the bound calculated by the first method is tighter than that by the second method, the mapping of bounds to screen space may not be exact. Hence the first method ends up using a smaller step size, and thus generating more triangles than the second one.

For the Bézier surface, $\mathbf{F}(u,v)$, the tessellation parameters satisfying the size criterion are computed in the object space as:

$$n_u = \sqrt{2}\frac{\| \overline{\left(\frac{\mathtt{x}(u,v)}{\mathtt{w}(u,v)}\right)_u}, \overline{\left(\frac{\mathtt{y}(u,v)}{\mathtt{w}(u,v)}\right)_u}, \overline{\left(\frac{\mathtt{z}(u,v)}{\mathtt{w}(u,v)}\right)_u} \|}{TOL_s},$$

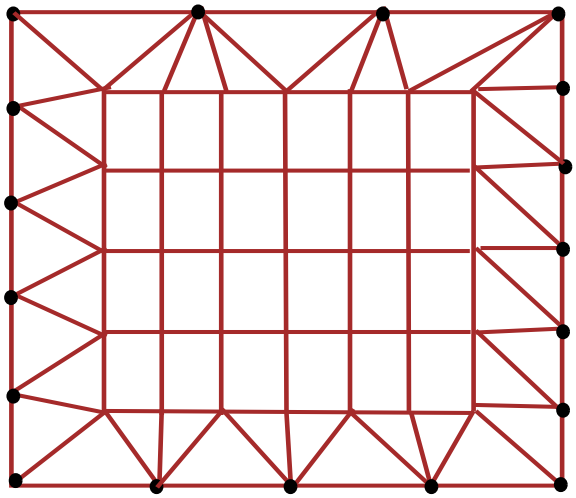


Figure 3: Coving and Tiling

where $\overline{\left(\frac{\mathtt{x}(u,v)}{\mathtt{w}(u,v)}\right)_u}$ is the maximum magnitude of the partial derivative of $\left(\frac{\mathtt{x}(u,v)}{\mathtt{w}(u,v)}\right)$ with respect to u in the domain $[0,1] \times [0,1]$.

The maximum magnitude of the second derivative is needed to calculate the stepsize satisfying the deviation criterion [AES91].

n_u steps need to be taken along the dimension u for the criterion to be satisfied. n_v for the patch is computed analogously. Computations of n_t, the number of steps for a curve is also calculated similarly. More details are given in [KM94].

3.2 Boundary Curve Tessellation

The number of steps that two adjacent patches are tessellated into, need not be the same. This can result in cracks in the rendered image. To avoid these cracks we need to make sure that the number of steps a boundary curve is tessellated into, is the same on patches on both sides of the boundary. This is easily achieved by generating triangle-strips (called coving triangles) at the boundaries (see Fig. 3) as described in [RHD89] for the boundaries of untrimmed patches. Since the trimming curves can themselves be boundary curves (especially when they arise from intersection of surfaces), we need an extension of the same concept here also. Given two surfaces $\mathbf{F}(u,v)$ and $\mathbf{G}(p,q)$ and a curve $\mathbf{C}$, that lies on both $\mathbf{F}(u,v)$ and $\mathbf{G}(p,q)$, can have different parametric representation in the (u,v) and (p,q) domains (Fig. 4). If $\mathbf{C}$ is not tessellated into the same number of points on both

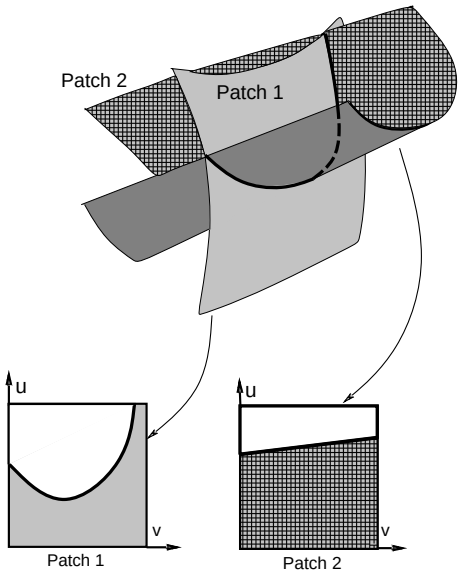


Figure 4: Trim Curves: Diffent representations in different domains

$\mathbf{F}(u,v)$ and $\mathbf{G}(p,q)$, cracks may occur. This is not as straightforward as on the patch boundary curves $u=0, u=1, v=0$ and $v=1$. The same trimming curve can be (and often is) represented differently on different patches. We need two steps to avoid cracks at trimming curves. First, we must take a curve's representation from the (u,v) space of the corresponding patch to the object (X,Y,Z) space. This is done by substituting the curve equation into the patch equation as follows: For a trimming curve $\{(\mathtt{u}(t),\mathtt{v}(t),\mathtt{s}(t))\}$[1] on an $m \times n$ patch $(\mathtt{x}(u,v),\mathtt{y}(u,v),\mathtt{z}(u,v),\mathtt{w}(u,v))$, the curve in the object space is

$$(\mathtt{x}(U(t),V(t)),\mathtt{y}(U(t),V(t)),\mathtt{z}(U(t),V(t)),\mathtt{w}(U(t),V(t)))\,.$$

where $U(t) = \mathtt{u}(t)/\mathtt{s}(t)$, and $V(t) = \mathtt{v}(t)/\mathtt{s}(t)$.

The second step is to unify the representation of the curve on all adjacent patches. This is done by finding the common curves on different patches and choosing one of the representations for all adjacent patches for the purpose of computing bounds: for each boundary curve we *associate* with it, one of the patches it lies on. For further details refer to [KM94]. Note that the both these steps are preprocessing steps and are not done at the display time. If a patch abuts another patch, as in a T-junction, they do not have a common boundary; cracks still may appear. Such cracks can

[1]If the trimming curve is specified as a piecewise linear curve, we use the definition of the curve as a valid tessellation and assume that the same set of points define the curve for all patches that the curve trims.

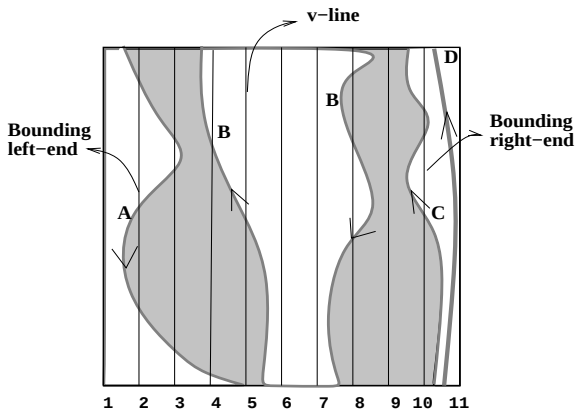


Figure 5: Active v-lines

also be avoided by preprocessing the model, but this preprocessing is computationally expensive. Fortunately such cases rarely occur in practice.

4 Tessellation

Given n_u and n_v, it is straightforward to construct a grid in the (u, v) domain. The grid points divide the domain into rectangles (we draw a diagonal to get the desired triangles). At the patch boundaries we construct coving triangles. We refer to these rectangles and triangles, *cells*. It is over the canvas of these cells that we need to trace the trimming curves.

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

A grid line perpendicular to the v axis, the isocurve $v = K$, is called a *v-line* (Fig. 5), K being the *v-value* of the v-line. The side of a cell that lies on a v-line is called its *v-edge* and the side that lies on a u-line is called its *u-edge*. The region of the patch lying between two adjacent v-lines is

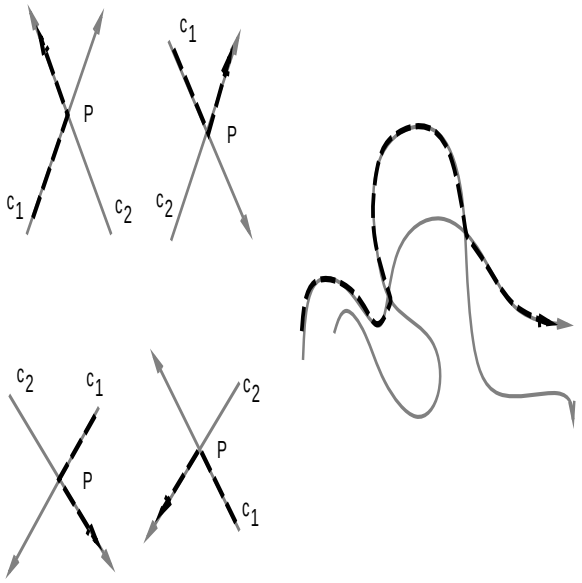


Figure 6: Trimming Curve Intersection

called a *v-strip*. The terms *u-line, u-strip* and *u-value* are defined similarly.

For simplicity, the tessellation algorithm is presented here for rectangular cells only; the extension for the triangular cells at the patch boundaries is straightforward. The main steps of the algorithm are enumerated below. They are elaborated in the subsequent subsections.

1. Eliminate redundant and intersecting curves. This is a preprocessing step.

2. Compute n_u and n_v for the patch and n_t for each trimming curve, using the method introduced in section 3.

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

4. **For each** active strip

   For each active cell in the strip

   Triangulate the cell.

4.1 Intersection

Initially we calculate all pairwise intersections between curves whose bounding boxes overlap. We use the recently developed algorithms for intersecting parametric and algebraic curves, which reduce the problem to an eigenvalue problem [MD94]. We merge two intersecting curves into one by eliminating sections of the curves. When a part of a curve lies in a region trimmed out by another curve, that part is redundant and can be discarded. Fig. 6 demonstrates how this is done. The basic idea is very simple. Let two curves c_1 and c_2 intersect at a point $\mathbf{P}$. $\mathbf{P}$ divides each curve into two segments, one terminating at $\mathbf{P}$ and the other starting at $\mathbf{P}$. Call these *in* and *out* segments, respectively. If a curve, say c_1, is a loop, then there exists another point $\mathbf{P'}$ where c_2 intersects it. (If there is more than one intersection, choose the point that comes immediately before $\mathbf{P}$.) The two segments are determined by the two points $\mathbf{P}$ and $\mathbf{P'}$ in that case. Draw the tangent vectors (remember that the curves are oriented) of the two curves at $\mathbf{P}$, $\vec{\mathbf{T}}_\mathbf{1}$ and $\vec{\mathbf{T}}_\mathbf{2}$. If $\vec{\mathbf{T}}_\mathbf{2}$ makes an angle less than 180° from $\vec{\mathbf{T}}_\mathbf{1}$ (in the counter-clockwise direction) then the in-segment of $\vec{\mathbf{T}}_\mathbf{2}$ and the out-segment of $\vec{\mathbf{T}}_\mathbf{1}$ are redundant. Otherwise the out-segment of $\vec{\mathbf{T}}_\mathbf{2}$ and the in-segment of $\vec{\mathbf{T}}_\mathbf{1}$ are redundant. If two curves have coincident segments, we discard the coincident part from both curves. If the curves are tangent to each other, we cannot discard any segments. Such cases are recorded as special cases. For multiple intersections, this process is repeated recursively.

Once we process all the intersections, each resulting trimming curve is a piecewise sequence of curves and no trimming curves intersect except at the patch boundaries. Note that some redundant curves that do not intersect any curve (curve D in Fig. 5) may still remain. This happens when a clockwise loop lies inside another (or is tangent to it)[2]. This containment is tested for each pair of non-intersecting curves whose bounding boxes overlap [NSK90].

We tessellate each curve on-line and get a piecewise linear representation: a sequence of points $p_0 \ldots p_n$. Note that the points of intersection must appear in this sequence. While the actual curves do not intersect, their piecewise approximations might. We keep TOL_d smaller than the minimum distance between two curves to avoid this. When two curves are tangential to each other (or come very close), we choose the point of tangency as an extra tessellant on each curve. This allows us to choose a TOL_d greater than zero, but it still must be smaller than the closest distance between the curves not in the δ neighborhood of the point of tangency, where δ is the tessellation step size.

[2]The case of non-loops can be reduced to this by extending such curves along the patch boundaries and turning them into loops.

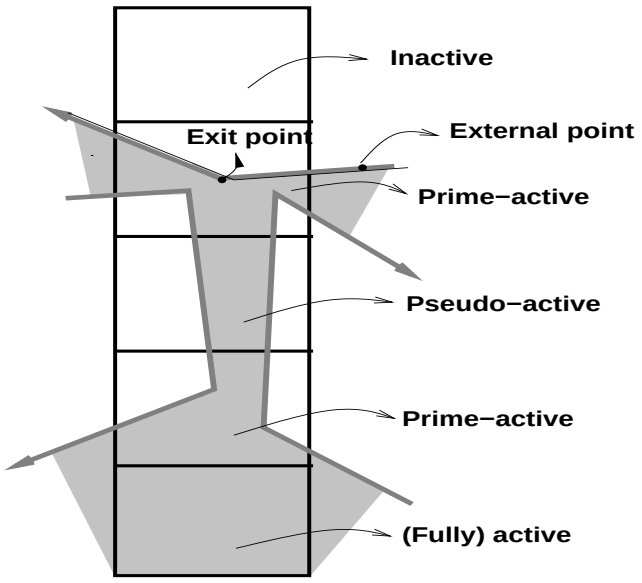


Figure 7: Active v-cells

4.2 Tracing

For each curve, we need to trace from p_0 to p_n marking any *cell-crossings*: the points where the curve crosses a u-edge or a v-edge. If the curve coincides with a u-edge or a v-edge, we do not call them crossings: we do not need to do any extra processing. Similarly if a curve tessellant lies on a cell-edge, we do not call it a a cell-crossing. Two outputs are produced:

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

2. The cell crossings and the $p_i s$ that lie inside the cell, the *in-points*, and those that are outside the cell and are adjacent to that cell's in-points – the *external points* (Fig 7). The in-points that are adjacent to an external point are called *exit points* of the cell. All cells that lie on the segment between an external and exit point are called *external cells*. Further, if there is a curve that doesn't intersect any cell boundary, and lies completely within a cell, that cell is marked as having a hole. While triangulating such cells, we should not triangulate the holes.

The minimum valued v-line a curve crosses is called the *left-end* of the curve. If at left-end, the curve moves from a higher u-value to a lower u-value, the left-end is called *bounding left-end*. An active range of v-lines always starts at the *bounding left-end*. We start at the first point p_0 of the curve, and take n tracing steps, each step processing $[p_i\ p_{i+1}]$.

If v-value(p_i) > v-value(p_{i+1}), the tracing step updates the *current bounding left-end* if v-value(p_{i+1}) < v-value(*the current bounding left-end*). All bounding left ends are found at the end of tracing.

The range of active cells (on active v-strips) are also found similarly. The minimum valued u-line that is crossed in a v-strip is the *bottom-end* of the curve on the strip. At the *bounding bottom-end*, the curve moves from a lower v-value to a higher one. All the bounding *bottom-ends* of a strip encountered by the tracing steps are also recorded.

The cells that have a cell-crossing (and hence are active) are called *prime-active* cells (Fig. 7). While triangulating the cells, we always start at the lowest v-valued active v-strip, and the lowest u-valued active cell on the v-strip. On a v-strip, if we encounter a *prime-active* cell that has its right v-edge trimmed out, that is the end of the current range. The next range of active cells starts at the next bounding bottom-end on the strip. Similarly if all cells on a strip have their top u-edges trimmed out, we move on to the next range of active v-lines.

Some active cells do not have any in-points and all its corners are trimmed out. These are *pseudo-active* and do not need any processing in the triangulation step. To triangulate the active cells that are neither prime nor pseudo active, we just draw one of its diagonals.

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

4.3 Cell Triangulation

This section describes the triangulation of prime-active cells. Once we know the distribution of points within (and adjacent to) a cell we connect these points into a set of triangles. Even though the rendered part of the cells can be potentially concave, most of them tend to be convex in practice and have few edges. Therefore, we optimize our triangulation algorithm for the most general case (even if the worst case complexity is a little high).

4.3.1 Cell Polygons

For each cell we know the polygons that need to be shaded: these consist of its in-points and the cell corners that are not trimmed out. To delineate these, we need the cell crossings sorted in counter clockwise order (starting at any crossing at which the curve enters the cell). Since most cells have few crossing of an edge, this sorting step doesn't become a bottleneck. Also, the order of crossing

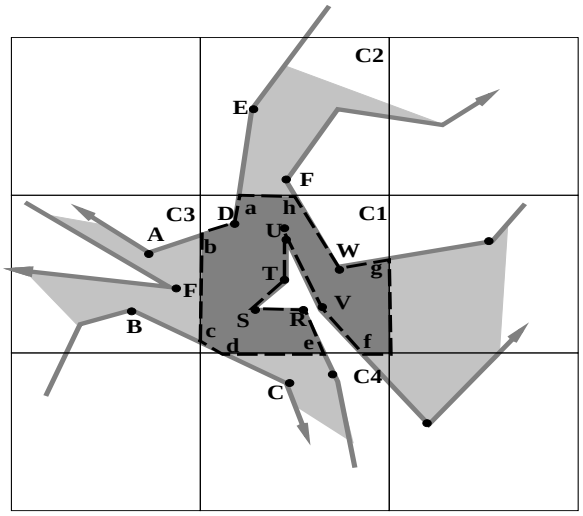


Figure 8: Trimmed Cell

doesn't change from one cell to the next, since trimming curves do not intersect any more. Only new crossings need to be inserted to, or deleted from, the sorted list. Furthermore, we do not need the actual intersection point of curves and edges. Just looking at the external and exit points of two crossings, we can decide their order. The polygons are constructed in the following manner:

1. Start at the first cell crossing, X_0, by the curve c_0.
2. Add all in-points on the curve c_0, till it crosses out of the cell at X_c.
3. Add the crossing X_1 next to X_c in the sorted order. If X_c and X_1 lie on different edges of the cell, add the intermediate corners of the cell before X_1. (If no corners are added, and there are no in-points in the cell, this is a pseudo active cell.)
4. If X_1 is the same as X_0, one polygon is complete. Output the polygon, delete all its points from the sorted list. Start a new polygon with the crossing next to X_c as the new X_0.

In Fig. 8, if we start at the crossing **a**, we add **D** before moving to the crossings **b**-**e** in that order. Points **R**-**V** are added next before the crossing **f** is encountered. The cell corner **P** is added before we move on to **g** since between **f** and **g** we switch edges as the curve went out of the cell at **f**. When we reach back to **a** thus, the polygon is complete. There are no more crossings to process, hence there are no more polygons to generate, and all generated polygons are simple.

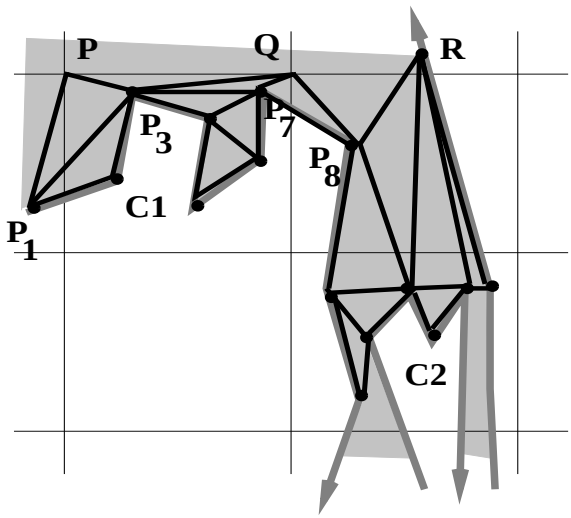


Figure 9: Triangulation

4.3.2 Triangulating simple polygons

We can triangulate this polygon and send the triangles down for rendering[3]. There is one problem, though. The dark shaded polygon in Fig. 8 has vertices **a-h** that are not on the original tessellation of the curve. Such extra points can be at two different parameter values of the curve for two different surface tessellations. While we can avoid cracks by generating a degenerate triangle between each pair of corresponding external and exit points and the crossing, e.g. **AbD** in Fig. 8, the rendered image isn't smooth near these skinny triangles.

We use the following technique to avoid introducing these extra points: Let us consider the intersection point **b**. If we use the external point **A** instead of **b**, we get a correct triangulation there. Similarly we can replace all crossings with the corresponding external points, and avoid creating extra tessellants on the curve. This polygon can now be triangulated using any polygon triangulation algorithms [PS85, CTV89, Sei91]. Our current implementation uses Fortune's algorithm and robust implementation of Delaunay triangulation [For87].

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

1. The external point of one cell is the exit point of another. If these points are included in polygons for both the cells, some overlapping triangles can be created. e.g. Figs. 10(a) and

[3]Note that if the capability to render general polygons is available, it will be faster to send these polygons directly down the pipeline.

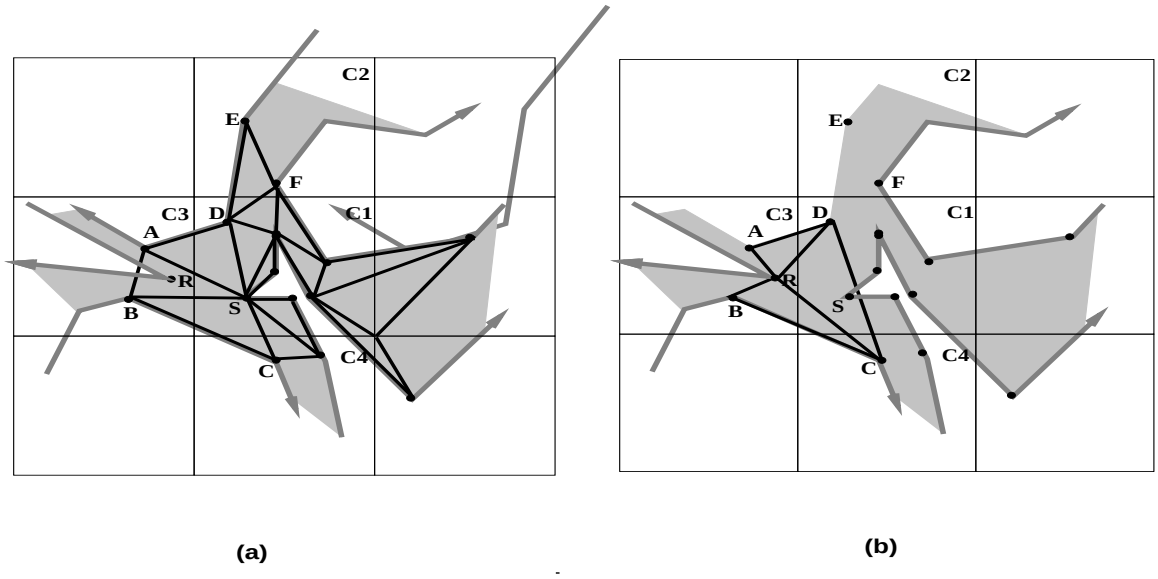


Figure 10: Conflicting Triangulation

(b) show a part of the triangulation for cells C1 and C3 respectively. We cannot draw both triangles **ADS** and **ADR**.

2. Since a cell doesn't keep any information about the triangles of other cells (the idea being that all cells could be processed in parallel on different processors), it can end up drawing wrong triangles. This is clear from Fig. 10(a). The triangle **ABS** intersects a curve in cell C3 and spans an untrimmed region.

One way to avoid these problems is to test if the triangle edges to external points intersect any curve. This is expensive, since even if there are no such intersections, this test must be made. A better way is to postpone drawing a triangle that has an external point for a vertex. Such leftover regions are done at the end of the triangulation step, in a cleanup step.

4.3.3 Cleanup

Once the cell that contains an external point is triangulated (C3 in this example), that external point becomes ready for the final triangulation. Consider the external point **A** of cell C1 in Fig. 10(a). We generate a new polygon **ARSD** (Fig. 11) for triangulation. This polygon has all the points in C1 that **A** is adjacent to and those in C3 that its external point **D** is adjacent to. This does waste a part of the work done earlier, but only if there are too many concavities in the region, which is

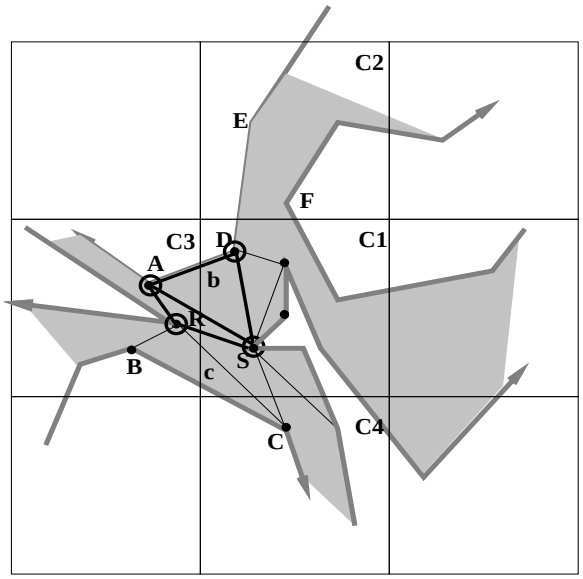


Figure 11: Cleanup Step

not common. These points are taken in the same respective orders as they were in the individual polygons for C1 and C3. (Notice that the vertices of a polygon are circularly ordered. The points in C3 must start at **A** and the points in C1 must end at **D**, since the direction of the curve is from **D** to **A**.) When a segment spans more than two cells, the points in all intermediate cells that are connected to any of the two end points are also included in the polygon. If one of the points, say $\mathbf{P_0}$ in the generated polygon lies in a cell not considered for this polygon, that cell can be processed now for efficiency: all the points in that cell, which are adjacent to the external point corresponding to $\mathbf{P_0}$, should be added to the polygon in order[4].

Extra polygons need to be generated for each pair of external points. These extra polygons have only three vertices most of the time. We perform a simple optimization here. If the only points an external point shares a triangle with, are exit points or cell corners, we just discard the triangles generated by this cell and use the triangles generated by the cell the external point lies in. The cells at the ends of the segment must not both throw away their triangles. A simple priority scheme takes care of that. The cell with lower priority discard its triangle. Fig. 12 shows a normally occurring case. No extra triangulation is required for this example.

[4]In fact we can process all pairs of external points. e.g. in Fig. 11, **BC** can be considered with **AD** since the corresponding crossings **b** and **c** are consecutive in the sorted order for cell C1 and the segment **bc** is not trimmed out.

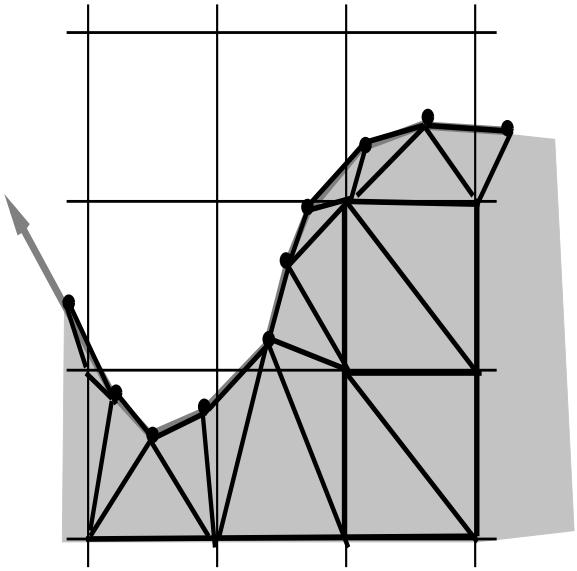


Figure 12: Triangulation: Common Behavior

5 Coherence

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

Whenever we evaluate points (and normals) of the surface we store them and their triangulation in memory. If the bounds for the next frame are higher, we evaluate some additional points and if they are lower we discard some points (see below). When memory is not at a premium, we can retain these extra points. It turns out that, triangle rendering takes more time than triangle generation (and can easily become a bottleneck) [KM94]. So while extra points may be retained, the triangulation must be redone using only the correct set of points. In our experience, we hardly ever need to store more than $60-70$ thousand triangles, needing about $3-4$ megabytes of memory. Thus the memory requirement is not stringent for today's graphics systems.

5.1 Incremental evaluation

When the tessellation bound for a patch increases, we always add complete v-lines or u-lines in the middle of a v-strip or u-strip respectively. This lets us retain the advantages of uniform tessellation,

even though the tessellants are not uniformly placed in the parameter space. For each trim curve, we introduce points between two existing points and split that tessellation step. The following discussion talks only about the curve, but updating the tessellation of patch is similar.

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

Instead, we introduce only $\max(K-2, 0)$ extra tessellant per interval. To decide where the next $\Delta_t = \overline{n}_t - (K-1)n_t$ tessellants are, consider the intervals that do not satisfy the tolerances. If there are fewer intervals than Δ_t, we get fewer than $\overline{n}_t$ tessellations but all criteria are still satisfied. In case there are more, we have two options:

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

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

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

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

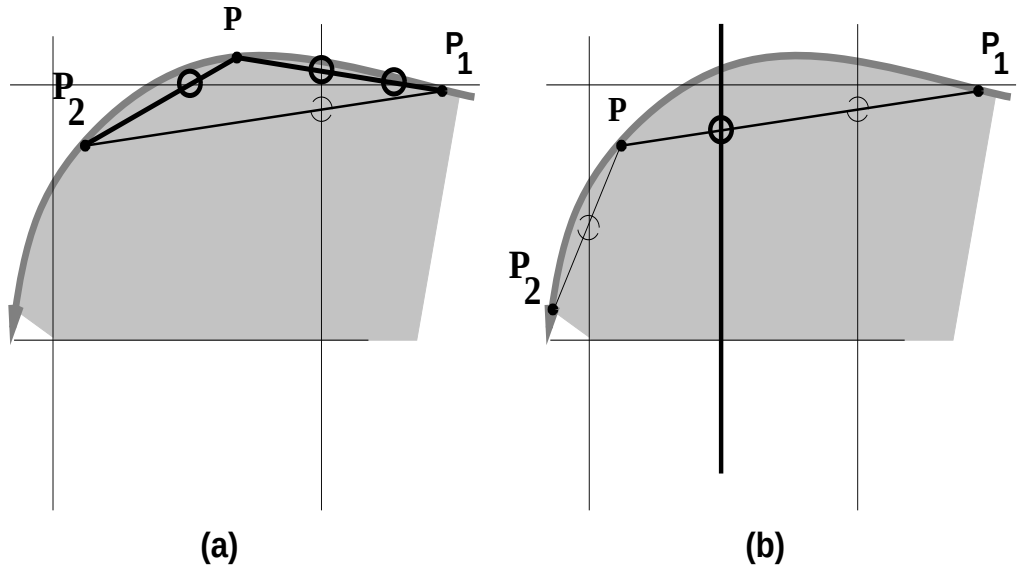


Figure 13: Coherent Tracing

5.2 Incremental Triangulation

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

5.2.1 Tracing

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

If we introduce the point P between P_1 and P_2 (Fig. 13(a)), only the crossings of segment P_1P_2 change. In a sense we need to trace only the part P_1PP_2 of the curve. Similarly if we delete the point P between P_1 and P_2, we just need to retrace P_1P_2. The sorted order of crossings of most cells remain mostly unchanged. We occasionally need to insert or delete a few crossings in the sorted list.

If we add a new v-line (Fig. 13(b)), only the curves crossing the adjacent v-lines (and those contained completely within them) potentially cross this new v-line. Again we need to trace only a part of the curve: the sections within the external points of the cells on this v-strip, P_1P_2, need to be traced. Addition of u-lines is processed similarly.

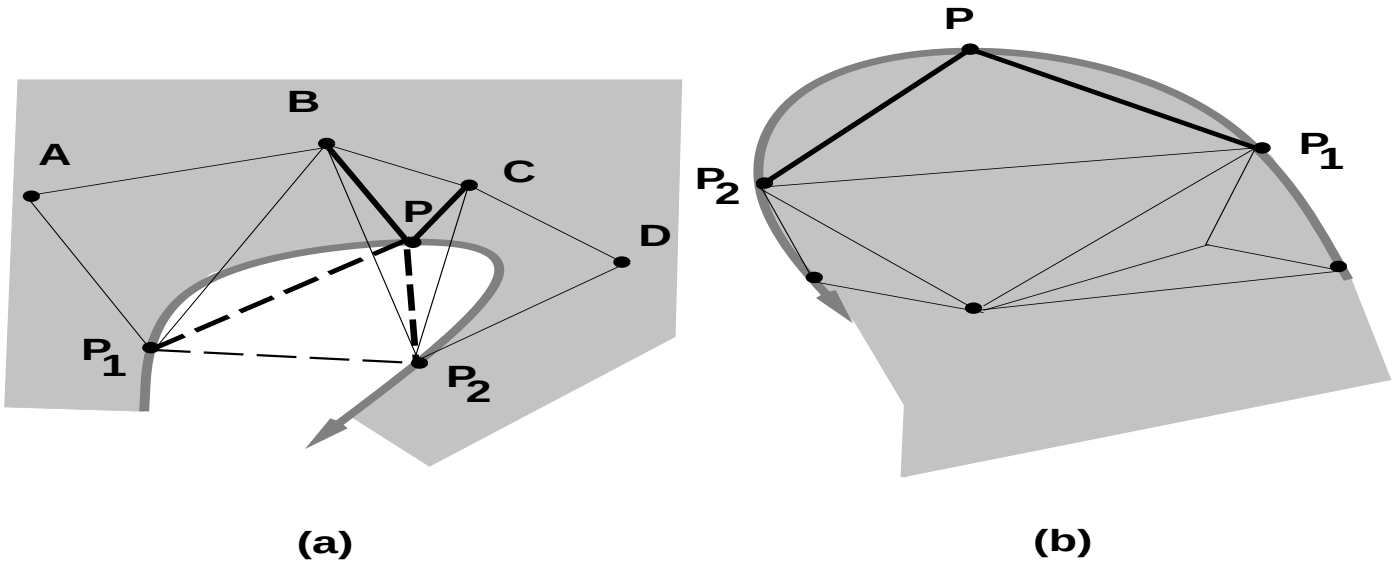


Figure 14: Coherent Triangulation

5.2.2 Triangulation

Once we have the new set of crossings we can update the triangulation also. When P is added in Fig. 14, We only need to update the triangles in the cells that P_1P and PP_2 cross. Two cases can occur:

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

2. We need to add some more region to the triangulated section. Fig. 14(b), we can just add the triangle PP_1P_2. If a new crossing adds a cell corner to the untrimmed region, that corner is added to the triangulation.

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

When a v-line is added in or removed from a v-strip, we need to retriangulate the region in the neighborhood of the v-strip. Most of the work is limited to partially trimmed cells. When a v-line

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

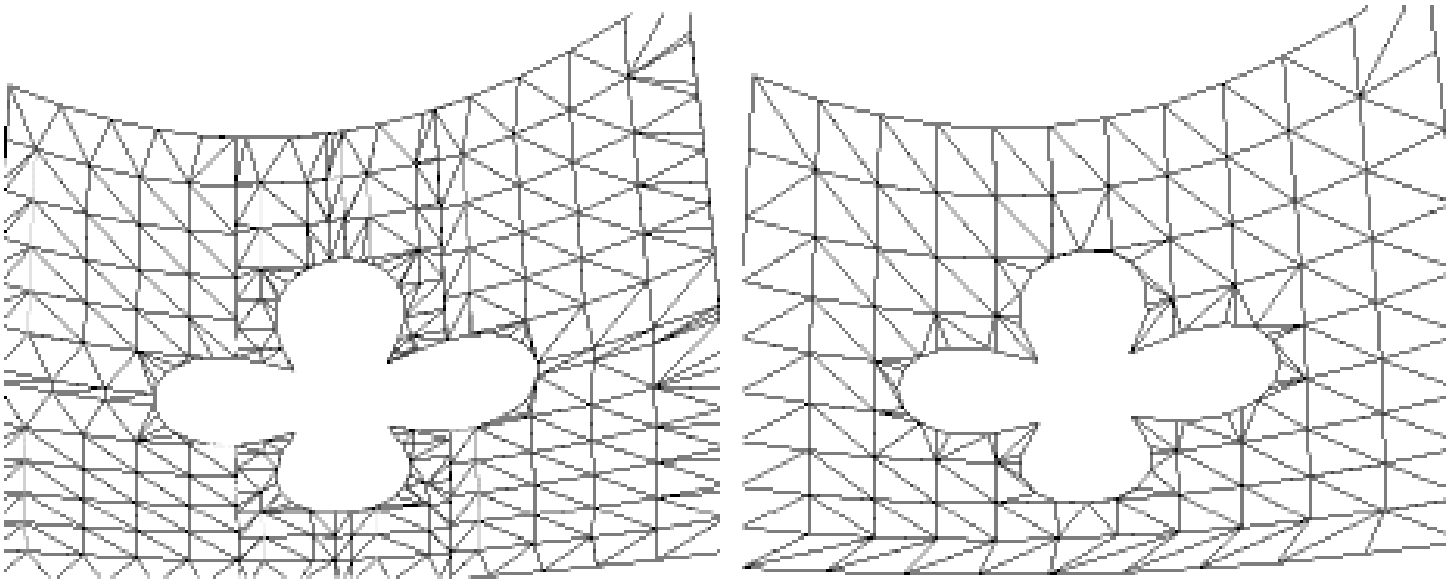
6 Implementation & Performance

Our algorithm performs well in practice and coving is no longer the bottleneck. The polygonization produced by our algorithm is compared to that of the implementation in SGI-GL library based on [RHD89] in Figs. 1 and 15.

We implemented this algorithm on two platforms – SGI-Onyx and the Pixelplanes 5 system at UNC. The Onyx was used with a single processors. The Pixelplanes 5 configuration included 30 graphic processors (though one of them is a master processor and does not perform the computations) and 14 renderers [Fea89]. The algorithm achieves load balancing by distributing neighboring patches onto different processors statically. No inter-processor communication is required. As a result it can be easily ported to any other multiple processor machine.

The performance of the algorithm on the SGI has been shown in Table 6. The SGI-GL implementation [Nas93] is based on the algorithm presented in [RHD89] and has a microcoded geometry engine implementation for surface evaluations. This implementation works in immediate mode. So, even though we had turned off all usage of coherence in our algorithm, we did not have to do any monotonic decomposition, while the GL implementation needlessly did it for every frame. The comparison should be looked at with that in mind. The PixelPlanes implementation includes coherence.

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



(a) [RHD89]'s Triangulation (b) Our Triangulation

Figure 15: Patch Triangulation (on a patch from the Trimmed Teapot)

7 Acknowledgements

We thank Henry Fuchs, Elaine Cohen, Anselmo Lastra, Russ Fish and David Johnson for many helpful discussions. We are grateful to Elaine Cohen and Richard Riesenfeld for the Alpha_1 models used to test the performance of our algorithms.

Model	SGI-GL Implementation	Our basic SGI Implementation	Our SGI Impl. with coherence	Our PixelPlanes Implementation
UNC Trimmed Utah Teapot	4	4	10	25
Utah Brake Assembly	0.33	.46	4	17

Table 1: Performance: Number of frames rendered per second

References

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

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

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

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

[BFHK93] R. Barnhill, G. Farin, D. Hansford, and R. Klass. Adaptive surface triangulation. Manuscript, 1993.

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

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

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

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

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

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

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

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

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

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

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

[For87] S.J. Fortune. A sweepline algorithm for voronoi diagrams. (2):153–174, 1987.

[GL] *The SGI Graphics Library manual.*

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

[KM94] S. Kumar and D. Manocha. Interactive display of large scale nurbs models. Technical Report TR94-008, Department of Computer Science, University of North Carolina, 1994.

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

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

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

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

[MD94] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves i: simple intersections. *ACM Transactions on Graphics*, 13(1):73–100, 1994.

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

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

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

[RHD89] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. In *Proceedings of ACM Siggraph*, pages 107–17, 1989.

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

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

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

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

[Vla90] V. Vlassopoulos. Adaptive polygonization of parametric surface. *Visual Computer*, 6:291–298, November 1990.

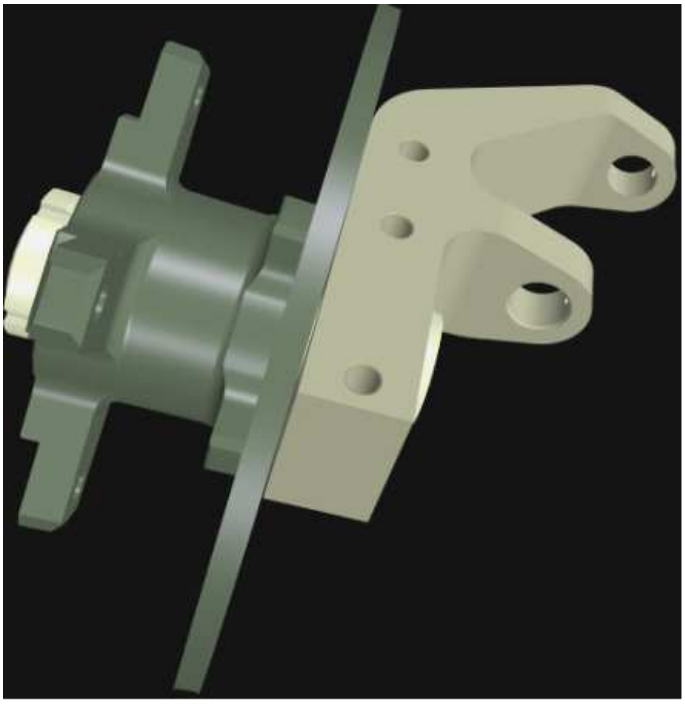


Figure 16: Alpha_1 Brake Hub

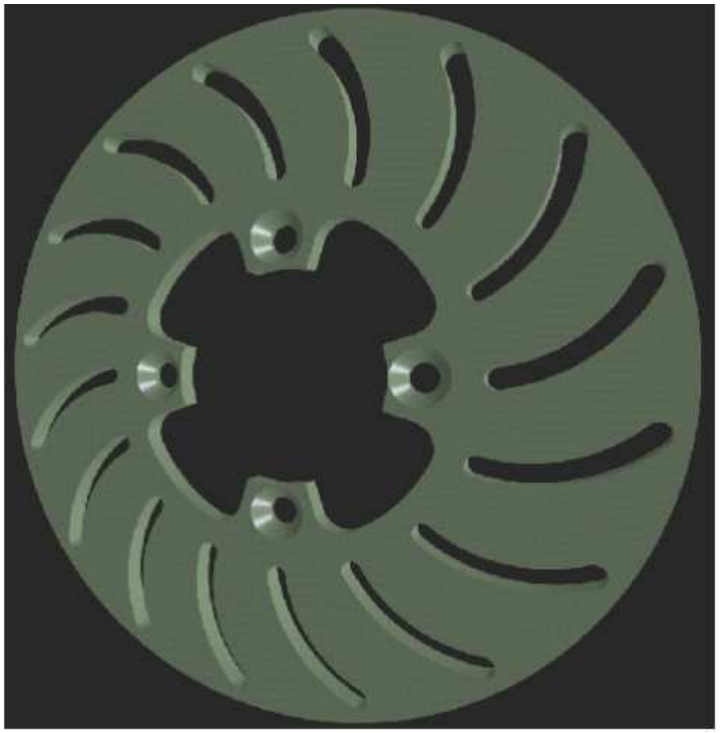


Figure 17: Alpha_1 Brake Rotor

Figure 18: Flower Trimmed Utah Teapot

Figure 19: UNC Trimmed Utah Teapot