View-dependent Adaptive Tessellation of Spline Surfaces *

Jatin Chhugani *Subodh Kumar*

Johns Hopkins University
Baltimore, MD 21218

Abstract

We present a novel algorithm for dynamic adaptive triangulation of spline surfaces. The algorithm is dynamic because it adjusts the level of detail of the triangulation at each frame. It is adaptive because it samples each surface patch more densely in regions of high curvature and less densely in regions of low curvature. The two have not been combined before. Our algorithm pre-computes a prioritized list of important samples on the surface. At rendering time, it adds these points in the specified order to the triangulation. Once the pre-computed points are exhausted and even more detail is required on some region of the patch, additional samples, now uniformly spaced, are added to the triangulation. The algorithm works well in practice and has a low memory footprint.

CR Categories and Subject Descriptors: I.3.m [Computer Graphics]: Spline, Surface display, Adaptive tessellation, Levels of detail

1 Introduction

Surface models are used in applications ranging from CAD industry and medical visualization to entertainment industry. Interactive display of these models is required for a variety of simulations. Although, hardware triangle rendering speed has increased to millions of triangles per second, the geometric detail of surface models has increased faster. Hence, software techniques to reduce the number of polygons sent to the graphics hardware remain popular.

Splines, especially of the Non-Uniform Rational B-Spline (NURBS) and Bezier forms [12], remain the representation of choice for a large class of application. Automobiles, submarines, airplanes, etc. are commonly modeled as splines. Hence, spline rendering has been an active area of research for over two decades. Ray tracing [33, 16, 25], pixel level subdivision [4, 30, 29], and scan-line based algorithms [32, 3, 22] have been used in the past to render surfaces. However, none of these are quite efficient on current graphics systems. In order to exploit fast triangle rendering hardware, research in the last decade has focussed on generating polygonal approximations of surfaces using uniform tessellation [28, 1, 24, 26, 20, 21] or adaptive tessellation [13, 31, 17]. The adaptive tessellation algorithms are intended for off-line static triangulation of models. A one-time *static* tessellation (in sufficient detail) of many real-world models, would require hundreds of million of triangles [2]. Schemes based on view-dependent mesh simplification [7, 34, 14, 15, 9] are commonly used to improve the rendering speed on any given graphics system. Typically, these require management of large amounts of data and are still unable to generate more detail than the initial tessellation. Note that these methods first generate a large number of triangles and subsequently reduce the count. We call these 'backward' techniques. 'Forward' techniques, on the other hand, perform view-dependent tessellation [28, 1, 20, 21]. The advantage of the forward technique lies in its ability for arbitrary precision. However, considerable time must be allocated to evaluate new samples at each frame and to update the triangulation. As a result, only simple surface sampling algorithms, e.g., uniform domain sampling, are used, thus causing over-tessellation for many areas of the model. (We have done an empirical test of the variance of curvature across a Bézier patch and found it to be several hundred times the mean in most cases.) The backward techniques, on the other hand, are able to utilize significant resources in a preprocessing step to limit dense sampling to areas of high curvature. At run-time, only the traversal of a data structure, which stores all sample points and connectivities, is required. The disadvantage is the large storage requirement and an upper limit on the detail of the model. Furthermore, due to geometric constraints, using high detail in one part of the model may enforce high detail in other parts where such high detail may not be necessary [9].

We present a novel approach that combines the advantages of both methods in a unique way. We have been able to factor out sufficient computation to the pre-processing stage to make interactive adaptive view-dependent spline tessellation possible. This method generates detail only where necessary thus reducing the total polygon count, has low memory overhead, allows arbitrary detail and still is simple to implement. In addition, the same basic methodology can also be used to improve the rendering of other classes of smooth surfaces that may be parametrized. Our algorithm starts by precomputing a good[1] object-space sampling for each surface element (in our implementation, a Bézier patch). It stores these domain points in a topologically sorted order so the points that reduce more the deviation between the resulting triangles and the surface appear first. During rendering, the algorithm determines the samples that must be added or deleted to minimally achieve a user-specified screen-space deviation. It maintains a Delaunay triangulation [5, 8, 23, 10, 11] of the (two dimensional) domain samples, incrementally adding and deleting the desired samples. The triangles generated by mapping the domain samples to the surface form the approximation and are sent to the rendering hardware. If detail beyond the precomputed set of points is required, additional points are generated by uniformly sampling each domain triangle. Note that once sufficient points are added in areas of high detail, each resulting triangular patch region is uniform enough. We avoid tessellation-cracks by choosing the same samples on both patches adjacent to any boundary curve. By speeding up the run-time sampling test and reducing the number of generated triangles (for a given deviation error), we are able to achieve a speed-up of about 2-4 over previous spline rendering methods [20, 21].

*supported in part by NSF CAREER award CCR-9733827 and ERC award EEC-9731748

[1] Our measure of goodness is the geometric deviation. Arguably overall deviation is important for rendering, shadows and other analyses [6].

1.1 Mathematical background

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

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

where the Bernstein function $\mathcal{B}_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$

The normal direction is given by $\mathbf{N}(u, v) = \mathbf{F}_u \times \mathbf{F}_v$where $\mathbf{F}_u$ and $\mathbf{F}_v$ are the partial derivatives.

The Delaunay triangulation of points on a plane has the property that the circumcircle of no triangle contains a point other than its three vertices. This property avoids long and skinny triangles. In fact, Delaunay triangulation produces the maximum possible smallest-internal angle of any triangle.

1.2 Organization

In the rest of this paper, we assume familiarity with NURBS and Bézier surfaces and Delaunay triangulation. We interchangeably use the terms triangle and triangulation for the domain points as well as the surface points. The meaning will be implicit in the context. We start with the details of pre-computed sampling in Section 2 and then present the run-time algorithm. Section 3 describes our implementation and reports its performance. Finally, conclusions are drawn and future directions listed in Section 4.

2 Algorithm

Our algorithm is based on a simple idea and has two main steps:

Pre-sampling: We compute a set of sample points on the domain. These sample points are 'good' in the sense that they are locally best at reducing the deviation of the resulting triangular approximation from the surface. These points also store the deviation of the approximation after the points are added. This information is used during rendering to decide which points to add.

View-dependent triangulation: At the rendering time, we start with a hierarchical organization of the pre-computed domain samples for each patch. We maintain a running Delaunay triangulation of the subset required for each frame. Based on the viewing parameters, we choose the samples that may be deleted and others that must be added to insure that a user-specified deviation in screen space is not exceeded. We incrementally update the Delaunay triangulation. If more samples than have been pre-computed are needed, we resort to uniform sampling [20] of the domain to add more samples.

One challenge is to decide which samples to pre-compute and in what order to add or delete them to maintain coherence in triangulation.

2.1 Pre-sampling

In principle, at the rendering time we need to find:

- which regions of a patch deviate more from the surface than desired and which points, if added to the triangulation, would reduce the deviation to the desired value.
- which points, if deleted, will leave the resulting triangulation still close to the surface.

The optimal answer seems untractable. Even reducing the choice to a pre-selected set of points, while possible in our framework, keeps the sampling and triangulation time long: there are too many combinations of points to consider. We instead use a heuristic. For each patch, we compute the sorted list of samples as follows:

1. Start with a minimal sample set (e.g., the four corners of the domain). These points are always included in the approximation.
2. Generate the Delaunay triangulation of the minimal set. This is our first approximation.
3. While the deviation of the surface from the approximation is greater than a user specified tolerance Δ_o:
   - Find the point on the surface that is the farthest from the current approximation (see section 2.2).
   - Append the corresponding domain point to the list of pre-samples. Store the resulting deviation with the point.
   - Insert the point in the Delaunay triangulation updating the current approximation. (see Section 2.3).

At the end of the process, we have an ordered list, S, of domain samples for each patch and the object-space deviation, D_i, between the approximation and the surface if all points $S_j, j \leq i$, are added to the triangulation for any i. Note that $D_{i+1} < D_i$[2] and $D_{|S|} \leq \Delta_o$. Thus, given a deviation bound d_o, we can find the prefix of S that generates an approximation with deviation less than d_o.

2.2 Deviation computation

In order to find the point on the surface with the maximum deviation from the current approximation, we compute the maximum deviation of each triangle and use their maxima. Note that we need to update the deviation of only the new triangles after a Delaunay update. To find the maximum deviation corresponding to a domain triangle $t = (p_1, p_2, p_3)$, $p_i = (u_i, v_i)$, of Bézier surface patch B:

- Compute, n, the unit normal to the plane of the triangle $T = B(p_1), B(p_2), B(p_3)$.
- Find the maximum distance, $|B(p)t_p|$, where t_p is the projection (along n) of $B(p)$ on T, $\forall p \in t$. We use the Powell's method [27] to find the maxima, with the centroid of t as the starting guess.

In case a triangle is degenerate and lies on a straight line, we compute the unit vector parallel to it and the objective function simply computes the cross product of the displacement vector and this unit vector. In case all the three vertices of the approximating triangle coincide, the function returns the norm of the displacement vector.

Note that a fixed order among the samples of a patch is acceptable for static tessellation but the relative "importance" of points is view-dependent and can vary for patches large on screen. We maintain a hierarchical partitioning of domains (a two level hierarchy is usually enough) into quads. As the projection of patch grows, we switch to a finer partition. We sample each sub-domain independently using the subsequence of patch pre-samples that lie on that sub-domain. The deviation for each domain is based on its own scale-factor (described in the next section).

[2]Technically, deviation could increase on adding a new point for some degenerate patches. Even in such cases, adding a sequence of samples always leads to a lowering of deviation. We add or delete such samples together and assign a single index to them.

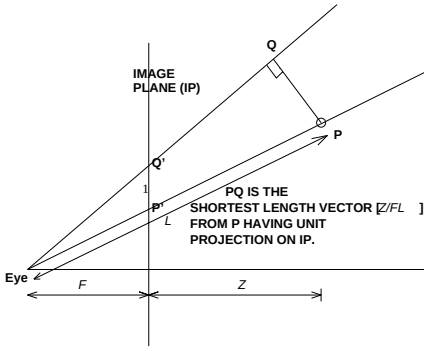

Figure 1 Scaling of a unit vector in screen-space

2.3 Delaunay Triangulation

We use the algorithm by [10, 11] to perform incremental Delaunay triangulation. The algorithm is outlined below.

To insert a given point, p in a given triangulation Δ, we need to find, t_i, the triangles whose circumcircle contain p. We perform this search by walking. We retain the centroid, C, of Δ, which serves as the starting point of this walk. Searching proceeds as follows:

- Find a consecutive pair of edges, (e_{j1}, e_{j2}) of triangle t_j such that p lies in the positive half-space of (e_{j1} and e_{j2}).
- If p is contained in the circumcircle of t_j, find one of the intercepted triangles.
- Otherwise, continue at another edge pair adjacent to one of the vertices of t_i,

If no intercepted triangle is found, we simply connect p to the visible vertices of the convex hull of current triangulation. If an intercepted triangle t is found, we perform a breadth first traversal of the adjacency graph to discover t_i, all intercepted triangles, whose overall boundary forms a star shaped polygon, P. We delete all t_i and add an edge between p and all vertices of P. The centroid vertex is also updated at this stage. Storing the centroid vertex reduces the expected search time to $O(n^{0.5})$ though the worst case remains linear. The insertion procedure takes $O(k)$ time where k is the number of edges in P.

To delete a point, p, we first locate it in the triangulation Δ. We delete all triangles adjacent to p thus obtaining again a star shaped polygon, P. If P has more than three edges, we insert each edge of P in a priority queue. The priority value for an edge $e_i = r_i^2 - (|c_i - p|)^2$ where c_i is the circumcenter of the triangle formed by edge e_i and e_{i+1} (addition being modulo the number of edges in P) and r_i is its radius. We then add triangles to Δ in the following order:

- Choose the edge, e_i with the minimum priority and add the triangle formed by e_i and e_{i+1} to Δ
- Replace e_i and e_{i+1} from P and add the new edge.
- Repeat until P is a triangle.

In the end, we update the centroid vertex. The expected search and actual deletion times are similar to the one's obtained during insertion.

2.4 Render-time sample selection

Our rendering algorithm allows the users to bound the maximum geometric deviation of the triangular approximation. This bound may be specified in pixel units. If the user-specified screen-space bound for a patch B is d_s, object-space deviation required for the approximation is $d_o = s(B)d_s$, where $s(B)$ is the scale-factor for patch B. The scale-factor of a point p in object space is the length of the smallest vector anchored at p that projects a unit vector in screen-space (see Figure 1).

Different patches, indeed different points of a patch, may have different scale-factors. We use an octree based spatial partitioning of space. For all patches contained in a sufficiently small partition, we use the same scale-factor. Typically partitions close to the viewpoint are more refined than those further away, as the scale factor close to the viewpoint varies faster. The sample selection proceeds as follows:

1. Start with the octree cubes used in the previous frame. We call a cube *terminal* if the scale-factor of the four corners of a cube differ by less than δS, a user-specified tolerance. If a cube is non-terminal, we subdivide it. Otherwise, if a cube's parent is terminal, we recursively use the parent. If δS is chosen to be $\frac{1}{d_s}$, the approximation does not under-deviate by more than a pixel.
2. The scale-factor of a cube is the smallest of the scale-factors of its corners. For each patch completely contained in a terminal cube, C with scale-factor $s(C)$, we choose all samples $S_i, i <= j$, such that the associated deviation $D_j < s(C)$ and $D_{j-1} > s(C)$.
3. Suppose the value of j in previous frame is j_{prev} and that in the current frame is j_{curr}. If $j_{curr} > j_{prev}$, we add samples $S_{j_{prev+1}} .. S_{j_{curr}}$ to the Delaunay triangulation, otherwise we delete $S_{j_{curr+1}} .. S_{j_{prev}}$.
4. If a patch lies in more than one terminal cube that are all adjacent to each other, we assign to the patch, the minimum scale-factor of those cubes.
5. For larger patches, however, we do need to use different scale-factor in different regions of the patch. We subdivide the domain of patches that span terminal cubes that are not adjacent to each other. We use the scaling algorithm described above to each subdomain k to compute the required object-space deviation d_{o_k}. For each subdomain, we find the subset of pre-computed samples, S_{k_i}, of the patch that lie in that subdomain and apply the algorithm in step 3.

Recall, that we pre-compute one linear ordered list of samples per patch. The Delaunay triangulation, and hence the choice of samples, depends on this order. In step 5 above, however, we may add sample S_j of the patch without adding another sample S_k for a $k < j$, if S_j and S_k lie in different sub-domains. As a result, we could theoretically obtain an approximation that deviates more than d_s from the surface. On the other hand, in Delaunay triangulation only neighboring points are connected by an edge in practice. Thus the triangulation of the samples of the subdomain is mostly a subset of the triangulation of the full patch. As a result, we would still get mostly the same samples had we chosen to pre-sample each subdomain separately, thus respecting d_s.

2.5 Uniform dynamic sampling

If the required deviation $d_o < D_n$ for a patch with n precomputed samples, we compute more samples in a lazy fashion. We do this render-time sampling using a variant of the uniform sampling method of Kumar et al. [19]. We store, for each triangle t of patch B, $t_u = \frac{1}{\sqrt{2|B_{uu_t}|_{\max}}}$ and $t_v = \frac{1}{\sqrt{2|B_{vv_t}|_{\max}}}$. (The maxima of

partial derivatives are substituted in practice by the maxima among the three corners of t.) At rendering time, the uniform step sizes in u and v directions are given respectively by $K_d t_u \sqrt{s(B) d_s}$ and $K_d t_v \sqrt{s(B) d_s}$, with constant K_d normally set to 1 (see [18] for details). Note that, this reduces to two multiplications per triangle in addition to one multiplication and one square root per patch.

2.6 Crack prevention

If we independently tessellate two adjacent patches, we could choose different samples on their common boundary. This results in a crack in the resulting approximation (Color Plate A). In our approach, we want to keep the tessellation of adjacent patches independent of each other to facilitate easy parallelization. We assume that the object space representation of boundary curves are the same. We pre-compute the sampling of each boundary curve separately from the interior. Ensuring that the same samples are used at the boundary of both patches adjacent to it eliminates cracks (Color Plate A). Bézier curve pre-sampling is a special case of Bézier patch pre-sampling and we refer the reader to Section 2.2 for details. Once each boundary curve is pre-sampled, we modify the interior patch sampling by deleting samples too close to the boundary. If a sample $S = (u, v)$ is closer to the boundary than an ϵ, i.e., u or v are either less than ϵ or greater than $1 - \epsilon$, we add it to the interior sample list only if: S is closer to curve samples S_i and S_{i+1} than $\frac{1}{2}|S_i S_{i+1}|$, where S_i and S_{i+1} are the two closest samples to S on the boundary curve.

Note that the role of ϵ is primarily to reduce the number of times the closest points are computed.

2.7 Discussion

Our algorithm combines pre-processing with render-time triangulation update. One attractive feature of this algorithm is its ability to trade-off storage of pre-computed data against render-time CPU usage. This can be effectively exploited to tune the rendering on different CPUs. We store the position (u, v) and deviation, D, for each pre-computed sample point. In addition the following pre-computed data may be retained and reused during rendering:

- The three spatial coordinates of the corresponding surface points and the three normal coordinates of the sample may be stored.
- The star polygon P that must be re-triangulated when updating the Delaunay triangulation may be stored to triangulation update cost. Note that the order in which points get added remains the same until a domain is split, which occurs only occasionally.
- The entire update to the triangles may be stored.

Another advantage of our algorithm is its application to model compression. Note that we need to store only four floating point values per sample. Moreover, the actual domain position of each sample is need not be very precise. In fact, using a single byte for u and another for v is enough for high quality tessellation. In addition, two bytes are enough to represent D (appropriately scaled). Thus, we may represent a spline patch with its original control points, some derivative values and $4n$ additional bytes, with n typically being less than 60 for bi-cubic patches. In return, we avoid significant tessellation cost (see Section 3).

3 Implementation and results

We have implemented our algorithm and tested it on a variety of models. All timings reported in this paper are from an Onyx2 with a 195 MHz R10000 InfiniteReality graphics. Our experiments consisted of viewing a variety of model from many viewpoints collected from a simulated walkthrough. We first compare the number of triangles produced by our algorithm with that by a uniform tessellation algorithm [20] in Table 1. This, combined with faster sample selection, results in the overall speedup, which is more pronounced for large models. Plate B shows the difference in uniformly sampled teapot and adaptively sampled teapot. Notice that the difference is significant in the lid and spout, which comprise patches with large curvature variance. Plate C shows our initial pre-samples. We set the initial stopping deviation to 0.8 pixels with a scale-factor of 1. Since our deviation bound is usually one pixel or more, pre-computed samples are sufficient for most views.

Model	Num Patches	Avg num of Tris/frame	
		Ours	Kumar et al. [20]
Goblet	72	2302	6744
Coke	330	3994	5488
Dragon	5079	16804	42115
Garden	38646	83733	122360

Table 1 The comparison of the number of triangles produced by our algorithm *vs* [20] for the same screen-space deviation of two pixels.

Model	Num Samples pre-computed	Pre-process time	If Vertex/Normal are pre-computed
Goblet	6,500	48.78 Seconds	0.41 Seconds
Coke	23,109	58 Seconds	.45 Seconds
Dragon	864,610	31 minutes	38 Seconds
Garden	838,152	13 minutes	14.1 Seconds

Table 3 Pre-sampling performance

In Table 2 we show the render-time behavior of our algorithm. The number of samples that need to be added to or deleted from the triangulation is less than 1% of the average triangle count and the overhead of Delaunay triangulation is less than 10% of overall time. Thus, the per-frame operation is quite efficient. In addition, uniform tessellation is needed for less than 0.15% of the patches on average.

Table 3 reports the pre-processing time. In addition, it lists the time it would take to pre-compute the three dimensional position and normal values. Note that the number of pre-samples is rather large, but we keep only eight bytes per sample, including the multiplication factor needed for uniform sampling. Color plates D and E show the levels of detail. Note the range of high detail in plate E. 98% of the samples in the zoomed up view are pre-computed.

4 Conclusion and acknowledgements

We have presented the first *view-dependent* adaptive spline tessellation algorithm. It can be updated at speeds that support interactive display of tens of thousands of surface patches on current graphics systems. This algorithm gracefully combines offline sampling and run-time triangulation to achieve fast tessellation with low triangle count and a small memory footprint. This provides an alternative scheme to view dependent mesh simplification. Tessellation based methods have better local control on adjustment of detail and allow more convenient maintenance of texture coordinates. The overhead of incremental triangulation, as we have shown, is very small.

We are working to extend our method to trimmed splines and subdi-

Model	Number of sample updates/ frame	Time to update Triangulation	Overall rendering frame-rate	Rendering rate of [20]
Goblet	24	.11 ms	72	42
Coke	53	.28 ms	65	23
Dragon	653	9 ms	18	5
Garden	1485	16.2ms	6	1.7

Table 2 Run-time behavior of our algorithm

vision surfaces. We have used Delaunay triangulation primarily due to the availability of efficient implementation of an incremental algorithm. More investigation is needed into other competing triangulation schemes. Furthermore, a triangulation algorithm that directly produces triangle strips would be immensely useful in speeding up the rendering of resulting models.

We thank Shankar Krishnan for his maxima finding code. Models were courtesy of Lifeng Wang, the modeling group at University of British Columbia and XingXing Graphics Co. (Garden), David Forsey (Dragon) and Alpha 1 system (Soda can and Goblet). Finally, we thank Jonathan Cohen for insightful discussions and the reviewers for their helpful comments.

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] D. Aliaga and J. Cohen et al. Mmr: An integrated massive model rendering system using geometric and image-based acceleration. In *Proc. Symposium on Interactive 3D Graphics*, pages 101–106, Atlanta, GA, 1999.

[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] L. P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[6] J. Cohen, M. Olano, and D. Manocha. Appearance preserving simplification. In *Proc. ACM SIGGRAPH*, 1998.

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

[8] L. De Floriani and E. Puppo. An on-line algorithm for constrained Delaunay triangulation. *Comput. Vision Graph. Image Process.*, 54:290–300, 1992.

[9] L. DeFloriani, P. Magillo, and E. Pupo. Building and traversing a surface at variable resolution. In *Proc. IEEE Visualization*, pages 103–110, 1997.

[10] O. Devillers. Improved incremental randomised delaunay triangulation. In *Proc. 14th Annual ACM Sympos. Comput. Geom.*, pages 106–115, 1998.

[11] O. Devillers. On deletion in delaunay triangulation. In *Proc. 15th Annual ACM Sympos. Comput. Geom.*, pages 181–188, 1999.

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

[13] D. Filip. Adaptive subdivision algorithms for a set of Bézier triangles. *Computer-Aided Design*, 18(2):74–78, 1986.

[14] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proc. ACM SIGGRAPH*, pages 209–216, 1997.

[15] H. Hoppe. View dependent refinement of progressive meshes. In *Proc. ACM SIGGRAPH*, 1997.

[16] J. Kajiya. Ray tracing parametric patches. *ACM Computer Graphics*, 6(3):245–254, 1982. (SIGGRAPH Proceedings).

[17] R. Klein. Linear approximation of trimmed surfaces. In *The Mathematics of Surfaces VI*. Springer Verlag, 1994.

[18] S. Kumar. *Interactive Rendering of Parametric Spline Surfaces*. PhD thesis, University of North Carolina, Department of Computer Science, 1996. Also available as UNC Technical Report TR96-039.

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

[20] S. Kumar, D. Manocha, and A. Lastra. Interactive display of large NURBS models. *IEEE Transactions on Visualization and Computer Graphics*, 2(4):323–336, Dec 1996.

[21] S. Kumar, D. Manocha, H. Zhang, and K. Hoff. Accelerated walkthrough of large spline models. In *Proc. Symposium on Interactive 3D Graphics*, pages 91–101, Providence, RI, 1997.

[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] Dani Lischinski. Incremental Delaunay triangulation. In Paul Heckbert, editor, *Graphics Gems IV*, pages 47–59. Academic Press, Boston, MA, 1994.

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

[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] L. Piegl and A. Richard. Tessellating trimmed NURBS surfaces. *Computer Aided Geometric Design*, 27(1):16–26, 1995.

[27] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge Univ. Press, 1993.

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

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

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

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

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

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

[34] J. Xia and A. Varshney. A dynamic view-dependent simplification for polygonal models. In *Proc. IEEE Visualization*, pages 327–334, San Fransisco, CA, 1996.