To my parents

Subodh Kumar. Interactive Rendering of Parametric Spline Surfaces
(Under the direction of Dinesh Manocha.)

ABSTRACT

This dissertation presents techniques for fast rendering of parametric spline surfaces. It presents algorithms and data structures needed to support the thesis that real-time display of surfaces (represented parametrically) is indeed possible on current graphics systems that are optimized to display triangles. It analyzes the sources of bottleneck in surface rendering and derives techniques to display tens of thousands of Bézier surfaces, $10 - 20$ times a second, by efficiently utilizing the graphics hardware.

In a more general framework, this work demonstrates the effectiveness of using higher-order surfaces, as opposed to polygons. Analytic representation of surfaces retains information that is often lost in a static translation to polygons. We meaningfully use this analytic information to obtain better images than those generated from purely polygonal models. On the other hand, since current graphics systems are optimized for displaying triangles, we perform on-line triangulation to still be able to make effective use of the hardware capabilities of current systems.

For specificity, we concentrate on trimmed Non-Uniform Rational B-Splines (NURBS) surfaces. The trimming curves that define the boundary of the surfaces may be NURBS or piecewise linear. The surfaces are off-line transformed to Bernstein basis and stored as trimmed Bézier patches. At rendering time, each Bézier patch is tessellated into triangles appropriate for the current rendering of the scene from the user's view point.

The overall algorithm first culls away the patches facing away from the user, or lying outside the view-frustum. The next step determines the sampling density required for each on-screen patch and its trimming curves, for a smooth display with Gouraud or Phong shading. Next, the sample points on the surface, and their normals, are evaluated and efficiently triangulated into strips. Finally the triangle strips are sent to the graphics pipeline. The algorithm exploits temporal coherence by performing only incremental triangulation of patches at each frame. The triangles used in a frame are cached in memory. For the next frame to be displayed, only a

small *update of detail* is performed to maintain a smooth image. This dissertation presents efficient techniques to prevent tessellation cracks across surface boundaries. It also discusses issues in parallel implementation of the tessellation algorithm on multi-processor systems.

This dissertation describes SPEED, a portable implementation of our algorithms, and reports its performance on some well-known graphics systems. While our algorithm is primarily designed for static models, many of the techniques are applicable to dynamic scenes as well. The performance of our prototype Bézier patch rendering system has been quite encouraging. We are able to render about five thousand Bézier patches, more than ten times a second, on a Silicon Graphics Onyx with four 200MHz R4400 CPUs and a Reality Engine II graphics pipeline. This is more than an order of magnitude improvement over the state of the art (e.g. SGI GL library). The major factors of this improvement include efficient visibility computation, tighter bound computation, efficient trimming, incremental triangulation and parallelism. Our results demonstrate that we can indeed generate high-fidelity images of large engineering models on current graphics systems, and obtain interactive rendering performance at the same time.

Acknowledgments

I wish to thank my advisor, Dinesh Manocha, for his excellent guidance and continued support and for his willingness to find time for me in his busy schedule. My gratitude also to the other members of my committee: Fred Brooks for teaching me to pursue real problems, Henry Fuchs for bringing out the best in me, Nick England for asking the right questions, Anselmo Lastra for close guidance, support and his helpful hand, and Turner Whitted for acting as my walking reference library.

I would like to thank Elaine Cohen, Russ Fish and David Johnson at the university of Utah for insightful discussions and the Alpha1 models. Thanks also to the Ken Fast, Greg Angelini and Jim Boudreaux at Electric Boat and to members of the Pixel-Planes, Modeling and Walkthrough groups at UNC, especially Shankar Krishnan and Atul Narkhede for generating a wealth of models for me to test my results on, and to the PIs for supporting my work. Thanks to David Ellsworth without whose help I could never have completed the Pixel-Planes 5 implementation. Thanks to Atul Narkhede for his polygon triangulation code, to Kyle Wilson for his subsequent upgrade of Atul's code and to Chun-Fa Chang for his implementation of the load-balancing controller.

This work was supported in part by ARPA ISTO Order No. A410, NSF Grant No. MIP-9306208, NSF Prime Contract 8920219, an Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, NSF Grants CCR-9319957 and CCR-9625217, ONR Contract N00014-94-1-0738, ARPA Contract DABT63-93-C-0048 and NSF/ARPA Center for Computer Graphics and Scientific Visualization.

Contents

List of Figures

List of Tables

Chapter I

Preview

1.1 Introduction

In many applications, like engineering design of large equipment, and walkthroughs of ships and submarines, models are traditionally represented as millions of polygons [Bro96]. It is important to be able to visualize these models to speed up their design, prototyping and evaluation. While graphics systems have reached the capability to render millions of transformed, shaded and z-buffered triangles per second [Fea89, MEP92, Ake93], they are not able to render such complex and detailed engineering models. For example, a high-end commercial graphics system, like an SGI Onyx with a Reality Engine II graphics accelerator and 2 Gigabytes of main memory, is not able to transform, Gouraud-shade, z-buffer and display the storage and handling room of a notional submarine [1] (see Figure 1.1) even once per second. Furthermore, empirical evidence shows that model sizes tend to grow ten fold every two years while the rendering speed only about doubles [Wal96]. These large models not only throttle the rendering pipeline, but also encounter system memory limitations. Researchers like Clark [Cla76] realized quite early that not all triangles need be sent to the graphics sub-system every frame. He advocated hierarchically grouping objects into clusters and spending only as much effort rendering each cluster as is needed for a satisfactory

[1] The Sub model consists of 900,000 triangles.

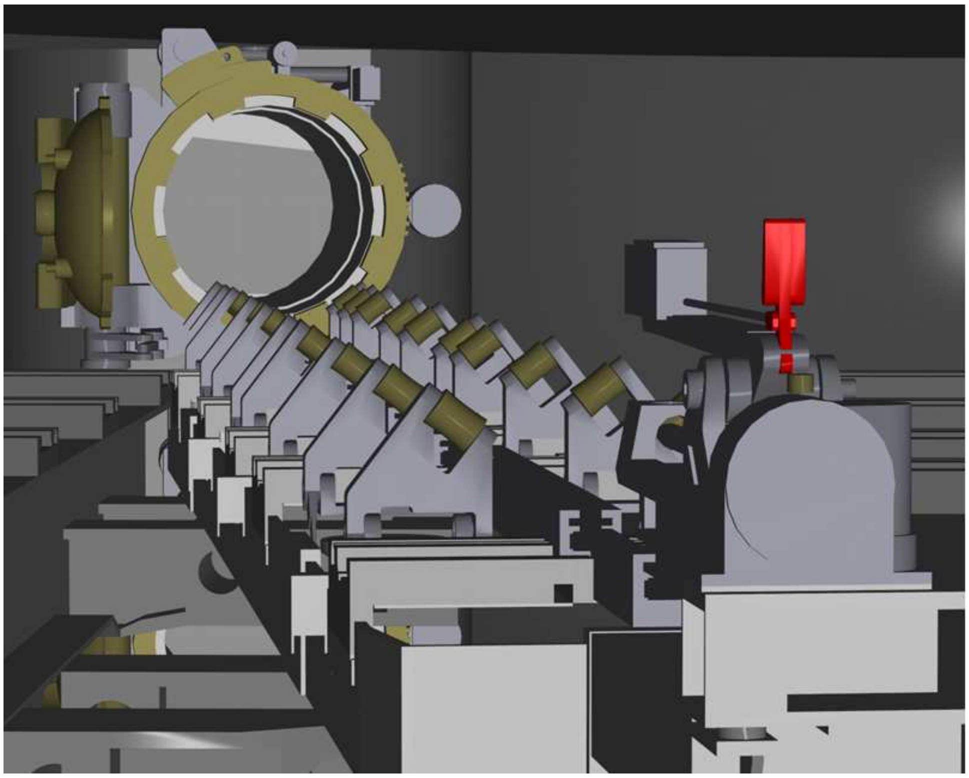

Figure 1.1: Sub: A Submarine storage and handling room: 900K Polygons

image of it. For example, objects far from the viewer may be rendered at a coarse detail thus requiring less resources.

The need to navigate through increasingly complex models, for simulation based design or design evaluation, has recently motivated considerable research in software techniques to speed up rendering. The main thrust of these techniques is to reduce the number of polygons passed to the pipeline —

- **Visibility computation**: For large models, the number of off-screen polygons is likely to be high. Spatial partitioning with the help of data structures like Octrees [Hun78] and BSP trees [FKN80] are regularly used by rendering systems to reduce the number of off-screen polygons passed to the graphics pipe. Recently Teller et al. [TS91] and Luebke et al. [LG95] described techniques to subdivide models into cells and portals and only render cells visible through these portals. Funkhouser et al. [FS93] use similar computation to fetch visible (and close to visible) parts of a model into memory, if the complete model does not fit into the main memory. These techniques are well suited for architectural environments but are not as effective for engineering models that are not easily partitioned into cells and portals.

- **Model reduction**: Graphics literature contains description of a number of techniques to reduce the number of polygons used to represent a model [DZ81, Tur92, SZL92, DLW93, HDD$^+$93, Var94]. In general, these methods generate a sequence of discrete approximations of parts of a model, each at a coarser level of detail from the previous one. At rendering time, the system picks and displays the appropriate version of each part based on its distance from the viewer. Switching smoothly between versions is a tough problem for these methods. The methods of progressive meshes [Hop96] alleviates this problem by varying detail more gradually. However, there remain difficult problems like handling degenerate models and efficiently switching detail to the desired level.

- **Textures:** Recently there has been an increase in support for rendering textured polygons on current graphics systems. Painting the image of parts of geometry as texture on a large polygon results in significant reduction in polygon count and hence an increase in rendering speed. Due to coherence, the image does not change much over a small time-interval. Hence this image, with a small warp to account for the change of view point, is re-displayed as texture [Ali96, SLS$^+$96]. The system must choose the part of the geometry to re-display using textures. This choice can significantly impact both rendering speed and realism. The maintenance of smoothness at the boundary between real geometry and texture is also a tough problem.

- **Image-based rendering:** Taking the concept of images further, some recent techniques dispense with geometry altogether and represent models as images [Che95, MB95, LH96, GGSC96]. The basic principle of image-based rendering is to generate multiple views from a small number of photographs. Such computation is quite slow on current graphics systems. Moreover, design of new geometry and its physical analysis is precluded by these methods due to the lack of exact geometry. The application of these techniques to computer aided design and manufacturing (CAD/CAM) remain to be proven.

In many applications involving CAD/CAM, virtual reality, animation and biomedical engineering, the object models are designed in terms of tensor product trimmed non-uniform rational B-spline (NURBS) surfaces, which can exactly represent many geometries. This class includes Bézier surfaces and other rational parametric surfaces like tensor product and triangular patches. Large-scale models of shapes of automobiles, submarines, airplanes, etc. typically require tens of thousands of such surfaces to represent them. Constructive solid geometry, surface fitting over scattered data or surface reconstruction are some other techniques that produce many thousands of NURBS surfaces. NURBS offer the benefit of exact geometric representation and require significantly less memory than polygonal or image representation. Exact geometry ensures that we are able to render images at as high levels of detail as allowed by pixel discretization of images. NURBS are a popular tool to model surfaces and have found their way into graphics standards like PHIGS+ and OpenGL. Unfortunately, current renderers of these sculptured models on commercial graphics systems, while faster than ever before, are not able to render them in real-time for applications requiring walkthroughs and immersive design. Most systems perform a one time, high-resolution approximation of the NURBS in terms of polygons, thus giving rise to the enormous database sizes alluded to earlier, and all the associated problems. This dissertation seeks to change that by speeding up surface rendering.

1.2 Surface rendering techniques

In this section we briefly review the current techniques to render curved surfaces. Due to its importance in design, curved surface rendering has been an active area of research for more than two decades. Forrest [For79] performed one of the earliest studies on benefits of shaded displays of curved surfaces. Forrest claims that simple wire-frame renderings do not lend themselves well to effective visualization of a surface, even with hidden line elimination and perspective projection. Forrest con-

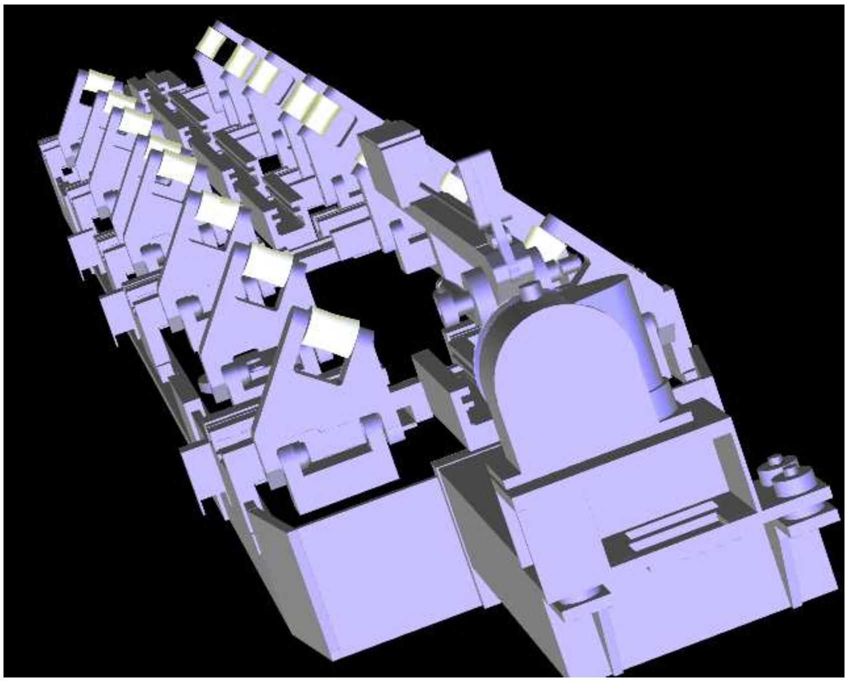

Figure 1.2: Pivot (from the Sub model): 4100 Bézier patches

jectures that real-time manipulation of smoothly shaded surfaces is too expensive to realize and advocates the usage of surface snap-shots from certain view points overlaid with wire-frames that are planar sections of the surface and garnished with a Gaussian curvature based shading. A number of techniques to display smoothly shaded surfaces have been devised in the past. We can broadly classify these techniques into four categories:

1.2.1 Pixel level subdivision

Catmull [Cat74] presents one of the first algorithms for displaying surfaces up to the accuracy of the resolution of display. This algorithm subdivides a surface into smaller and smaller sub-patches until each sub-patch is no bigger than the size of one screen pixel. The pixel is then assigned the color of that sub-patch. Catmull devised a fast subdivision algorithm for bicubic patches. While resulting in effective images, this method is quite slow due to the depth of subdivisions. The adaptive forward differencing of Shantz et al. [SL87, SC88] is similar in principle and handles trimming curves. It converts the NURBS basis to a forward difference basis and scales the difference up or down depending on the speed of parametric curves on the surface. Apart from having slow speed, this method also suffers from low accuracy, specially in case of high degree surfaces.

1.2.2 Scan-line display

Scan-line based algorithms for display of curved surfaces are faster than pixel level subdivision in general. These algorithms process the screen-pixels in scan-line order. For each scan-line, the intersections of the scan-plane with the surfaces are used as spans. Whitted [Whi78] advocates subdividing surfaces at silhouettes and in areas of high curvature. This decomposes surfaces into elements, each of which are free of silhouettes except at their boundaries and are relatively flat. He defines surface elements using four boundary curves and decomposes each curve into segments such that only one intersection may occur with any scan-line. Whitted uses Newton's iteration to compute the extrema of spans and interpolates the normal across a span for fast shading. Blinn [Bli78] does not pre-process the patches but performs Newton-Raphson iterations to compute all extrema along each of the screen coordinates. He also presents heuristics to handle singularities and many other special cases. Lane et al. [LCWB80] perform the scan-line intersection by recursively subdividing patches. If the convex hull of a patch intersects the current scan-line, the patch is subdivided into smaller sub-patches so that the convex hull of each sub-patch bounds it more tightly. Once the patches are flat enough, the convex hull almost coincides with the patch, and is used to compute the spans.

Most scan-line based methods exploit spatial coherence to speed up span computation. However, due to the inherent complexity of scan-line intersection, these algorithms are too slow for real-time display of large models.

1.2.3 Ray tracing

Whitted [Whi79] presents one of the earliest methods of producing photo-realistic images of bicubic surfaces using ray tracing. He uses bounding volumes to eliminate surfaces not intersected by a ray. If a ray intersects the volume bounding a surface, that surface is subdivided using Catmull's scheme [Cat74] and the process is repeated until the volume is small enough to be approximated as the point of intersection. Whitted also performs anti-aliasing by filtering in the areas of high luminance gradient. Kajiya [Kaj82] presents a numerical solution and reduces the ray patch intersection in real projective 3-dimensional space to intersection of algebraic curves in 2-dimensional complex space. It uses results from elimination theory (see Chapter II) to compute the points of intersection. Nishita et al.'s algorithm [NSK90] is subdivision based, but performs the surface subdivision using an operation called Bézier clipping. Bézier clipping eliminates large parts of a surface that do not intersect a ray. This technique is applicable to trimmed surfaces as well. Although ray tracing can generate high quality images, it can involve millions of ray intersections, each needing thousands of operations. As a result, these methods are too slow for interactive display.

1.2.4 Polygonal approximation

A number of algorithms for approximating curved surfaces with polygons are described in the literature [Cla79, KM84, FMM86, RHD89, Baj90, FK90, AES91, LC93, KS95]. As [Cat74], these methods subdivide the surfaces into smaller components, but only until each component is flat enough to be approximated by a polygon. (In this sense the algorithm of Lane et al. may also be classified as a polygonal approximation method.) Some apply the subdivision recursively, stopping when their flatness test is satisfied, while others first pick a number of sample points on the surface and then polygonize these points. Once the approximating polygons are computed, standard polygon rendering techniques are applicable. One common problem with these techniques are occurrence of cracks between adjacent patches if their triangular ap-

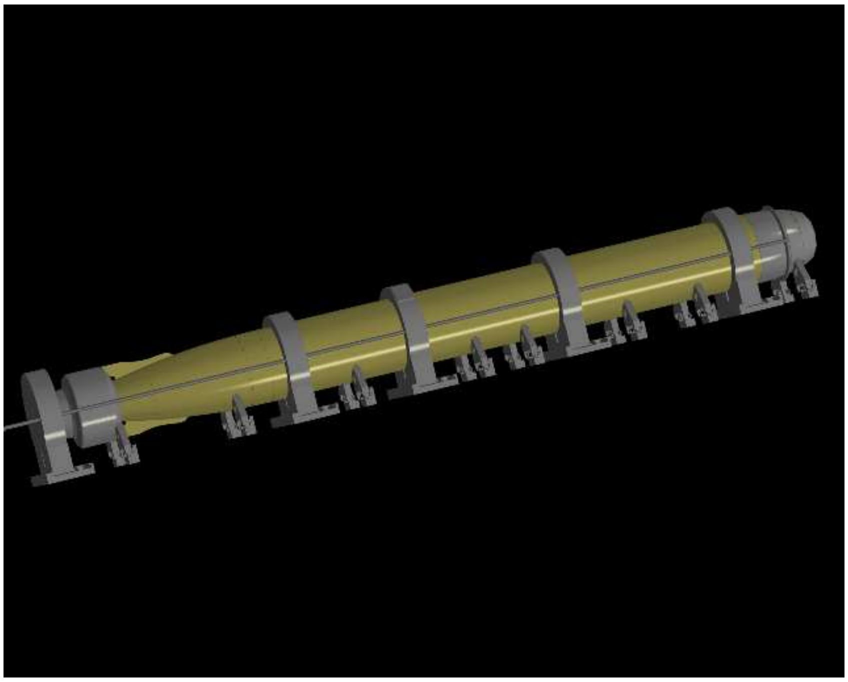

Figure 1.3: Shipping line (from the Sub model): 3400 Bézier patches

proximations do not match at the boundary. A common solution [Cla79, RHD89] to this problem is to stop subdividing a boundary if the adjacent boundary reaches its subdivision limit.

Most recent efforts for fast display of surfaces have concentrated on the approximation method. As opposed to the methods described earlier, these methods are able to make efficient use of recent advances in triangle rendering and hardware capabilities available on current graphics systems and hence come close to real-time rendering. Chapter VII presents a detailed comparison of our technique, which generates polygonal approximations of surfaces, with similar techniques.

1.3 Thesis statement

My thesis is:

> `We can perform` *interactive* `display of models consisting of many` *thousands of* `spline surfaces on` *current* `graphics systems.`

By interactive rendering we mean real-time display rates of at least 10 frames a second. We employ the native shading and interpolation model used by the underlying graphics system, typically Gouraud or Phong shading. The type of splines we are interested in are primarily NURBS, as used in computer aided design of mechanical parts. Current systems include Pixel-Planes 5, using which we are able to render Phong shaded images of models consisting of approximately thirty thousand tensor product rational trimmed Bézier surfaces at interactive rates. Some of the issues in such interactive rendering are described next.

1.4 Issues in polygonal approximations

Some of the main issues to consider when using polygonal approximation for NURBS are:

1. *NURBS form:* A NURBS surface is commonly represented in terms of the standard basis [Far93, Wat95] as a mesh of control points and a pair of knot vectors. A point on the surface can be computed using matrix multiplication, forward-differencing, the Oslo algorithm, or variants of the Cox de Boor algorithm [Far93]. On the other hand, we can use a different basis, e.g. the Bernstein basis, and represent the surface as a collection of Bézier patches. We must pick the representation that facilitates efficient evaluation of the points (geometry) and normals (for shading) on the surface. Since the Bernstein form can be evaluated more efficiently, we use the Bézier representation.

2. *Polygonal instancing:* We must decide when to generate the polygonal approximation for a given set of Bézier patches. One extreme is that we generate the approximation once and subsequently do not spend any frame time on the generation of any polygonal instances. This amounts to discarding the patch information, i.e. the exact geometry, and representing the model as polygons. To generate images that are an accurate representation of the original surface, we may need a large number of polygons, possibly overwhelming the polygon

rendering pipeline. Moreover, even this large number does not guarantee good quality from all view points (viewer positions). The other extreme is to spend a considerable amount of time at each frame determining the optimal number of polygons required to obtain a smooth image of the surface from the current view point. This method is suited for low-end graphics systems that do not have high triangle rendering performance. Optimally allocating time among determining the number of polygons needed, generating the desired polygons, and rendering those polygons is a tough problem. The tradeoffs are discussed in Chapter VI.

3. *Polygon Form:* While the freedom to output general polygons can simplify the surface tessellation algorithm, most graphics systems today are optimized for displaying triangles and not general polygons. Our method directly generates triangles.

4. *Visibility:* For an environment with a large number of Bézier patches, even if we display most patches at a relatively low detail, the number of resulting triangles can grow to be hundreds of thousands. At the same time, if we could determine the set of patches that lie off-screen, we can save a number of operations in tessellating these patches and displaying the resulting triangles. Our method performs the standard view-frustum visibility computation in addition to a new method of computing whether all of a Bézier patch is back-facing.

5. *Sampling size:* A tessellation algorithm must determine the required density of tessellation. Depending on the smoothness of the desired image, the algorithm must determine the number of triangles to generate. We use a combination of pre-processing and a low cost on-line projection to speed up the estimation of the required sampling size.

6. *Sampling distribution:* Sampling distribution is related to the sampling size. Not only must the algorithm determine the number of points but also the location of those points on the surface. An adaptive sampling algorithm may place

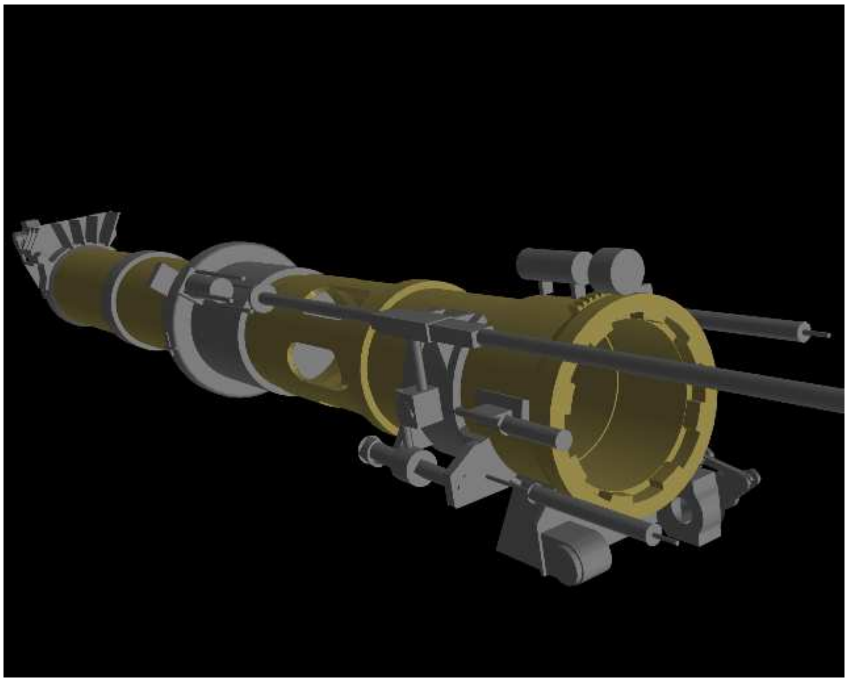

Figure 1.4: Torpedo tube (from the Sub model): 1200 Bézier patches

the points anywhere on the surface, whereas a uniform sampling algorithm places the points on a uniform grid. As shown in Chapter II, we can exploit the structure in grid lines to efficiently compute surface properties (i.e. points and normals). Uniform tessellation also makes the computation of sampling size easier. On the other hand, it requires relatively more samples for smooth images. In that sense, the choice of sample distribution is yet another handle to control the tradeoff introduced in Item 2 above. Our algorithm is based on uniform sampling, but uses restricted adaptive sampling to obtain the benefits of both techniques.

7. *Boundary matching:* When generating polygonal approximations, the sample points chosen at the boundary of two abutting surfaces must match up exactly. Otherwise cracks appear in images as shown in Figure 1.5. In particular, if parallel processing is used, and the two adjacent surfaces are allocated to different processors, they either must communicate or some information must be duplicated to ensure that the two tessellations match up. If we desire for no

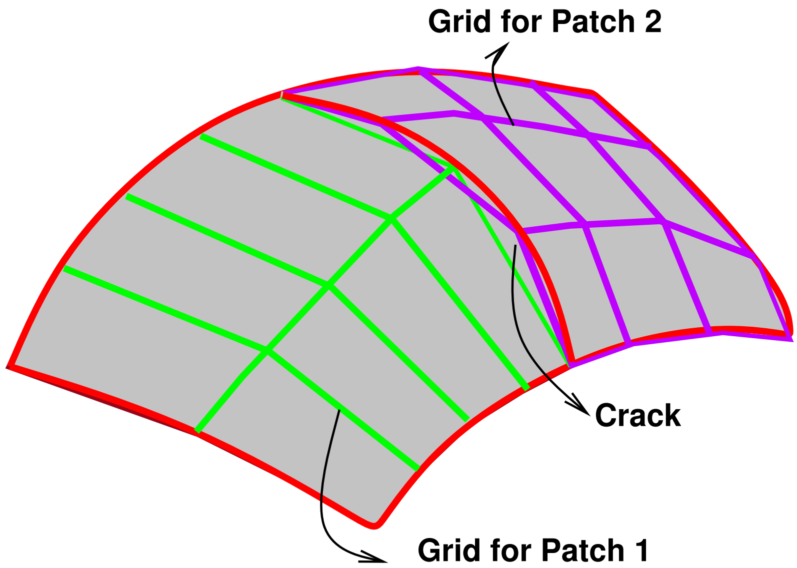

Figure 1.5: An exaggerated example of a crack due to polygonization

communication, we must also ensure that the method used to compute sample distribution is not confused if the two boundary curves have two different representations[2]. We use a dual representation of trimming curves to ensure that two adjacent surfaces have the same tessellation at their common boundary.

8. *Parallelism:* Current high-end graphics systems, in general, consist of multiple processors. In order to quickly compute approximations for large spline models, our overall algorithm effectively utilizes any available parallelism. An important issue to consider is load-imbalance. We present static and dynamic load balancing algorithms in Chapter VIII.

9. *Pipeline:* Finally, the NURBS rendering algorithm must determine the overall order in which sub-processes are performed. This is a function of the structure of the underlying graphics system and the choices made in the items above. The general pipeline of our technique is demonstrated in Figure 1.6. The mappings of these stages to processors on the two target machines, SGI's and Pixel-Planes

[2]A NURBS curve does not have a unique parametrization

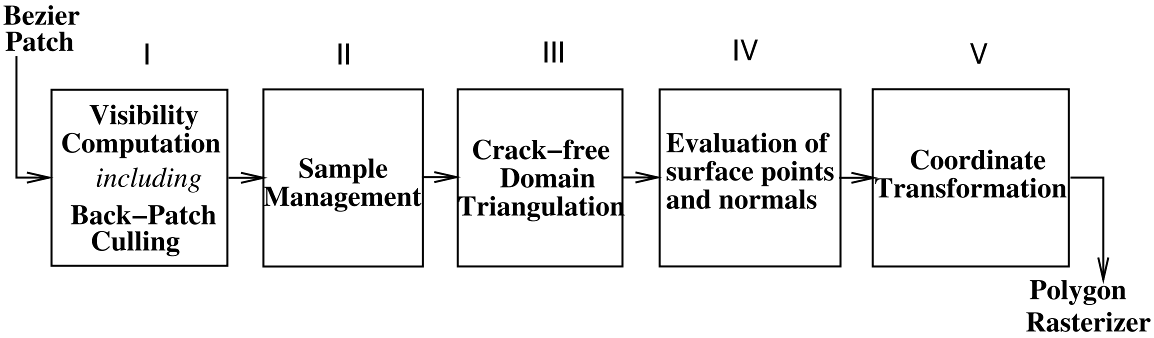

Figure 1.6: System pipeline

5 are discussed in Section 1.6.

As mentioned above, a number of NURBS tessellation algorithms have been proposed recently. Some are based on adaptive sampling, while others use uniform sampling. In particular, Rockwood et al. [RHD89] proposed a real-time algorithm for trimmed NURBS surfaces. A variant of this algorithm has been implemented as SGI GL and OpenGL primitives. Their method uses the Bézier representation for surfaces, and performs tessellations as a function of the view point. However, this implementation often oversamples the surface, and in a few cases, undersamples it. Furthermore, the operations used on trimming curves are relatively expensive and affect the performance of the overall algorithm. At the same time, it is not guaranteed to be crack-free. Overall, our method can render, in real-time, models more than an order of magnitude larger than those by [RHD89]. Some techniques to improve the quality of tessellation and the efficiency of its computation are presented in [FMM86, AES91, AES93]. Luken et al. [Luk93, LC93] use the standard basis to represent NURBS but generate general polygons. Cracks can appear in their images as well. We defer the discussion of these algorithms until after we have presented our tessellation algorithm. In chapter VII we analyze our algorithm and offer a more detailed comparison with other techniques.

1.5 Main contributions

This dissertation presents a number of different techniques to speed up the rendering of large-scale trimmed NURBS models on current graphics systems. At an abstract level, the NURBS representation is transformed to the Bézier form and these Bézier patches are uniformly tessellated and triangulated at rendering time, based on the viewing position. This causes an almost continuous update of detail of the triangular mesh, as opposed to generation of discrete levels of detail. The algorithm computes *tight bounds* on surface properties in order to calculate the sampling density for the tessellation. It also ensures *crack-free* tessellation. In addition to view-frustum culling, the algorithm performs *back-patch culling*, an extension of back-face culling to curved-surface solid models, and makes use of *coherence* between successive frames to reduce the number of polygonal instances computed. It employs a load-balancing algorithm to distribute load when multiple processors are available. The main contributions of this dissertation are briefly described next.

1.5.1 Back-patch culling

A user in a virtual environment with solid models cannot typically see about half of each solid, the part that lies on the back-side of the solid with respect to the view point. Polygon renderers perform back-face culling to reduce processing on polygons that face away from the view point. This dissertation presents a back-patch culling algorithm that culls a full Bézier patch, if all points on the surface face away from the user. To contrast with back-face computation, instead of testing the direction of a single polygon normal with respect to the view point, we need to test the normal directions for all points on the surface. Since there are an infinite number of such points, the back-patch culling algorithm uses a bound on the normals. The implication of using a bound is that the test is slightly conservative: some of the back-facing patches may not be culled. Also note that a Bézier patch must be completely back-facing to be culled as a back-patch. Patches containing the silhouette, for instance, are retained. In practice we are able to cull $30-40\%$ of the patches. The basic steps of the algorithm are summarized below:

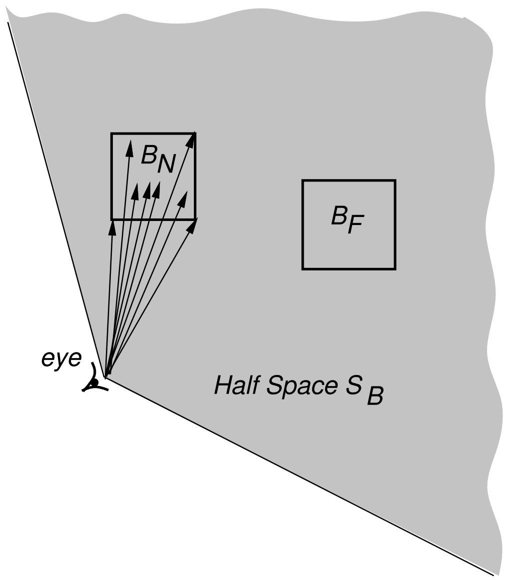

Figure 1.7: Back-patch computation

- For each trimmed Bézier patch, **F**, compute a pseudo-Gauss map (see Chapter II). The relevant properties of this map are that it is also in Bézier form, and the normal at each point on the patch **F** is represented by a point on the map.
- Compute B_F, a bounding volume for **F** (Figure 1.7).
- Based on the bound B_F, compute back-space S_B, a half space in which all normals of the patch **F** must lie for **F** to be a back-patch. (See Chapter III.)
- Compute B_N, a bounding volume for the pseudo-Gauss map.
- If B_N is contained in S_B, **F** is a back-patch. Note that this is the only step that needs to be computed at run-time. All other bounds are pre-computed and stored.

Different types of bounding volumes can be used for B_F and B_N. These can be spheres, bounding boxes, convex hulls etc. If we use spheres to bound both B_F and B_N, we can reduce the test to a single vector difference and a single dot product.

Other bounds are relatively more expensive, but are tighter on average, and find more back-patches. Empirical results (see Chapter III) show that the sphere results in the best overall speed-up. The results are summarized in Table 3.1, for different bounding volumes B_N. The back-patch algorithm affords us an overall rendering speed-up of $20-30\%$. While the fraction of patches culled is small for the spherical bound, it requires fewer tests on average. A vector difference followed by a dot product is counted as one test in this table, as all cases of bounding volumes reduce to a number of such tests.

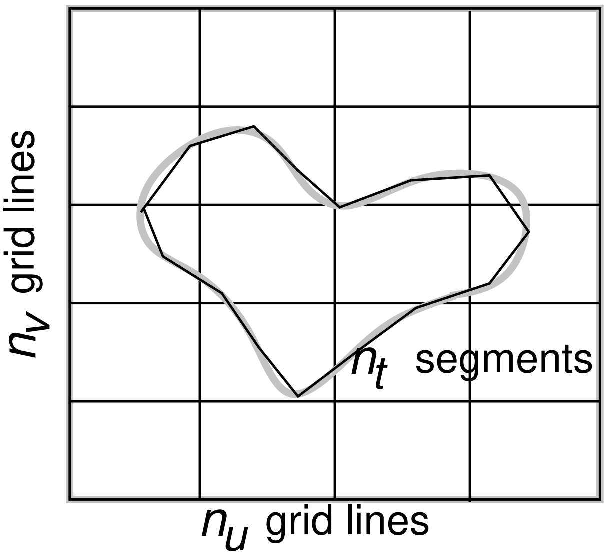

Figure 1.8: Uniformly sampling a patch and its trimming curve

1.5.2 Dynamic sampling

Our algorithm performs sampling tests at two levels. We group proximate patches into clusters. If object hierarchies are available from the modeler, we use those hierarchies. We maintain bounding spheres about each cluster. If the screen size of a sphere is small, we use a pre-determined sparse sampling for Bézier patches comprising the cluster. For each patch of a cluster that covers a large area on screen, we compute the sampling size n_u and n_v, and for each trimming curve on such a patch, we compute the sampling size n_t (see Figure 1.8). $n_u \times n_v$ is the uniform grid size, and n_t is the number of uniformly spaced sample points on the trimming curve. These numbers are based on the following criteria:

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

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

- *Tangent criterion*: The change of tangent between consecutive samples should be bounded.

- *Normal criterion*: The difference between surface normal at consecutive samples should be bounded.

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

We have found that these quality constraints can be approximately satisfied by using estimates of bounds on derivatives and curvatures of the patches. In practice, our primary criterion is the (exact) size criterion (see Chapter IV). If all triangles on screen are restricted to be small, we generate quite smooth images and find that other criteria are approximately satisfied. By factoring out parts of the computation that are independent of the view point and may be performed off-line, we are able to speed up the on-line computation of the sampling size.

1.5.3 Efficient processing of trimmed patches

Traditionally, real-time tessellation of trimmed surfaces has been considered hard [Nas93, Seg95]. We have devised an efficient scheme to tessellate trimmed surfaces. Our algorithm can generate valid triangulations for a patch with an $n \times n$ grid and m trimming points in $O((m + n^2)\log^* m)$ time. [*def:* $\log^* m$ is the number of logs in $\log \circ \log \circ \cdots \log(m) = 1$.] Conceptually, the algorithm is quite simple:

1. March along (the piecewise approximation of) the trimming curve, marking all intersections with the $n_u \times n_v$ grid.

2. For grid cells that are retained by the trimming curve, but not intersected by it, generate two triangles by adding a diagonal to the rectangular grid cell.

3. The remaining cells form a staircase like polygonal chain[3] as shown in Fig. 5.1. The trimming curve forms another polygonal chain, offset from the staircase. These two chains define a planar straight line graph (see Chapter II).

4. Triangulate the graph using one of the known efficient algorithms designed for this purpose, e.g. [Sei91, CCT92].

In this way we do not generate any extra triangles. The actual graph triangulation is the most expensive operation and takes $O(n \log^* n)$ time for n points on average. All operations, apart from triangulation, take time proportional to the sample size and are simple enough to result in a small running time in practice.

1.5.4 Incremental computation

In any interactive application, the user navigates through the environment relatively slowly. In other words, the scene displayed does not change much between successive frames. This implies that the polygonal instance used in one frame is almost valid in the next frame. Our algorithm exploits this coherence. It stores the polygonal instance used in every frame, quickly makes the small change required to satisfy the quality constraints for the next frame, and re-displays it in the next frame. This method, itemized below, eliminates a number of operations required for instance generation:

1. If the screen size of the bounding volume of a cluster changes by a small amount, re-display the previous instance for all patches of the cluster.

2. Otherwise, recompute the sampling size for the patch.

[3] This chain can be degenerate, e.g. it is null if a trimming curve is contained in a cell.

3. Insert or delete appropriate sample points (see Chapter VI).

4. Update the triangulation. Due to the small change, the triangulation can be updated in $O(n \log^* n)$ time, on average. Compare it to $O((m + n^2) \log^* m)$ in the previous sub-section.

This technique results in non-uniform sampling. But we only insert or delete a whole grid line at a time. This ensures that the efficiency gained by computing surface properties on a whole grid-line is still maintained.

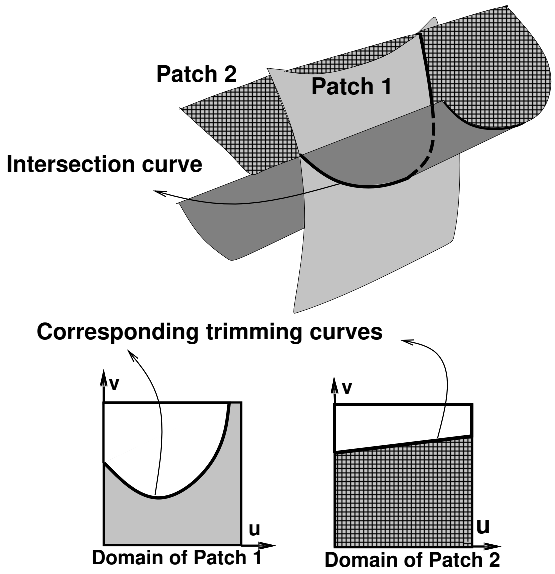

Figure 1.9: An example of different parametric representations of a trimming boundary

1.5.5 Crack prevention

Presence of cracks in images has been a persistent problem in the display of polygonal approximation of surfaces. The problem is that tessellation should be based on geometric properties, which require complex computations. Most renderers, including the one presented here, rely on parametric properties for the computation of sampling size. This means that the parametric representation of two curves representing the same boundary must also be the same. This, in turn, implies that either such consistency must be guaranteed by the modeler, or for each patch, we must find the patch adjacent to it along each boundary. We have chosen the second option, as consistency can be enforced when needed by using the adjacency information. In general, modelers cannot guarantee a consistent representation of trimming curves. A trimming curve is defined in the parametric domain of its patch, and when two trimming curves on two separate patches abut, their respective parametric representations may be arbitrarily different (see Figure 1.9). However, the space representation of these curves must geometrically be the same. We exploit this fact and maintain dual representations of the trimming curves. See Chapter II for details. Using this representation, we guarantee that a boundary curve results in the same samples for patches on both sides of that boundary. Since our triangulation algorithm does not introduce any additional points on a boundary curve, polygonal instances are free of cracks.

1.5.6 Load balancing

Due to the dynamic nature of polygonal instancing, the operations needed to render a patch can vary appreciably over a long sequence of user navigation. This dissertation shows that a static load balancing does not perform very well, and goes on to present a distributed, lock-free load balancing algorithm for shared memory multi-processors. In brief, the algorithm proceeds as follows:

1. Each processor, p_i, maintains a load queue, Q_{p_i}. Q_{p_i} holds identifiers for all patches that are currently allocated to p_i.
2. Once all patches in Q_{p_i} are tessellated and Q_{p_i} becomes empty, p_i searches for a non-empty queue Q_{p_j}.
3. Processor p_i deletes the identifiers in the bottom half of Q_{p_j}, adds them to Q_{p_i}, and starts processing the corresponding patches.
4. The algorithm terminates when all queues are empty.

Two important features of this algorithm are that it uses no blocking locks, and the load balancing part of the algorithm is executed only by processors with empty

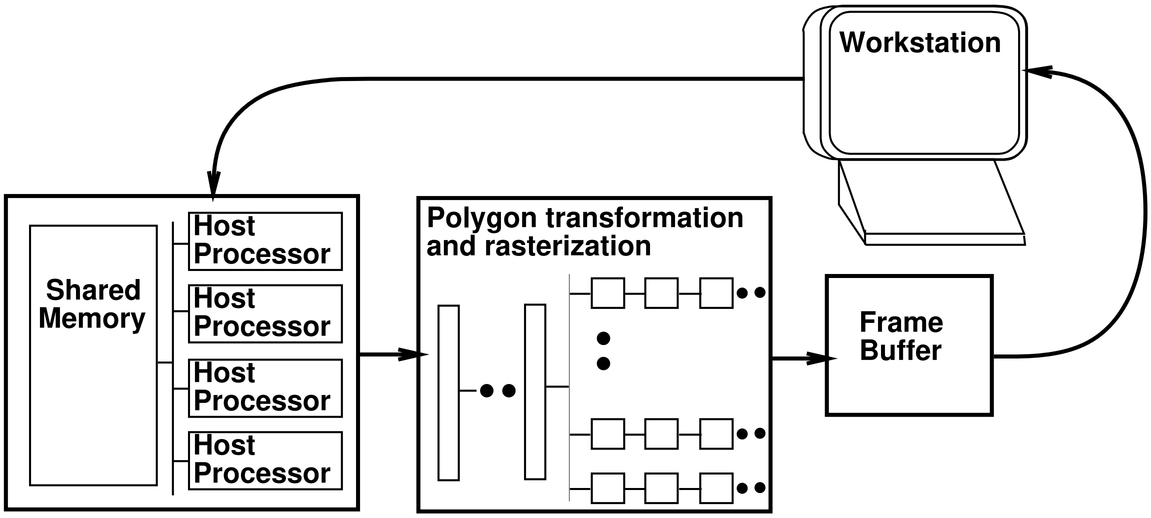

Figure 1.10: Silicon Graphics system architecture

queues. This means that a processor that is busy tessellating patches never gets interrupted for any other operation. By resorting to dynamic load balancing we are able to achieve $10-15\%$ speed-up over static load balancing, and by eliminating locks we achieve an additional 15% speed-up (see Chapter VIII and figure 8.4).

1.6 Overall implementation and performance

We have implemented our trimmed Bézier surface renderer on a Silicon Graphics (SGI) R3000 with a VGX graphics accelerator, an SGI Onyx (with up to 4 R4400 CPUs and a Reality Engine II), on an SGI Power Challenge (with up to 16 CPUs) and on the Pixel-Planes 5 graphics system (with up to 50 GPs and 17 Renderers). Our renderer is built around a core Bézier system called SPEED[4].

Our current implementation on the SGI-Onyx (with 4 R4400 CPUs, 2 Gigabytes of memory, and a Reality Engine II) can display about *five thousand* Bézier patches and that on the Pixel-Planes 5 [Fea89] can display more than *thirty thousand* Bézier patches at interactive frame rates (about 10 frames a second).

[4]Surfaces of Parameter Elegantly and Efficiently Displayed

1.6.1 SGI implementation

Figure 1.10 shows the general architecture of the SGI systems we used. In our implementation, we compute the visibility, sampling and surface triangulation, stages I-IV in Figure 1.6. The stage V and rasterization are performed on the graphics accelerator. The final images are Gouraud shaded. The performance of the algorithm on the SGI Onyx with one R4400 CPU is shown in Table 1.1. We have compared our implementation to that in the standard GL library on SGIs. The SGI-GL implementation is based on the algorithm presented in [RHD89, Nas93] and has a microcoded geometry engine implementation for surface evaluations. That implementation does not take advantage of the multiple host CPUs since the tessellation is done on the Reality Engine hardware.

Model	Num. Patches	SGI-GL primitive	Our overall algorithm	Speed-up due to visibility	Speed-up due to Coherence
Goblet	72	1(4.3 FPS)	36.25	2.67	7.11
Pencil	570	1(1.9 FPS)	45.28	2.89	8.47
Dragon	5354	1(0.1 FPS)	36.47	2.19	7.82

Table 1.1: Speedup due to our techniques (on SGI-Onyx)
See Color plate B for example images of the models.

Although it is difficult to compare two different algorithms and implementations (for example, the design constraints are quite different), we performed the experiments using identical sets of viewing parameters. We recorded these viewing sequences from a non-trained user's attempt to inspect the model closely. Also, a count of the number of patches is not the best measure of the complexity of the model or its rendering. However, given that our models were designed to solve specific problems (such as mechanical design) and not to demonstrate rendering speed, a count of patches gives, in general, a fair idea of the performance.

Table 1.1 shows the relative speed-ups of our algorithms on the SGI Onyx using only one of the CPUs. The third column shows the performance of the standard GL implementation as a baseline, while the fourth shows the speed-up achieved by our overall algorithm. The fifth column corresponds to the speed-up due to visibility computation and the sixth shows the speed-up due to incremental computation.

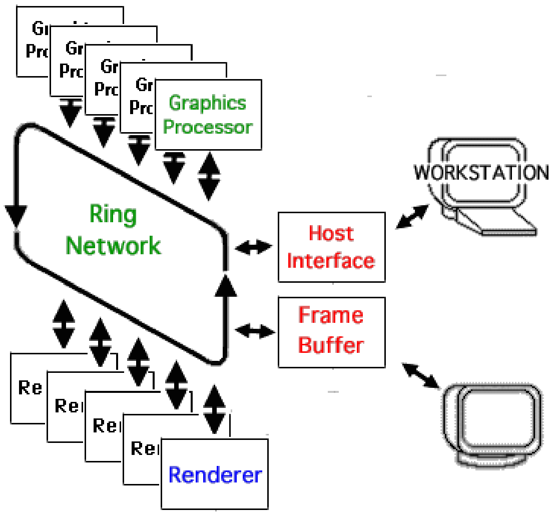

Figure 1.11: Pixel-Planes 5 system architecture

1.6.2 Pixel-Planes 5 implementation

Pixel-Planes 5 [Fea89] uses extensive parallelism to increase rendering performance. This has become the practice in high-performance graphics accelerators [Ake93]. Figure 1.11 presents a block diagram of the Pixel-Planes 5 system. Front-end geometry processing, such as transformation and setup for rasterization, is performed on the Graphics Processors (GPs) which contain Intel i860 RISC microprocessors running at 40 MHz, 8 MB of main memory, and communications hardware. Triangle rasterization and shading is performed on renderer boards which contain arrays of 128 by 128 1-bit processors with local memory [Fea89] and an instruction sequencer. The processing units are connected by a 160 million word per second ring communications network.

All stages, I-V, of our tessellation algorithm run in parallel on the GPs. The

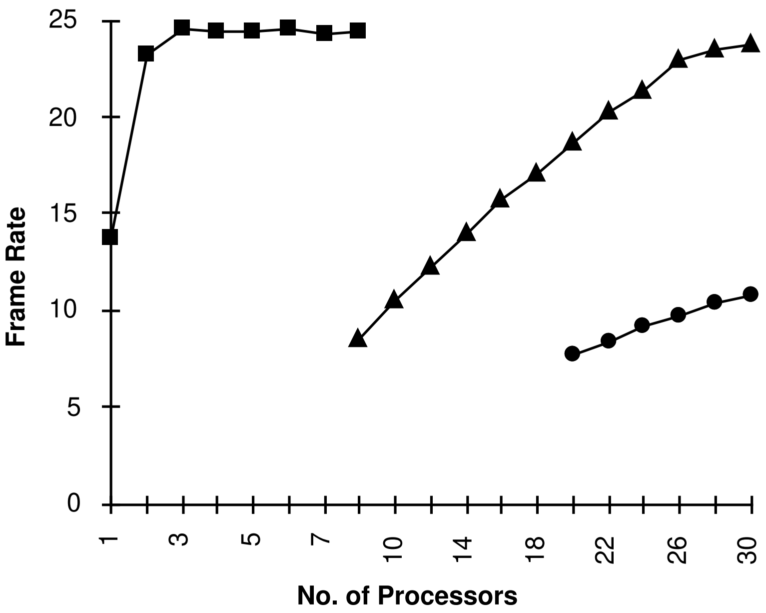

Figure 1.12: Performance *vs* number of processors

Frame rate for three different curved-surface models running on Pixel-Planes 5 as the number of processors varies. Markers: Squares – Utah teapot (32 patches), Triangles – Car panel (1860 patches), Circles – Forsey's Dragon (5354 patches).

algorithm does not require any inter-processor communication during execution. This property not only improves the parallel speed-up, but also will make it easier to port the code to another multi-processor machine.

It is not easy to evaluate the tessellation performance of a particular implementation of this algorithm separately from the triangle rendering performance of the machine on which it is executing. In Figure 1.12, we show the total system performance - tessellation and rendering as a function of the number of GPs, for three models, a simple Utah teapot modeled with 32 Bezier patches, a car body panel consisting of 1700 patches, and a dragon modeled with 5354 patches.

The experiments were run on a medium-sized Pixel-Planes 5 configuration with a maximum of 31 GPs, and 11 renderers. We varied the number of GPs to obtain an idea of the speed-up obtained from parallelism. The size of the dragon model constrains us to a configuration with a minimum of 20 GPs (due to memory requirements). The graphs show the frame rate for a high resolution frame buffer (1280 $\times$ 1024 pixels).

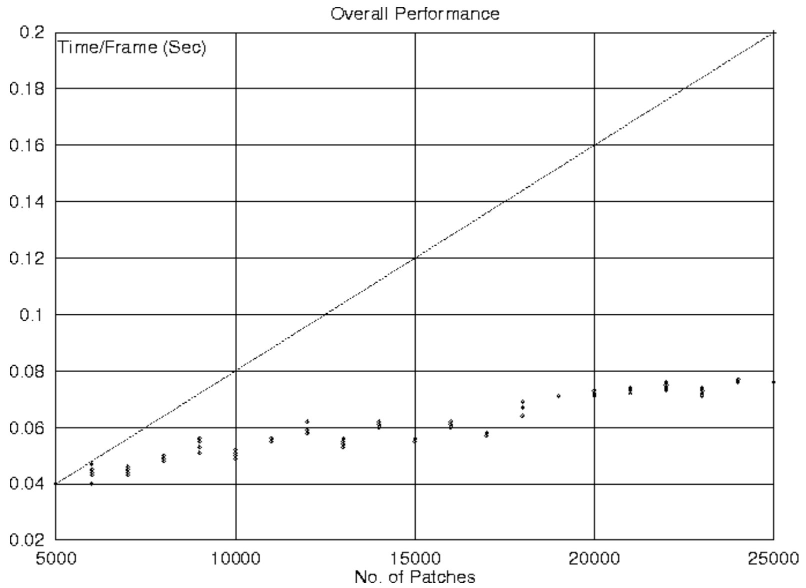

Figure 1.13: Performance of the algorithm *vs* size of the model

The graph in Figure 1.13 shows the scalability of algorithm. The size of the model increases along the `X` axis, while the average time taken to render a frame increases along the `Y` axis. The solid line represents linear slow-down. The dots represent actual data points of our system and show that the slow-down is well below linear. Extrapolating, we expect to gain even better performance improvements as a function of the model-size, as the graphics systems become faster.

1.7 A guide to chapters

In the rest of this dissertation, a basic familiarity with NURBS and Bézier surfaces is assumed. Some mathematical background is required to understand our algorithms better. Chapter II lays these mathematical foundations and introduces our terminology. Chapter III describes visibility processing and explains *back-patch culling*. It also discusses a hierarchical combination of different visibility algorithms into one unified framework. The algorithm for dynamic tessellation of Bézier surfaces, based on tight bounds on surface properties, is presented next. Chapter IV presents methods to compute the sampling required for a smooth display. Chapter V discusses the 2D triangulation of the sample on the domain of the surface. Once these connectivities are established, we impose the same connectivities on the 3D points on the surface and generate the final triangles for display. The use of coherence and its effect on speed-up is demonstrated in Chapter VI. In Chapter VII we compare our algorithm for tessellation of trimmed NURBS to previous techniques. Chapter VIII discusses the load-balancing issues in a parallel implementation of our algorithm. It presents a novel distributed algorithm to balance processor load on a shared memory architecture with little overhead. In most of our presentation we assume a rectangular parametric domains for NURBS which corresponds to tensor-product NURBS. Although this form is quite general and is able to model most surfaces used in computer aided design, it can result in degenerate and unwieldy parametrizations. Chapter IX shows how our algorithms can be extended to non-rectangular domains. Finally Chapter X concludes the dissertation.

Chapter II

Mathematical Background

2.1 Surfaces

This chapter introduces our notation and the mathematical theory underlying our algorithms. [FVFH93] is an excellent introduction to graphics in general. See [de 78, Far93] for introduction to splines.

We refer to the 3D space in which the NURBS (Non-Uniform Rational B-Spline) model is defined as *object space* [Wat95]. We use *homogeneous* coordinates. Viewing transformations, like rotation, translation, and scaling, map objects to *eye space*, and a perspective transformation maps them to the *screen space*.

A curve or a surface may be represented algebraically. For example $y = f(x)$ defines a curve. In its implicit form, $f(x, y, z) = 0$ may define a curve (or a surface). The curve (or surface) is given by the set of points $\{x, y, z\}$ such that $f(x, y, z) = 0$. In general, It is difficult to design geometry using such representations. Moreover, computing a set of points on a given curve or surface is compute-intensive. That is why parametric polynomials are more popular for geometric modeling. These are easier to design and evaluate. A parametric polynomial, $p(u)$ is a mapping from a parametric domain u to a range of values in $\mathcal{R}^3$.

A homogeneous point on a three dimensional NURBS curve has four coordinates: {`wx,wy,wz,w`}. The curve itself is defined by four parametric polynomials, one for each coordinate. All polynomial are functions of the same one dimensional parametric

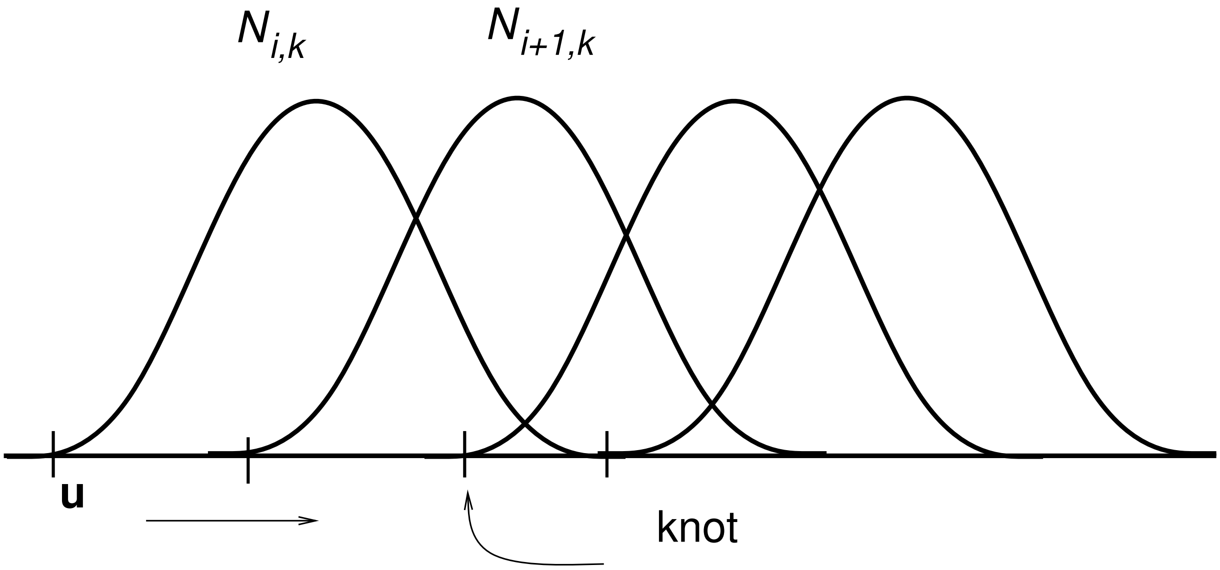

Figure 2.1: The B-Spline basis functions

domain u. Thus, for any given parameter value, the four polynomials evaluate to the four coordinates of a point on the curve. A NURBS curve is composed of a number of segments or *spans*. In the parametric domain, these spans are defined by a *knot vector*. A knot vector is sequence of knots, or parameter values, $\{u_i\}$, along the domain. A NURBS polynomial is specified by a scaled sum of a set of basis functions. Notationally, when these scales are 4-tuples, the set of four implied polynomials form a curve. Each 4-tuple represents a homogeneous point in object space. These points are called *control points*. The fourth coordinate of these points are referred to as their *weights*. We assume non-negative weights in the rest of this dissertation. This assumption ensures that the surface lies inside the convex hull of its control points. While we can relax this restriction at an added computational cost, it is not worth incurring as most surface representations used in CAD do, in fact, have positive weights.

Formally, a point on the curve $\mathbf{s}(u)$ of degree $k-1$ is given by

$$\mathbf{s}(u) = \frac{\sum_{0}^{n_c} w_i \mathbf{p_i} \mathcal{N}_{i,k}(u)}{\sum_{0}^{n_c} w_i \mathcal{N}_{i,k}(u)},$$

where $\mathbf{p}_i$ are the control points, w_i their weights, and $\mathcal{N}_{i,k}$ the blending functions composed of the standard basis functions (see Figure 2.1).

$\mathcal{N}_{i,k}(u)$ is defined recursively over the knot interval $[u_i, u_{i+k}]$ as follows:

$$\mathcal{N}_{i,1}(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{N}_{i,k}(u) = \frac{(u_{i+k} - u)\mathcal{N}_{i+1,k-1}(u)}{u_{i+k} - u_{i+1}} + \frac{(u - u_i)\mathcal{N}_{i,k-1}(u)}{u_{i+k-1} - u_i}$$

The n_c control points are specified in object space and determine the shape of the curve. Without loss of generality, we assume a normalized domain in this presentation: $u \in [0, 1]$.

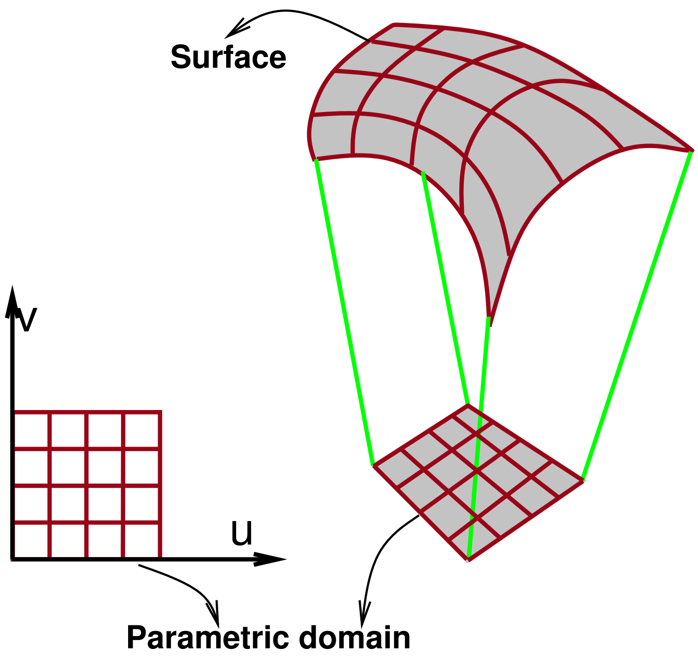

Figure 2.2: A surface patch and its parametric domain

A tensor product NURBS surface is the tensor product of two NURBS curves and is defined over a two dimensional parametric domain, $\mathcal{D}$, with coordinates u and v, $0 \leq u \leq 1, 0 \leq v \leq 1$. The shape of the surface is controlled by two one-dimensional arrays of knot vectors, and a two-dimensional array of control points [Far93]. Figure 2.2 illustrates the relationship between a domain and a surface patch. Formally, the tensor product surface is:

$$\mathbf{S}'(u,v) = \frac{\sum_{i=0}^{n_{su}} \sum_{j=0}^{n_{sv}} w_{ij} \mathbf{p}_{ij} \mathcal{N}_{i,k}(u) \mathcal{N}_{j,k}(v)}{\sum_{i=0}^{n_{su}} \sum_{j=0}^{n_{sv}} w_{ij} \mathcal{N}_{i,k}(u) \mathcal{N}_{j,k}(v)}$$

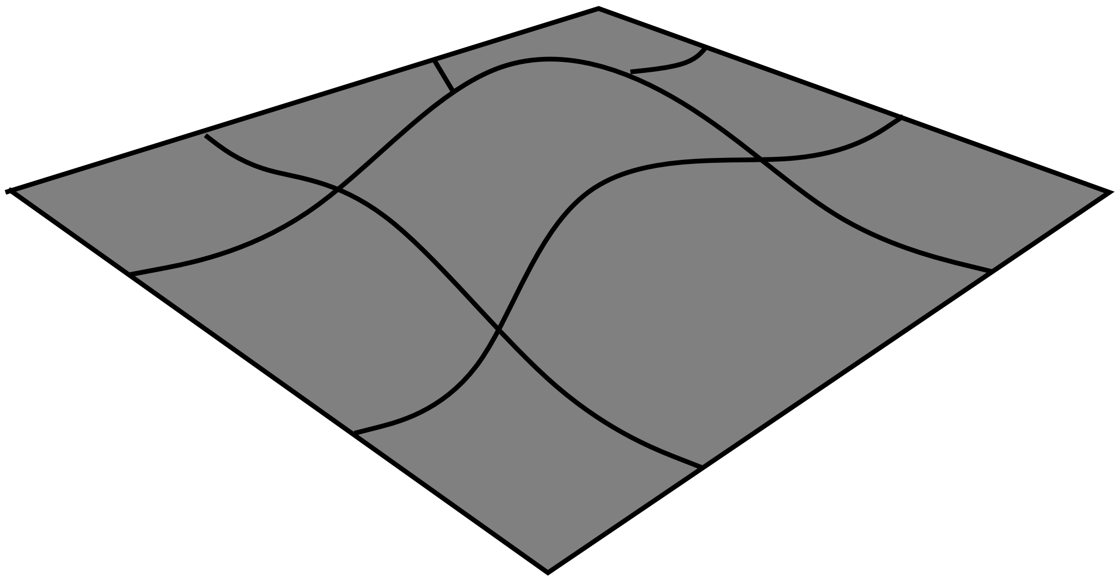

Figure 2.3: Tensor product of basis functions

Figure 2.3 shows an example of tensor product of basis functions. A surface is described by a mesh of control points $\mathbf{p}_{ij}$ (see Figure 2.4) and there exist two knot vectors, one along each of parameters u and v. A trimmed NURBS surface, $\mathbf{S}(u,v)$, is a subset of $\mathbf{S}$'(u,v) defined by a set of trimming curves. A trimming curve is a closed piecewise sequence of NURBS curves defined in the domain, $\mathcal{D} = [0,1] \times [0,1]$, of $\mathbf{S}'(u,v)$. For consistency and unique representation, we define the following trimming rule to obtain $\mathcal{D}_s$, a subset of $\mathcal{D}$:

- A curve (moving in the direction of increasing parameter value) trims out the part of the surface domain immediately to the left of it. More precisely, consider a point q on the curve with tangent direction $\vec{w}$. A domain point q', $|qq'| \to 0$, does not belong to $\mathcal{D}_s$, if the counter clockwise angle $\vec{qq'}$ makes with $\vec{w}$ is less than π (see Figure 2.5).

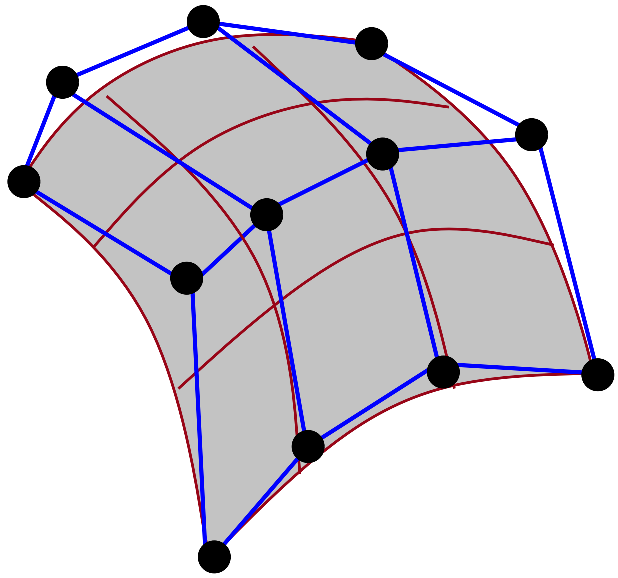

Figure 2.4: A surface patch and its control points

- If the line segment q_1q_2, $q_1, q_2 \in \mathcal{D}$, does not intersect a trimming curve, either both q_1 and q_2 belong to $\mathcal{D}_s$ or both belong to $\mathcal{D} - \mathcal{D}_s$.

Now,

$$\mathbf{S}(u,v) = \mathbf{S}'(u,v) \mid (u,v) \in \mathcal{D}_s$$

In general, the trimming curves are not allowed to intersect. Intersecting trimming curves do not have well defined interiors. [KM94b] presents an algorithm to generate non-intersecting trimming curves from a set of potentially intersecting trimming curves in a semantically consistent way.

Bézier surfaces are special types of NURBS surfaces, that do not have any knots except at the corner points. The multiplicity of a u-knots is the same as the u-order of a Bézier surface and that of a v-knot is same as the v-order of the surface. One of the advantages of the Bézier representation is that it is more intuitive. It is also more easy to evaluate than general NURBS.

Using knot insertion [Far93] we decompose each NURBS surface into a series of rational Bézier patches, and each NURBS curve into a series of rational Bézier curves. Knots may be inserted at any parameter value without altering the shape of the NURBS. The control points, however, are altered when we insert (or remove)

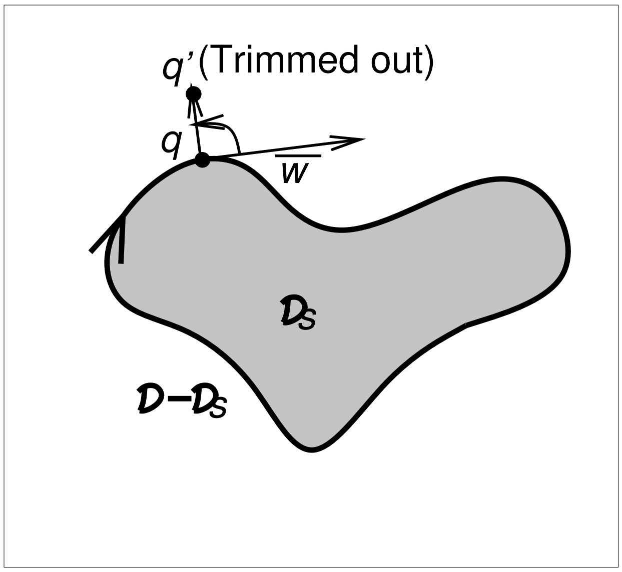

Figure 2.5: Trimming rule

knots. We insert more knots at each knot value in the knot vectors of a NURBS of order n until the multiplicity of each knot equals n. The resulting control points are in Bézier form. The control points are updated as follows:

The insertion of knot u', $u_j < u' \leq u_{j+1}$, results in $n_c + 1$ control points $\mathbf{p}'_i$ from $\mathbf{p}_i$ as follows:

$$\mathbf{p}'_0 = \mathbf{p}_0;\ \mathbf{p}'_{n+1} = \mathbf{p}_{n_c};\ \text{and}\ \mathbf{p}'_i = (1 - a_i)\mathbf{p}_{i-1} + a_i\mathbf{p}_i\ \text{where}$$

$$a_i = \begin{cases} 1 & i = 1, \cdots, j-k \\ \frac{u'-u_i}{u_{i+k}-u_i} & i = j-k+1, \cdots, j \\ 0 & i = j+1, \cdots, n_c \end{cases}$$

We insert the minimum number of knots as a function of the knot sequence of the original surface and its order. Knots spaced closer than a user-specified tolerance[1], in the normalized parametric domain are coerced to the same value before applying the knot insertion algorithm. Since the Bézier representation is the underlying primitive of our system, we introduce it here.

A rational 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, $w_{ij}, 0 \leq i \leq m, 0 \leq$

[1]we use 2×10^{-5}

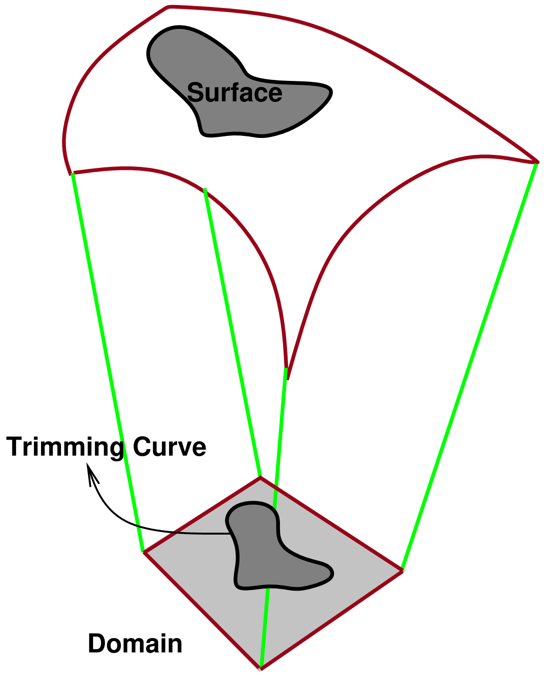

Figure 2.6: A trimmed surface patch

$j \leq n$:

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

where the Bernstein function $\mathcal{B}$ 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 Bézier 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, $w_k, 0 \leq k \leq n$:

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

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

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

The partial derivatives of a Bézier patch is computed as follows:

Let us denote the numerator of $\mathbf{F}$(u,v) by $\mathbf{P}(u,v)$ and the denominator by $W(u,v)$.

$$\begin{aligned}
\mathbf{P}_u(u,v) &= \sum_{i=0}^{m-1}\sum_{j=0}^{n}(w_{i+1,j}\mathbf{p}_{i+1,j} - w_{i,j}\mathbf{p}_{i,j})\mathcal{B}_i^{m-1}(u)\mathcal{B}_j^n(v) \\
W_u(u,v) &= \sum_{i=0}^{m-1}\sum_{j=0}^{n}(w_{i+1,j} - w_{i,j})\mathcal{B}_i^{m-1}(u)\mathcal{B}_j^n(v) \\
\mathbf{P}_v(u,v) &= \sum_{i=0}^{m}\sum_{j=0}^{n-1}(w_{i,j+1}\mathbf{p}_{i,j+1} - w_{i,j}\mathbf{p}_{i,j})\mathcal{B}_i^{m}(u)\mathcal{B}_j^{n-1}(v) \\
W_v(u,v) &= \sum_{i=0}^{m}\sum_{j=0}^{n-1}(w_{i,j+1} - w_{i,j})\mathcal{B}_i^{m}(u)\mathcal{B}_j^{n-1}(v)
\end{aligned}$$

Notice that $\mathbf{P}_u, \mathbf{P}_v, W_u$ and W_v are each in Bézier form themselves. This means that these functions also lie in the convex hull of their respective control points. We will use this property later. To compute the derivative of the rational patch we use the quotient rule:

$$\left(\frac{\mathbf{P}}{W}\right)_u = \frac{\mathbf{P}_u W - \mathbf{P} W_u}{W^2}, \quad \left(\frac{\mathbf{P}}{W}\right)_v = \frac{\mathbf{P}_v W - \mathbf{P} W_v}{W^2}.$$

We drop u, v and t from notation, if it is implicit in the context.

2.2 Data representation

Our system allows trimming curves to be piecewise Bézier or piecewise linear. In the rest of this dissertation, we only allude to the Bézier form. Linear curves can be thought of as a special case of Bézier curves. However, in the actual implementation the piecewise linear curves are not further tessellated.

A degree $m \times n$ rational Bézier surface is represented in our database by a mesh of $(m+1)\times(n+1)$ 4-tuples. A 4-tuple, $\{wx, wy, wz, w\}$, corresponds to the control point $\{x, y, z\}$ and its rational weight w. Similarly, a Bézier trimming curve of degree n' is stored as a sequence of n' $\{wu, wv, w\}$-tuples. Note that the trimming curves and their control points are specified on the domain of a patch $\mathbf{F}$. In addition to this domain representation, $\mathbf{f}_d$, of a trimming curve, we also maintain its space representation, $\mathbf{f}_s$, where $\mathbf{f}_s$ is an approximation for $\mathbf{F} \circ \mathbf{f}_d$. [$\circ$ denotes function composition] We use an approximation because the degree of $\mathbf{F} \circ \mathbf{f}_d$ is too high and it is not always parametrizable. The space curve $\mathbf{f}_s$ is represented by its control points, a sequence of $\{wx, wy, wz, w\}$-tuples. By storing a consistent space representation of the trimming curves, we are able to tessellate two adjacent patches independently of each other without introducing any cracks in the image. (See Chapter IV for details.)

2.3 Surface evaluation

In order to render a surface we must evaluate points on it and normals at those points. As shown earlier, the normal are cross products of partial derivatives, which themselves are Bézier surfaces. In this section we show how to efficiently compute points on a Bézier surface (and curve).

The de Casteljau algorithm [Far93] is a popular technique to compute points on a surface. Hardware systems [DBB+89, BEWD91] have been designed around this algorithm. This method is based on the following observation:

$$\mathcal{B}_i^n(u) = (1-u)\mathcal{B}_i^{n-1}(u) + u\mathcal{B}_{i-1}^{n-1}(u)$$

This implies the recursion:

$$\mathbf{p}_i^r(u) = (1-u)\mathbf{p}_i^{r-1}(u) + u\mathbf{p}_{i-1}^{r-1}(u), 0 \le i \le n-r, 1 \le r \le n$$

where p_i^0 are control points of a curve of degree n and p_0^n is a point on the curve. Notice that the derivative of the curve is $n[\mathbf{p}_1^{n-1} - \mathbf{p}_0^{n-1}]$ and does not need to be evaluated separately. To evaluate a point on a surface of degree $m \times n$, we use the following recursion:

$$p_{i,j}^{r,r} = [1-u \quad u] \begin{bmatrix} \mathbf{p}_{i,j}^{r-1,r-1} & \mathbf{p}_{i,j+1}^{r-1,r-1} \\ \mathbf{p}_{i+1,j}^{r-1,r-1} & \mathbf{p}_{i+1,j+1}^{r-1,r-1} \end{bmatrix} \begin{bmatrix} 1-v \\ v \end{bmatrix}, \quad 0 < r \le k, \quad i,j \in 0 \cdots k-r$$

where $k = \min(m, n)$. If $m \neq n$, $\mathbf{p}_{i,j}^{k,k}$ form a curve, which can be evaluated using the algorithm for curves described above. The time taken by the de Casteljau algorithm to evaluate a point on a Bézier curve of degree n is proportional to n^2. On the other hand the modified Horner's rule [Far93] results in the evaluation of the point in $O(n)$ time, and each of $\mathbf{F}_u$ and $\mathbf{F}_u$ in an additional $O(n)$ each. This results in a faster evaluation technique for the method based on the modified Horner's rule:

$$\mathbf{p}^n(u) = \left[\left\{\begin{pmatrix} n \\ 0 \end{pmatrix} u'\mathbf{p}_0 + \begin{pmatrix} n \\ 1 \end{pmatrix} u\mathbf{p}_1\right\} u' + \begin{pmatrix} n \\ 2 \end{pmatrix} u^2\mathbf{p}_2\right] u' \cdots + \begin{pmatrix} n \\ n \end{pmatrix} u^n\mathbf{p}_n$$

To evaluate a point $\mathbf{F}(u, v)$, we employ the rule once for u and once for v. This generalizing of the technique to two dimensions yields a computational complexity of $O(mn)$:

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

where

$$\mathbf{P}_i(v) = \sum_{j=0}^{n} w_{ij}\mathbf{p}_{ij}\mathcal{B}_j^n(v)$$

However, notice that $[\mathbf{P}_i]$ is independent of u. We can efficiently evaluate multiple points on the curve $\mathbf{F}(u, v_0)$ for a fixed v_0 by storing parts of computation. We store the value of $[\mathbf{P}_i(v_0)]$ and re-use it for each evaluation of $\mathbf{F}(u, v_0)$. Thus each evaluation of a point along an isoparametric line on a surface of degree $m \times n$ requires only $O(m + n)$ time, as opposed to $O(mn)$ for the Horner's rule or $O(m^2n^2)$ for the de Casteljau algorithm.

2.4 Gauss maps

Gauss maps provide a convenient way to characterize the normals of a surface, which we need in order to determine if the surface is visible. We define the Gauss map of a surface with a continuous unit normal vector field as follows [O'N66]:

> The Gauss map $\mathbf{G}_r$ of a surface, $\mathbf{F}$, is a map $\mathbf{G}_r : \mathbf{F} \to S^2$, the 2-Sphere, i.e. the standard sphere embedded in 3D, which maps point $\mathbf{F}(u, v)$ to the vector $\mathbf{U}(u, v)$, translated to the origin, where $\mathbf{U}(u, v) = \frac{\mathbf{N}(u,v)}{|\mathbf{N}(u,v)|}$.

In other words, if $\mathbf{N}(u,v)$ is the normal vector at a point (u,v) of a surface, $\mathbf{G}_r(u,v) = \mathbf{S}_p$, where $\mathbf{S}_p$ is a point on the unit sphere centered at the origin $\mathbf{O}$ and $\vec{\mathbf{S}_p\mathbf{O}}$ is parallel to $\mathbf{N}(u,v)$. Thus the function $\mathbf{G}_r(u,v)$ can be used to compute the unit normal of the surface at the point (u,v). However, $\mathbf{G}_r$ is relatively expensive to compute. Moreover, for many degenerate parametrizations a continuous unit normal vector field does not exist. Fortunately, since we are interested only in the direction of the normal vectors, we are able to use a pseudo-Gauss map, $\mathbf{G} = \mathbf{F}_u \times \mathbf{F}_v$ which gives the normal direction for each (u,v). Note that $\mathbf{G}(u,v)$ is parallel to $\mathbf{G}_r(u,v)$, but is not necessarily of unit length. The vector between a point, $\mathbf{G}(u,v)$, and the origin lies in the same direction as $\mathbf{G}_r(u,v)$. Thus $\mathbf{G}(u,v)$ is a scalar multiple of $\mathbf{G}_r(u,v)$. Unfortunately even the pseudo-map is not defined for all points on a Bézier surfaces. For instance, if $\mathbf{F}_u(u,v)$ and $\mathbf{F}_v(u,v)$ are parallel, $\mathbf{G}(u,v)$ is not defined. Since we use these maps only to compute visibility, we assume continuity of visibility and ignore degenerate points. We have found this to work well in practice. The actual visibility computation algorithm is presented in Chapter III. A more conservative approach would mark all patches with any degenerate normals as always visible. The main benefit of using the pseudo-Gauss map formulation is that $\mathbf{G}$ is itself a Bézier surface, and hence is specified by a mesh of control points. Thus it is easy to bound the extents of a pseudo-map by computing a bound on the extents of the convex hull of the control points.

If $\mathbf{F}$ has a polynomial representation, the pseudo-normal surface is a $(2m-1) \times (2n-1)$ degree Bézier patch. If $\mathbf{F}$ is rational, the degree of the cross-product is $4m \times 4n$. Let $\mathbf{P}(u,v) = (X(u,v), Y(u,v), Z(u,v))$.

$$\mathbf{F}_u = \frac{\mathbf{P}_u W - \mathbf{P} W_u}{W^2}, \qquad \mathbf{F}_v = \frac{\mathbf{P}_v W - \mathbf{P} W_v}{W^2}.$$

Therefore, the pseudo-normal surface is:

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

2.5 Solving simultaneous equations

In our application we need to find common zeros of polynomials. There is well-researched elimination theory [Sed83, MD92] that derives techniques for computing zeros. In particular, we use the method of resultants [Usp48].

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

There exist a variety of resultant formulations. We use Sylvester's resultant for two polynomial equations [Usp48]. Consider the problem of finding common roots of $f(x) = 0$ and $g(x) = 0$ where

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

Assume, without loss of generality, that $n > m$ and $a_n \neq 0$. Also assume that f and g are not identically 0. If there exists a common root, polynomials f and g have a common divisor. As shown by Sylvester, this divisor is a function of x *iff* polynomials f_1 and g_1 of degrees less than n and m exist such that they are not identically 0 and

$$f g_1 + f_1 g = 0 \tag{II.1}$$

Assume now that

$$\begin{aligned} f_1 &= \lambda_{n-1} x^{n-1} + \lambda_{n-2} x^{n-2} \cdots \lambda_0, \\ g_1 &= \mu_{m-1} x^{m-1} + \mu_{m-2} x^{m-2} \cdots \mu_0, \end{aligned}$$

To solve for λ's and μ's, not all 0, such that Equation II.1 holds, substituting f_1 and g_1 in (II.1) and equating coefficients, we obtain $m + n$ homogeneous linear equations. These equations have a non-trivial solution *iff* their determinant is 0. This determinant, also known as Sylvester's determinant, is:

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

If f and g are each functions of two variables, say x and y, we choose one of those variable, say x, to be primary and consider each entry of the determinant an expression in y. We need to compute the zeros of this expression. This problem reduces to computing the eigenvalues of a companion matrix [MD92]. Good implementations of eigenvalue evaluators are available as part of numerical libraries like EISPACK and LAPACK. The resulting algorithms are fast, accurate, need no initial guess of the solutions, and do not suffer from convergence problems [MD92].

In Chapter IV we will need to find roots of derivatives of Bézier functions. This theory is useful for that. We first compute the Sylvester's determinant each of whose entries is an array of coefficients. The value of the determinant is a function of these coefficients. To solve for its zeros, we construct the companion matrix and employ LAPACK routines to find the eigenvalues.

2.6 Planar straight line graphs

A planar straight line graph (PSLG) is defined by a a set of points and line segments on a plane. We are interested in only closed PSLGs, the segments of which enclose a finite region on the plane, and will hereafter use *PSLG* to mean *closed PSLG*. For our application a PSLG can be thought of as a generalized polygon; segments occur on the boundary as well as the interior (see Figure 2.7). None of the segments intersect, except when they abut and have a common end point. A triangulation of a PSLG (see Figure 2.7) is a decomposition of its enclosed region into triangles with the following

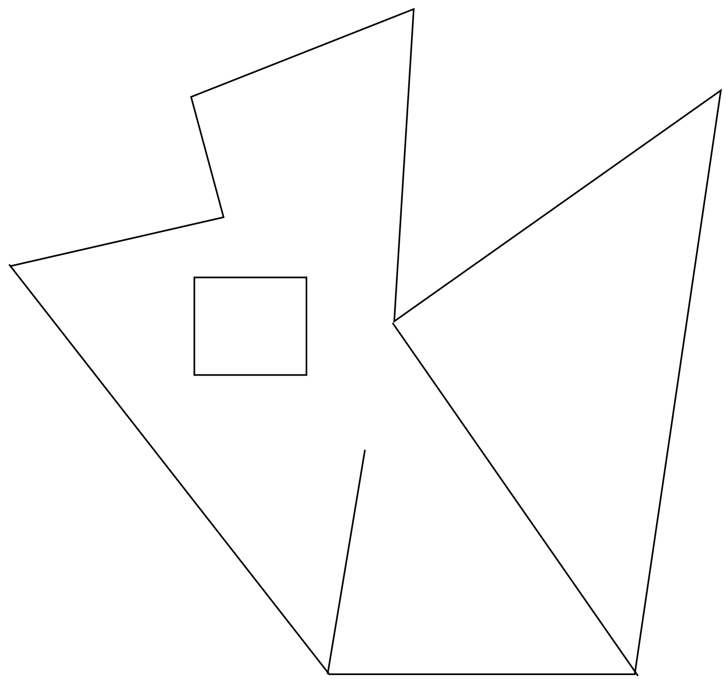

Figure 2.7: A closed planar straight line graph

constraints:

1. Each point of a PSLG is a vertex of a triangle, and each segment of PSLG is an edge of a triangle.

2. No triangle edges intersect.

3. No vertex of any triangle lies on the edge of another triangle (except at its endpoints).

As we will see in Chapter V, efficient triangulation of PSLGs is an important part of the algorithm for surface tessellation. A number of techniques to triangulate polygons and PSLGs have been proposed in computational geometry literature [DP89, BE92, O'R94, Kum95]. A number of these techniques attempt to optimize different properties of the triangulation, e.g. angles of the triangle, size of triangles etc. In particular, constrained Delaunay triangulation [O'R94] has the useful property of being regular and is therefore routinely used in applications like fluid dynamics and finite element analysis. It avoids long and thin triangles. In fact, it maximizes the minimum angle of any triangle. Unfortunately, it is expensive to compute. It takes $O(n \log n)$ time to compute the Delaunay triangulation of a PSLG containing n vertices.

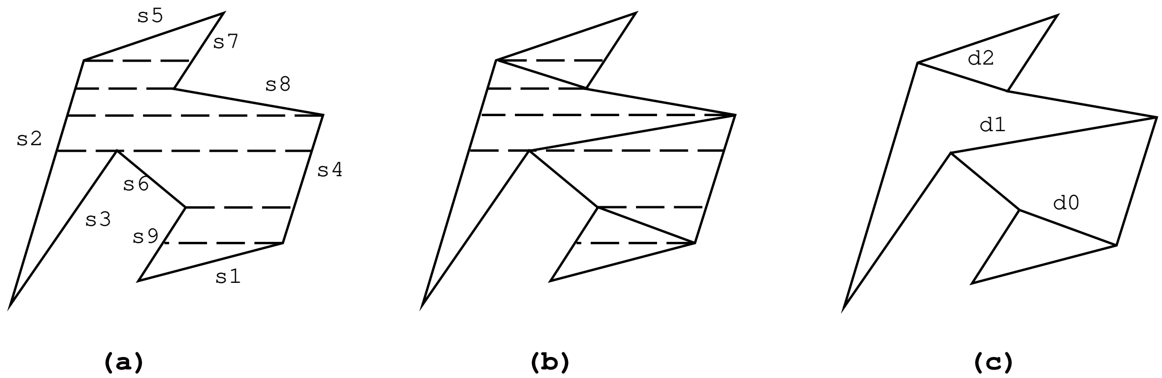

Figure 2.8: Triangulation of a PSLG

In our system, we used the public domain implementation by Narkhede et al. [NM95] of a randomized incremental algorithm devised by Seidel [Sei91]. The [Sei91] algorithm triangulates PSLGs by first decomposing it into a number of monotone polygons, and then triangulating each monotone polygon. It can triangulate a PSLG with n vertices in $O(n \log^* n)$ expected time. In practice, we have found the implementation of Narkhede et al. to be quite efficient. A brief description of the basic idea of [Sei91] follows.

Seidel's algorithm starts by decomposing the PSLG into a set of trapezoids (see Figure 2.8). It first randomly orders the edges of the input set, and then adds these edges to the trapezoidal data structure in that order, updating the trapezoidization each time. It decomposes the trapezoids into monotone polygons in linear time (Figure 2.8(c)). A *monotone polygon* has two monotonically vertical polygonal chains. The monotone polygons are obtained from the trapezoidal decomposition by adding a diagonal between two vertices of the original PSLG if they lie on different sides of a trapezoid (Figure 2.8(b)).

Narkhede's implementation employs the greedy scheme of Fournier et al. [FM84] to triangulate each monotone polygon by repeatedly cutting a convex corner. It starts with the lowermost point, $\mathbf{p}_l$, and connects it to each vertex visible from it by drawing diagonals. A *diagonal* of a PSLG divides it into two components. The diagonal does not intersect any edge of PSLG but intersects every line between the two components. Each diagonal generates a new triangle reducing the size of the untriangulated part of the polygon by one, while maintaining its monotonicity property. Once all points directly visible from $\mathbf{p}_l$ are connected to it, the algorithm moves to the next lowest point and repeats the process until the complete polygon is triangulated. We modified the [FM84] algorithm to avoid thin triangles and vertices of high degree, as described in Chapter V.

Chapter III

Visibility

3.1 Overview

While viewing a large model composed of many thousands of Bézier patches, usually not all patches are visible from a given view point. For instance, a significant part of the model may lie outside the viewing volume. In addition, for a solid model, approximately half of the model on screen faces away from the viewer. These back-facing patches are hidden behind other patches and are not visible to the viewer.

In general, the exact computation of the visible portions of a NURBS model is a non-trivial problem requiring silhouette computation [KM94a]. In this chapter we show that it is relatively simple to perform a conservative visibility test. We present an efficient algorithm to determine if a patch is back-facing in a given frame. We then cull it away after stage I of our system pipeline (Figure 1.6) and avoid its further processing. For a small cost early in the process we are able to ease much load on the triangle rendering pipeline.

Our test is conservative; some back-facing patches may not be culled, but a visible patch is never culled. The standard z-buffering performed by the rasterizer ensures correct visibility. For most solid models we have examined, back-patch culling eliminates about 30-40% of the patches. Note that by restricting our granularity to patches, our method retains both front-facing patches, and those on the silhouette. Still we are able to cull away large parts of models. This is specially true for mod-

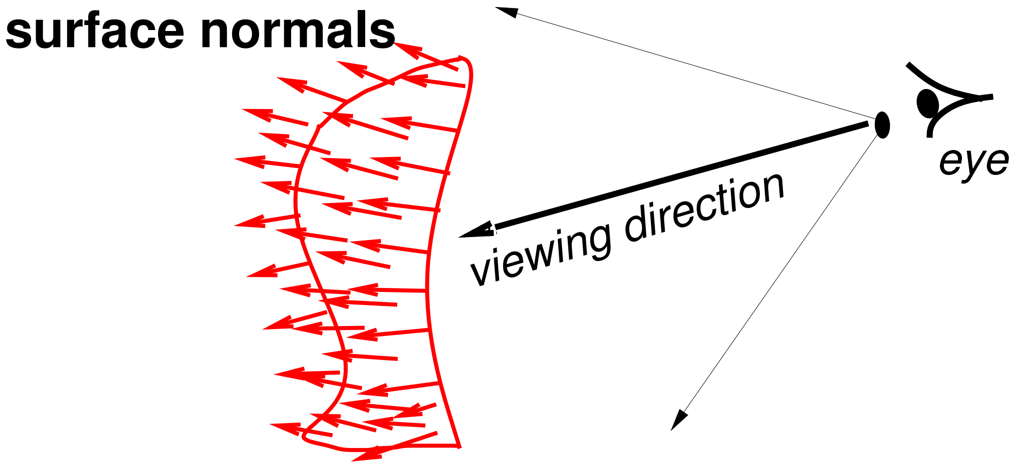

Figure 3.1: A back-facing patch

els with a large number of small patches, which, in our experience, many real world models indeed have. Table 3.1 lists the performance of back-patch culling for some representative models.

Model	Num. Patches	% patches culled	Speedup on PXPL5	Speedup on Onyx
Pencil	570	35%	23%	27%
Dragon	5354	38%	32%	32%
Car	10,012	29%	19%	24%
Sub-room	21,900	42%	29%	NA

Table 3.1: The effectiveness of back-patch culling
See Color plate B for example images of the models.

Overall, back-patch culling improves the observed rendering performance by about $20 - 35\%$.

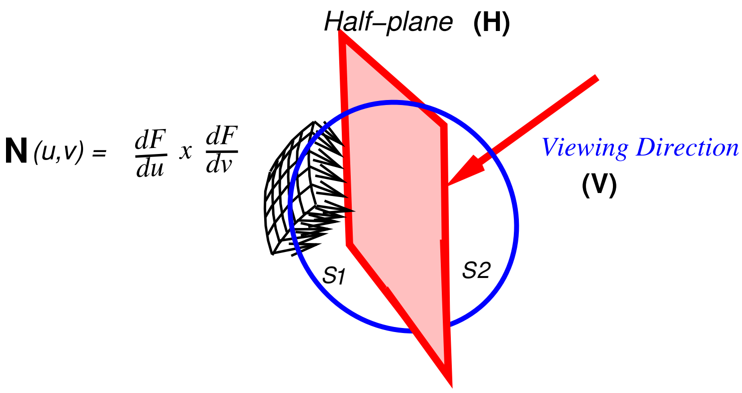

Figure 3.2: The back-patch computation for parallel projection

3.2 Patch clipping

A Bézier patch, **F**, is contained in the convex polytope of its control points [Far93]. Let us denote this convex polytope as P_F. We denote by B_F the smallest axis-aligned bounding box that encloses P_F.

The first phase of visibility processing involves checking whether a patch lies in the viewing volume at all. We perform this computation by testing if B_F intersects the view frustum. Since P_F bounds **F** more tightly than B_F, using P_F for the intersection test yields fewer false acceptances. In practice, though, the test is much cheaper for B_F, making it more efficient overall. On average, the intersection test for P_F requires approximately 6 times more operations than that for B_F, and is able to report only about $10 - 15\%$ more patches as lying outside the view frustum. This result is not surprising, as the test using P_F performs better only in the neighborhood of the boundary of the view-frustum. If B_F intersects the view-frustum, while P_F does not, it is quite likely that **F** lies close to the view-frustum.

3.3 Back-patch culling

In order to improve rendering performance, most renderers of polygonal solid models detect back-facing polygons and eliminate them early in the pipeline. The normal of a back-facing polygon faces away from the viewer's eye. Analogously, if all the surface normals of a Bézier patch point away from the eye-point, we refer to it as a *back-patch* (Figure 3.1). A similar technique was independently proposed by Shirman et al. [SAE93]. They maintain a floating cone that bounds all normals and derive expressions to compute if the cone faces away from the user. Our approach is simpler and more general. It is easily customized to different applications by appropriately choosing the complexity of the back-patch test, as shown later. We first demonstrate the technique for back-patch computation under parallel projection, since that is easier to understand, and later extend it for perspective projection.

3.3.1 Parallel projection

The set of normals to a Bézier surface, $\mathbf{F}$, are contained in the pseudo-Gauss map introduced in Chapter II. Call this map $\mathbf{G}$. Corresponding to the direction of viewing vector $\mathbf{v}$, define a half-plane $\mathbf{H}$ perpendicular to $\mathbf{v}$ that delineates two half-spaces $S1$ and $S2$ (see Figure 3.2). If the pseudo-Gauss map of the surface lies entirely in $S1$, so must all the normals of $\mathbf{F}$, which means $\mathbf{F}$ faces away from the viewer.

Recall (from Chapter II) that a rational Bézier patch, is defined as

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

and the pseudo-map, $\mathbf{G} = \mathbf{F}_u \times \mathbf{F}_v$, is defined as

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

After expanding the expression for $\mathbf{G}$, simplifying, and dividing by W we obtain

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

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

Let us denote $\mathbf{G} = \frac{\mathbf{G}_N}{\mathbf{G}_D}$. In order to evaluate the control points of $\mathbf{G}$, we must ensures that the degree of $\mathbf{G}_N$ is the same as that of $\mathbf{G}_D$. Hence, we elevate the u and v degrees [Far93], respectively, of $\mathbf{G}_N$ by one. Now we have $\mathbf{G}$ in rational Bézier form. Coefficients of $\mathbf{G}_D$ are the weights and those of $\mathbf{G}_N$ are the weighted

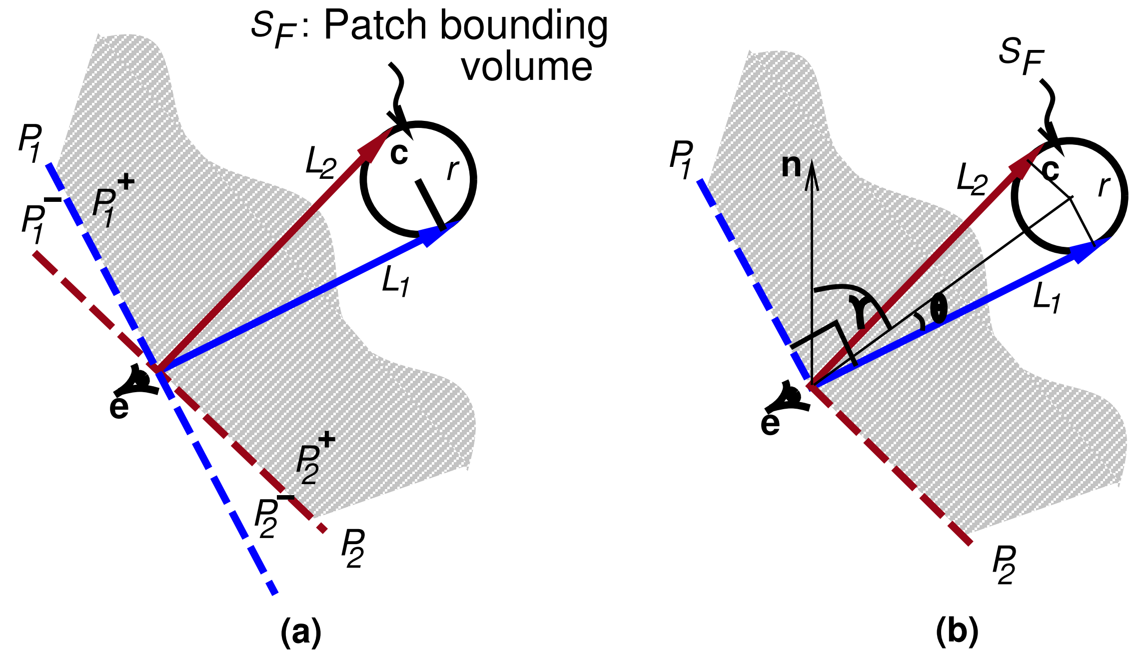

Figure 3.3: The back-region of a patch

control points. Note that in practice we do not need to divide by the weights of the pseudo-normal patch as the vectors $[X\ Y\ Z]$ and $[WX\ \ WY\ \ WZ]$ are parallel. This means that the open pyramid anchored at the origin that bounds the normal surface remains the same. Hence all the normal directions are still contained in the convex hull (or bounding box) of $[WX\ \ WY\ \ WZ]$.

We pre-compute and store V_N, a bounding volume for the pseudo-map. At display time, given the viewing direction $\mathbf{v}$, we compute the half-plane $\mathbf{H}$ and check whether all the points of V_N lie in $S1$. If V_N lies in $S1$, we mark $\mathbf{F}$ as a back-patch.

We next generalize the back-patch computation algorithm to handle perspective projections.

3.3.2 General back-patch condition

In general a point $\mathbf{p}$ on a patch, with normal $\mathbf{n}$, is back-facing if

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

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

Consider S_F, a sphere with radius r and center $\mathbf{c}$, that bounds $\mathbf{F}$. We use S_F to compute the *back-region*, a region in space that is guaranteed to contain all normal directions of $\mathbf{F}$, if it is a back-patch.

Figure 3.3(a) demonstrates this in 2D. $\mathbf{e}$ is the eye-point. In the following discussion, assume all normals are rooted at $\mathbf{e}$. The rays L_1 and L_2 bound S_F, and lines P_1 and P_2 are, respectively, perpendicular to L_1 and L_2. A normal direction, $\vec{\mathbf{n}}$, lying in either P_1^- or P_2^- may form an obtuse angle with a vector $\vec{\mathbf{vp}}$, where $\mathbf{p} \in S_F$. If $\vec{\mathbf{n}} \in P_1^+ \cap P_2^+$, it is guaranteed to make an acute angle with $\vec{\mathbf{ep}}$ for every point $\mathbf{p} \in S_F$. Hence the intersection of half-spaces P_1^+ and P_2^+, call it $\mathbf{H}$, is the back-region (shaded in the figure).

Thus, if the angle between a normal direction $\vec{\mathbf{n}}$ and $\vec{\mathbf{ec}}$ is smaller than that between P_1 and $\vec{\mathbf{ec}}$, $\vec{\mathbf{n}}$ lies in the back-region (see Figure 3.3(b)). Mathematically, this means:

$$\begin{aligned}
\gamma &< \frac{\pi}{2} - \theta \;\; (\gamma < \frac{\pi}{2}, \theta < \frac{\pi}{2}) \\
\Rightarrow \cos(\gamma) &> \cos(\frac{\pi}{2} - \theta) \\
\Rightarrow \cos(\gamma) &> \sin(\theta) \\
\Rightarrow \frac{\vec{\mathbf{ec}} \cdot \vec{\mathbf{n}}}{|\vec{\mathbf{ec}}||\vec{\mathbf{n}}|} &> \frac{r}{|\vec{\mathbf{eC}}|} \\
\Rightarrow (\mathbf{c} - \mathbf{e}) \cdot \vec{\mathbf{n}} &> r|\vec{\mathbf{n}}|
\end{aligned} \tag{III.2}$$

Notice that we assert $\gamma < \frac{\pi}{2}$ and $\theta < \frac{\pi}{2}$. This is required for the back-region test to be useful. As can be seen from Figure 3.3(b), larger values of γ or θ will result in trivial rejection and $\mathbf{F}$ would not be marked back-facing. In practice, this condition holds for many real world models that typically consist of a large number of relatively flat patches. In case the condition does not hold for a patch, we subdivide it into smaller sub-patches until the condition holds for the sub-patches.

Notice also that $\mathbf{e}$ is the only part of this expression that is not known until run-time. The rest of the expression is pre-computed off-line and stored. Thus at run-time the back-region test performs one vector difference and one dot product.

Also note that the test as described above is only for one normal value, $\vec{\mathbf{n}}$. Of course, normal direction at each point on $\mathbf{F}$ must lie in the back-region for $\mathbf{F}$ to be marked back-facing. Although there are infinite such normals, we can compute a bound on these normals and perform the back-patch test on the bound. The pseudo-Gauss map, $\mathbf{G}$ is useful for this purpose. Since each normal direction of $\mathbf{F}$ corresponds to a point on $\mathbf{G}$, if $\mathbf{G}$ lies completely in the back-region, $\mathbf{F}$ is a back-patch. We can test this by computing a bounding volume for $\mathbf{G}$. We elaborate this in the next section.

3.3.3 Choice of bounds

If B_N is a minimum volume bounding box for $\mathbf{G}$, and each corner of B_N satisfies the back-patch test, $\mathbf{F}$ is guaranteed to be back-facing. Thus at most eight back-region tests are required to determine if $\mathbf{F}$ is back-facing. If, instead, we use the convex hull of $\mathbf{G}$, we get less conservative results as the convex hull forms a tighter bound around $\mathbf{G}$. On the other hand, we must trade this increased effectiveness for a loss of efficiency. One back-region test is required for each point on the hull, and there typically are more than eight. Investigating this trade-off further, if we used a spherical bound on $\mathbf{G}$, say S_N with radius r_N and center $\mathbf{c}_N$, the number of tests reduces to one. We perform the test only for $\mathbf{c}_N$. The maximum deviation a normal direction may have from $\mathbf{c}_N$ is $\rho = \cos^{-1}(\frac{r_N}{|\mathbf{c}_N|})$. This means all points inside S_N satisfy the back-region test if:

$$\cos(\gamma) \;>\; \sin(\theta + \rho) \qquad \text{(see Equation III.2)}$$

Again, θ and ρ are determined independently of the viewing parameters, and are pre-computed and stored. Table 3.2 lists the performance of back-patch culling on various models. Notice that some models have almost half of the patches culled away. Since most patches are relatively flat, we have found that bounding boxes routinely out-perform convex hulls. For a rational bicubic patch, the control polytope of **G** consists of 81 control points, and its convex hull typically has about half as many points. In comparison, the bounding box has only eight points. The spherical bounding volume performs better than the bounding box on average, even though the tests are less precise. The number of tests per patch reduces to one, while the average culling decreases from 36% to 29%.

Model	Num Patches	Spherical bound Percent culled	Spherical bound #Tests/ patch	Cuboidal bound Percent culled	Cuboidal bound #Tests/ patch	Convex Hull Percent culled	Convex Hull #Tests/ patch
Pencil	570	33%	1	35%	1.44	36%	1.67
Dragon	5354	32%	1	38%	1.80	44%	2.11
Car	10,012	22%	1	29%	1.66	33%	1.75
Sub-room	21,900	32%	1	42%	1.67	NA	NA

Table 3.2: The effect of different bounds on the pseudo-normal patches

For small systems where calculation or storage of extra bounding boxes is too expensive, the view-frustum can be used instead of the bounding volume for each patch, as the result of the test on parts of a patch outside the view-frustum is not useful; those parts are invisible anyway. The test still remains the same, except the same back-region is used for every patch. This strategy is specially effective for applications that use narrow fields of view.

There are two potential inefficiencies with the back-patch computation algorithm described in this section:

1. Even for trimmed patches, the algorithm computes the normals for the complete $[0,1] \times [0,1]$ domain. Thus, at times, a patch is not marked back-facing even if only the trimmed-out part of the patch may be facing towards the user.

2. For values of $\mathbf{e}$ close to $\mathbf{c}$, θ becomes large, making the test quite conservative.

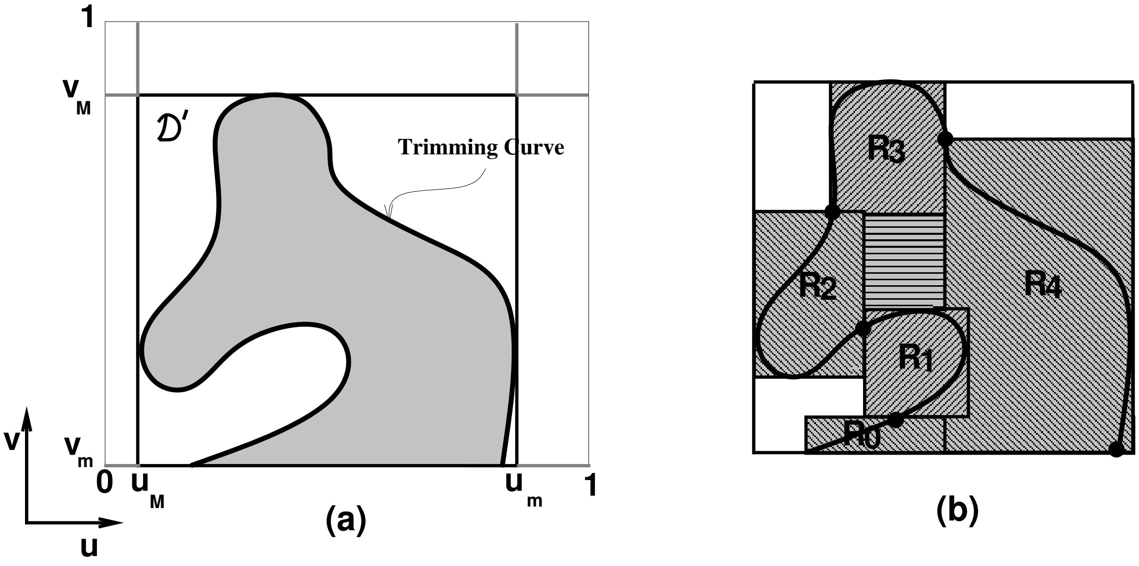

Figure 3.4: Increasing the detection of trimmed back-patches

The solutions to these problems are discussed in the following sections.

3.3.4 Trimmed surfaces

Trimming curves are defined on the domain of trimmed surfaces as shown in Fig. 3.4(a). The trimmed domain, $\mathcal{D}_s$, is a subset of $\mathcal{D} = [0,1] \times [0,1]$. Using $\mathcal{D}$ for visibility computation can lead to reduced effectiveness — a trimmed patch may be back-facing while the un-trimmed patch is not. We define *fullness* of a trimmed domain, $\mathcal{D}_s$, as the ratio of its area with that of $\mathcal{D}$, $\frac{\mathcal{A}(\mathcal{D}_s)}{\mathcal{A}(\mathcal{D})}$. $\mathcal{A}$ denotes area. We do not compute exact fullness, but estimate it by tessellating the trimming curves into piecewise linear segments, and calculating the enclosed area. The first step to increase the fullness involves the computation of the tight-fitting domain, $\mathcal{D}'$ (Fig. 3.4(a)):

> Compute the minima and maxima of the trimming curve in the domain in both u and v coordinates, respectively, $u_m, u_M, v_m,$ and v_M. The new domain, $\mathcal{D}'$, is obtained by subdividing the original patch at $u = u_m, u = u_M, v = v_m$ and $v = v_M$.

Unfortunately, for many trimmed patches this does not result in a good bound. The domain of such patches are further subdivided as follows:

1. Decompose each curve by subdividing it at its inflection points.

2. Find the closest fitting rectangular box around each component of the curve. $\{\mathbf{R}_0 \cdots \mathbf{R}_k\}$ (Fig. 3.4(b)).

3. Add rectangles to fill the region $\mathcal{D}' = \mathcal{D} - \cup\{\mathbf{R}_i\}$.

4. If $fullness(\mathcal{D}') < K$, we recursively subdivide $\mathcal{D}'$, where K is a user-defined constant. We have found $K = \frac{3}{4}$ to work well in our implementation.

This subdivision decomposes a patch $\mathbf{F}$ into a set of possibly overlapping sub-patches, say, $\{\mathbf{F}_1, \mathbf{F}_2 \ldots\}$. One possibility is to treat these patches as separate primitives, and perform visibility culling on each of them. Unfortunately, this method leads to patch-proliferation and inefficiency. In addition, it also complicates the triangulation algorithm. Furthermore, extra processing is required to prevent cracks in the rendered image. Instead, we compute a bounding volume for each sub-patch, and finally merge them back for the back-patch test by using the union of the volumes. This is more effective than using the entire domain $\mathcal{D}$. The decomposition heuristics that we have found to work well are:

- Do not subdivide $\mathcal{D}'$ for more than 3 levels.
- Always merge sub-domains that overlap.
- In general, avoid merging well separated sub-domains. If the number of such sub-domains is large, merge the closest ones until no more than $3 - 4$ are left.

3.3.5 Analytic Bound

The back-patch condition outlined in Section 3.3.2 can be applied to any type of surface, not just Bézier patches, as long as we know how to find a bound on normals. For specific cases we can apply that condition in an analytic manner. For example, for Bézier surfaces, the visibility can be tested as follows:

A patch is back-facing if, $\forall (u, v) \in [0, 1] \times [0, 1]$,

$$
\begin{aligned}
& \mathbf{EF}(u,v) \cdot \mathbf{N}(u,v) > 0 \\
\Rightarrow\ & (\mathbf{F}(u,v) - \mathbf{E}) \cdot \mathbf{N}(u,v) > 0 \\
\Rightarrow\ & \mathbf{F}(u,v) \cdot \mathbf{N}(u,v) - \mathbf{E} \cdot \mathbf{N}(u,v) > 0 \\
\Rightarrow\ & \mathbf{F}(u,v) \cdot \mathbf{N}(u,v) > \mathbf{E} \cdot \mathbf{N}(u,v) \qquad \text{(III.3)}
\end{aligned}
$$

The LHS and RHS of Equation (III.3) can each be written as Bézier functions. The visibility test reduces to checking if there exists a $(u,v) \in [0,1] \times [0,1]$ such that the bivariate function $(\mathbf{F}(u,v) - E) \cdot \mathbf{N}(u,v) \leq 0$, which, in turn, can be solved by testing if the function has any roots in the domain $[0,1] \times [0,1]$. This is a high degree function, $4m + 4n$ for an $m \times n$ Bézier surface, and solving for exact roots leads to efficiency and accuracy problems. In addition, this operation is too expensive to be performed at interactive rates. Fortunately, we can bound the function by the minimum and maximum valued control points (using the Bernstein basis). This results in the following algorithm:

1. Pre-compute m_F, the minimum value of $(\mathbf{F}(u,v) \cdot \mathbf{N}(u,v))$. m_F is the minimum control point of the scalar function $\mathbf{F} \cdot \mathbf{N}$. The control points of $\mathbf{F} \cdot \mathbf{N}$ are computed as follows:
   - Compute the $[X \quad Y \quad Z]$ control points of $\mathbf{N}$ using the algorithm in section 3.3.1.
   - Pairwise multiply $[WX \quad WY \quad WZ]$ functions of $\mathbf{F}$ and $\mathbf{N}$. This results in the rational control points of $\mathbf{F} \cdot \mathbf{N}$.
   - Elevate the degree of W to $4m \times 4n$.
   - Divide each rational control point of $\mathbf{F} \cdot \mathbf{N}$ by the corresponding control point of W.
2. Pre-compute and store the maximum value of each coordinate, respectively, of $\mathbf{N}$(u,v). Call the vector of the three maxima, $\mathbf{M_N}$.
3. At run-time, if $m_F > \mathbf{M_N} \cdot \mathbf{E}$, the patch is back-facing.

In general, the analytic method results in slightly more conservative estimates than the pseudo-map method described earlier, but it does not degrade in performance for values of **e** close to **c**, as the pseudo-map method does.

3.4 Hierarchical extension

Hierarchical spatial partitioning [Cla76, FKN80, TFFH94, KM96] is routinely used in computer graphics to speed up visibility computations. We employ this well known technique and use a hierarchy of bounding boxes. The extension for view-frustum clipping is straightforward. We group patches, and compute volumes tightly enclosing the group. If a group intersects the boundary of the frustum, we test smaller sub-groups of patches and proceed recursively until the decision to retain or cull each patch is reached. We construct a similar hierarchy for back-patch culling.

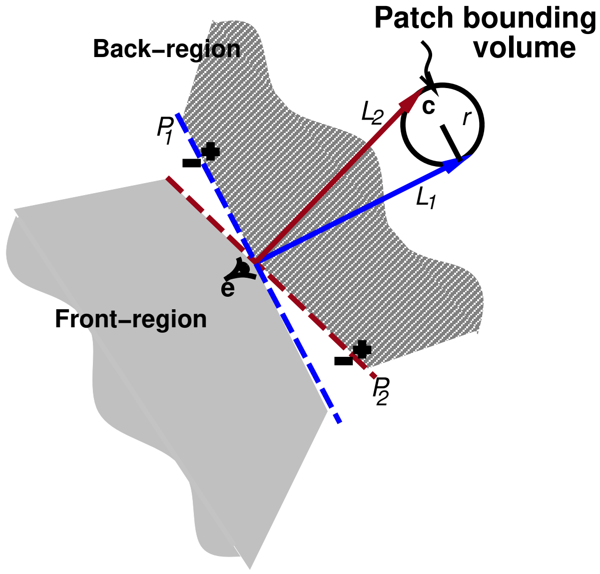

Figure 3.5: The Front-region of a patch

3.4.1 Hierarchical back-patch computation

Just as we can determine if a patch is completely back-facing, we can also determine if a patch faces completely towards the user, i.e. if it is completely front-facing. The advantage of detecting front-patches is that if a group of patches are all known to be front-facing, we do not need to test the visibility for its sub-groups. Front-region of a patch **F** is just a reflection of its back-region about the eye-point as shown in Figure 3.5. The argument is the same as that for back-region

> If a normal direction $\vec{\mathbf{n}}$ lies in the front-region, it is guaranteed make an obtuse angle with all vectors in the cone bounded by L_1 and L_2.

Now, in addition to the standard hierarchy of object bounding volumes, we maintain a hierarchy of pseudo-Gauss maps. We call this the *Normal* hierarchy, as opposed to the standard *object* hierarchy. Our approach of using bounding box hierarchy is similar to R-trees [BKSS90] and exhibits logarithmic traversal time on average. This approach greatly simplifies the merging of the normal and object space hierarchies into one structure. Thus we have to store and traverse only one tree. The object-space hierarchy is constructed based on object-space adjacency. In fact, for normal space partitioning also, grouping patches too far apart in object space renders the method ineffective. Thus object-space proximity is important for both hierarchies. We exploit this to construct a single hierarchy for both spaces.

3.4.2 Combined data structure

The optimal hierarchies for view-frustum and back-patch culling are different. It is possible to maintain two hierarchies, one for view-frustum culling and one for back-patch culling, but this requires storage and traversal of two trees. Since proximity is still the basic criterion for both trees, we are actually able to merge them meaningfully into one structure. We call this a *One*-tree or an Object-Normal Exclusion tree. It facilitates the exclusion of invisible objects from further processing, and is based on both object and normal spaces.

Since it is not always prudent to group objects based on a combination of the two spaces, we maintain the freedom to group objects based entirely on one space. A small sphere can never be completely back-facing, but it could still be view-frustum culled. Similarly about half of nearly parallel patches have a high probability of facing

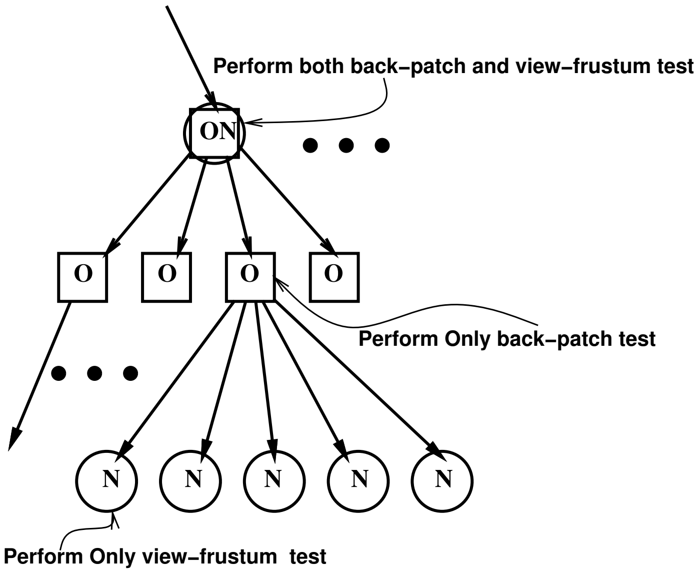

Figure 3.6: *One*-Tree

the same way but they may not be close in object space. Hence we allow different criteria for grouping at different nodes of the tree. This leads to three different types of nodes in a ONE-tree (see Figure 3.6) —

- O-nodes: Perform only view-frustum test.
- N-nodes: Perform only back-patch test.
- ON-nodes: Perform both tests.

Allowing different node types can potentially increase the tree size, but it allows us to avoid performing potentially ineffective tests. The type of a node is determined at the time of hierarchy construction. More details on the construction and traversal of ONE-trees can be found is [KM96].

Chapter IV

Surface Sampling

4.1 Overview

To display a NURBS surface, we dynamically compute a triangulation of points on the surface as a function of the viewing parameters. In each frame, these points can be picked by uniformly or adaptively sampling the domain. Uniform sampling involves choosing constant step sizes along each domain parameter and evaluating points on a regular grid in the parametric domain of a patch. It normally generates more triangles than are necessary for a smooth image of a given scene. By comparison, adaptive sampling generates fewer triangles than uniform sampling. Adaptive sampling algorithms normally use a recursive subdivision approach with stopping criteria based on flatness and surface area:

- If the patch satisfies the stopping criteria, approximate it with a quadrilateral (or two triangles).
- Otherwise, subdivide the patch into smaller patches. Tessellate each patch recursively.

Thus the stopping criteria are based on the local properties of the surface. Other techniques like adaptive forward-differencing [SC88] are, similarly, based on local properties. On the other hand, uniform sampling methods use global surface properties to determine their sampling size. This normally results in a conservative estimate of the required size, but the estimation is be done only once per patch. That is the prime reason why uniform subdivision methods are generally faster than adaptive methods in practice [FMM86]. We use uniform subdivision. (Chapter VII presents more detailed analysis of the reasons why we chose uniform sampling.)

This chapter presents the algorithm to compute the sampling size required for smooth display of a trimmed Bézier patch. For each patch, we compute n_u and n_v where $n_u \times n_v$ is the grid size on the domain. For each trimming curve of a patch we compute n_t, the number of piecewise linear segments it must be tessellated into. Using results from elimination theory we are able to quickly compute the sampling density required for a patch.

4.2 Quality Definition

The performance of uniform tessellation algorithms is a direct function of the step sizes. Our problem, in this chapter, is to find the number of sample points needed for a good tessellation of a surface, given the current viewing parameters. A number of criteria can be prescribed to define the quality of tessellation and thus derive the step size. We assume all these criteria are specified in screen coordinates.

1. *Size criterion*: Each edge of the approximating triangles is bounded by a user-specified constant.

2. *Tangent criterion*: The change of tangent between consecutive samples is bounded.

3. *Normal criterion*: The difference between surface normals at consecutive samples is bounded.

4. *Deviation criterion.* The maximum deviation of triangles from the surface is bounded.

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

Figure 4.1: Image quality based on triangle size

The criteria listed above are related to each other and not all of them need to be satisfied for a given application. For our application (rendering using Phong or Gouraud shading) the surface deviation and normal deviation criteria are the most important. The primary effect of a bounded surface deviation is the smooth appearance of silhouettes, whereas a bounded normal deviation bounds the error in the interpolated intensity value at any point in the image. Unfortunately, these criteria require the computation of second and higher order surface-derivatives and their cross products. For rational bicubic surfaces, this leads to expressions of degrees more than 100, which are both costly and numerically unstable to evaluate. On the other hand, the triangle size criterion is much easier to satisfy. Moreover, in general, small triangle sizes approximately satisfy other criteria as well. For example, the smaller the triangle size gets, the closer the approximation is to the surface. In the limit, the triangular approximation approaches the surface. The basic problem is to guess the size of triangles that results in smooth images. In our experience, for a high resolution screen (1024×1280) pixels, triangles with less than 50 pixels on each side generate smooth images using color interpolation. If a polygonal approximation of a Bézier model satisfies our quality criteria we call it a *satisfactory instance*. Figure 4.1 demonstrates this. The rightmost image is the reference image generated with triangles close to a pixel in size. The image on the left is Gouraud interpolated and has all triangles less than a 100 pixels on their side, while the image in the middle column has triangles with edges up to 50 pixels long.

At times the triangle size criterion results in undersampling as shown in Figure 4.2. Even though each step is small enough on screen, the curve is not well approximated. This happens for surfaces with high curvature. We add an extra factor to account for such cases: for surfaces with high curvature we reduce the size of approximating triangles as described in Section 4.3.3.

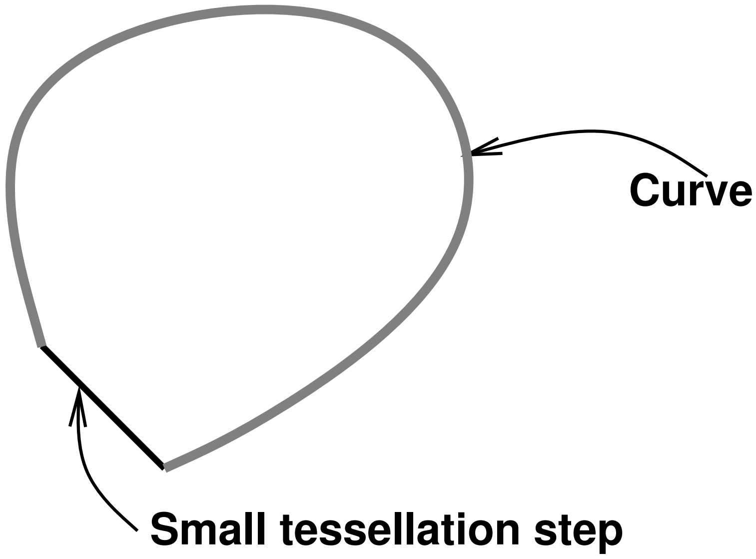

Figure 4.2: Undersampling of a curve with high curvature

4.3 Sampling size computation

To compute the step size in the parametric domain so as to satisfy our quality criteria, we need to compute bounds on polynomials. There is considerable literature on computation of such bounds [LR81, FMM86, Roc87, AES91]. In particular, Rockwood et al. [Roc87] and Abi-Ezzi et al. [AES91] use these bounds to compute the sampling size. Suffering from the use of loose bounds, both these methods often oversample the surface. Our method improves the tightness of bounds thus resulting in less oversampling. (See Chapter VII for more details on these comparisons.)

4.3.1 Computation space

Note that the quality criteria are specified in screen space, whereas we perform our sampling in the parametric domain of the surface. This means that we need to find a relationship between the step size on the domain and the resulting triangle size on the screen. Consider the domain $\mathcal{D}$ of a surface $\mathbf{F}$, and points p_1 and p_2 on $\mathcal{D}$. Assume $T(\mathbf{F}(p))$ maps the point $\mathbf{F}(p)$ to screen space. Also assume, $|T(\mathbf{F}(p_1))T(\mathbf{F}(p_2))| = f_1(|\mathbf{F}(p_1)\mathbf{F}(p_2)|)$ and $|\mathbf{F}(p_1)\mathbf{F}(p_2) = f_2(|p_1p_2|)$. We are interested in the function $f_3 = f_1 \circ f_2$. Note that f_2 depends only on the control points of $\mathbf{F}$ and is independent of the viewing parameters. We may compute f_3 in two ways:

1. Transform the surface (control points) into screen space. Use the transformed representation to compute the bounds on derivatives, and the step sizes as a function of these bounds [Roc87, RHD89].

2. Determine bounds on the surface in object space as a pre-processing step. At run time, compute the step size as a function of the pre-computed bounds and viewing parameters [LR81, FMM86, AES91].

The first method finds f_3 in one step and all computation must be performed at rendering time. On the other hand, the second method factors f_3 into two, f_1 and f_2. We first find the relationship between the domain step size and the step size in the object space, which is independent of the viewing parameters and may be pre-computed and stored. The second step maps the triangle sizes in the object space to that in the screen space in order to compute the sampling size at rendering time. f_1 depends on how much the viewing and perspective transformations scale the lengths of vectors. We compute the minimum scaling over all vectors that lie in the intersection of the view-volume and the convex hull of the surface. For parallel projection, and a linear transformation matrix, the relationship is quite simple, as the scaling is given by the norm of the matrix. For perspective projection, the minimum scaling occurs at one of the hull points. (We may using a bounding box instead of a convex hull to reduce the computational costs.) Given Δ_o, the step size in the object space, we compute Δ_s, the maximum step size in screen space as follows:

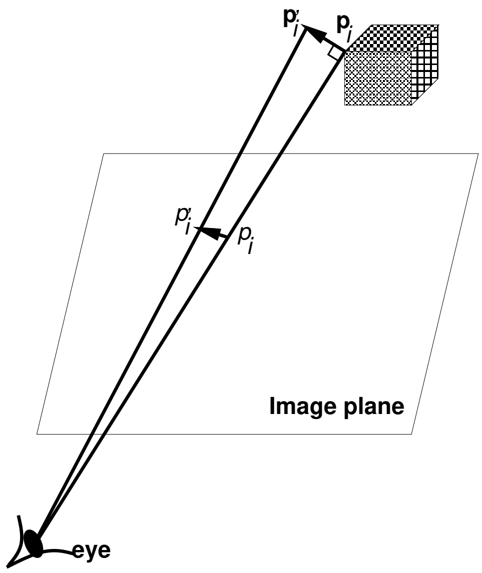

Figure 4.3: Scaling behavior of perspective transformation

For each point $\mathbf{p}_i$ on the hull we compute r_i, the length of the major axis of the ellipse projected onto the screen by the circle, c_i, with center $\vec{\mathbf{p}}_i$ and radius Δ_o. c_i lies in a plane perpendicular to $\vec{\mathbf{p}_i\mathbf{e}}$, where $\mathbf{e}$ is the eye-point. Note that a vector perpendicular $\vec{\mathbf{p}_i\mathbf{e}}$ scales the least on projection. $\Delta_s = \min_i(r_i)$.

We require only two mathematical formulations for our step size computations: one for the computation of the difference bounds (criteria 1-3), and the other for the deviation bounds (criteria 4 and 5). We first demonstrate the formulation of the difference bounds computation using the triangle size criterion. The next section provides the framework for deviation bound computation.

4.3.2 Difference criteria

A rational Bézier surface is given as

$$\mathbf{F}(u,v) = (X(u,v), Y(u,v), Z(u,v), W(u,v)) =$$
$$\left(\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}\mathbf{r}_{ij}\mathcal{B}_i^m(u)\mathcal{B}_j^n(v), \sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}\mathcal{B}_i^n(u)\mathcal{B}_j^m(v)\right), \tag{IV.1}$$

where $\mathbf{r}_{ij}$ are the control points in object space, w_{ij} are the weights and $\mathcal{B}_i^m, \mathcal{B}_j^n$ are the Bernstein polynomials. After applying all of the viewing transformations (rotations, translation, perspective), let the new control points in screen space be: $\mathbf{R}_{ij} = (X_{ij}, Y_{ij}, Z_{ij})$ and let W_{ij} be the corresponding new weights. Let Δ be the required step size in screen space. For adjacent samples to be closer than Δ, the step sizes along the u and v parametric directions are given as [RHD89]:

$$n_u = m\frac{\max \parallel W_{ij}\mathbf{R}_{ij} - W_{i+1,j}\mathbf{R}_{i+1,j} \parallel}{\Delta * min(W_{ij})}$$
$$n_v = n\frac{\max \parallel W_{ij}\mathbf{R}_{ij} - W_{i,j+1}\mathbf{R}_{i,j+1} \parallel}{\Delta * min(W_{ij})} \tag{IV.2}$$

for $(1 \leq i \leq m, 1 \leq j \leq n)$.

In practice these bounds are good for polynomial surfaces only, when $W_{ij} = 1$. However, after the perspective transformation, all the polynomial parametrizations are transformed into rational formulations. Furthermore, if the weights of a Bézier patch vary considerably (typically by a factor of two to three), loose bounds cause an oversampling of the surface. Oversampling implies an increase in the number of triangles generated, which can lead to rendering bottlenecks.

We compute improved bounds for the rational surfaces in object space as part of the pre-processing phase. They are used to compute the step sizes as a function of the viewing parameters as shown in [FMM86, AES93]. An algorithm for computation of bounds based on the size criterion is presented in [AES91]. However, the bounds in [AES91] should be modified to use the mean value theorem for vector valued functions, as we show next. In contrast, our method computes the exact extrema (as a part of pre-processing). Hence, for a given Δ, our bounds are tighter. We illustrate the derivation of our bounds on a Bézier curve (it is applied in a similar manner to the surfaces). Given a rational curve

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

let

$$X(t) = \frac{x(t)}{w(t)}, \ldots, Z(t) = \frac{z(t)}{w(t)}.$$

Given a step size δ in the domain, we want to compute tight bounds on the length of the vector $\mathbf{C}(t+\delta) - \mathbf{C}(t)$. It follows from the mean value theorem that

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

where $t_1, t_2, t_3 \in [t, t+\delta]$. t_1, t_2 and t_3 need not be equal (as assumed in [AES91]). As a result

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

where $\overline{X}^{'}(t), \overline{Y}^{'}(t), \overline{Z}^{'}(t)$ represent the maximum magnitude of $X^{'}(t), Y^{'}(t), Z^{'}(t)$, respectively, in the domain $[0,1]$ and $\|V\|$ is the L_2 norm of the vector V. Given these maximum magnitudes of the derivatives and Δ, we choose the step size δ satisfying the relation

$$\delta \leq \frac{\Delta}{\| \overline{X}^{'}(t), \overline{Y}^{'}(t), \overline{Z}^{'}(t) \|}.$$

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

$$n_u = \frac{\left\| \overline{\frac{X(u,v)}{W(u,v)}}_u, \overline{\frac{Y(u,v)}{W(u,v)}}_u, \overline{\frac{Z(u,v)}{W(u,v)}}_u \right\|}{\Delta},$$

where $\overline{\frac{X(u,v)}{W(u,v)}}_u$ corresponds to the maximum magnitude of the partial derivative of $\frac{X(u,v)}{W(u,v)}$ with respect to u in the domain $[0,1] \times [0,1]$. n_v is computed analogously.

The maximum values of the partial derivatives are computed in the following way. Let

$$fx(u,v) = \left(\frac{X(u,v)}{W(u,v)} \right)_u = \frac{(X_u(u,v)W(u,v) - X(u,v)W_u(u,v))}{W(u,v)^2}.$$

$fy(u,v)$ and $fz(u,v)$ are defined in a similar manner. The maximum of $fx(u,v)$ in the input domain corresponds to one of the common roots of

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

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

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

It may also occur at $fx(0,0), fx(0,1), fx(1,0)$ or $fx(1,1)$. We find all maxima and pick the maximum of those.

Thus the problem of computing the maximum derivative vector is equivalent to computing the zeros of polynomial equations. For the maxima of fx, the two equations are:

$$\begin{aligned}
X_{uu}W^2 - XWW_{uu} - 2W_uX_uW + 2XW_u^2 &= 0, \\
X_{uv}W^2 - XWW_{uv} - W_vX_uW + 2W_uXW_v - W_uX_vW &= 0.
\end{aligned}$$

The degrees of these curves are $(3m-2, 3n)$ in (u, v) for the first curve and $(3m, 3n-2)$ in (u, v) for the second curve. This is rather high, but using the method described in section 2.5 we are able to compute accurate solutions without numerical problems. Note that all these computations are part of the pre-processing stage.

Similarly, the maximum of $fx(0, v)$ corresponds to computing the roots of $fx_v(0, v) = 0$, which can be computed using root-finders or subdivision properties of Bézier curves [LR81]. Based on the solutions of these equations, we compute the maximum values of $fx(u, v)$ in the domain $[0, 1] \times [0, 1]$. Let the maximum value be at $[u_x, v_x]$. $fy(u, v)$ and $fz(u, v)$ are computed similarly. The complexity of the bound computations reduces significantly for polynomial surfaces, as $W(u, v) = 1$ and the resulting equations are of much lower degrees.

Given these bounds in object space, we compute the step size in screen space as a function of the viewing transformations. These bounds are invariant with respect to rigid body transformations like rotations and translations but vary with the perspective transformation.

The computation of the bounds for the tangent and normal difference criteria can be computed similarly. In one case we seek to bound the length of the vector connecting two points on the surface, and in the others we seek to bound the angle between tangents or normals. These can be computed by starting with $\mathbf{F}_u, \mathbf{F}_v$ and $\mathbf{F}_u \times \mathbf{F}_v$, respectively, instead of $\mathbf{F}$. Owing to the expense of such computations, we do not explicitly satisfy any of these criteria in our implementation. In practice, we do not notice any artifacts as a result of that.

4.3.3 Deviation criteria

For small values of Δ, the size-criterion bound derived above works quite well. In case the surface area is small and curvature is high, using only the triangle size criterion results in an undersampling of the surface (Figure 4.2). The solution to this problem involves the deviation criterion. The bound on deviation is computed using results from [FMM86]:

> For a linearly parametrized triangle $T = l(u, v)$ between three points on a surface at $l(0,0), l(l_1, 0)$ and $l(0, l_2)$:
>
> $$\max_{(u,v)\in T} \|\mathbf{F}(u,v) - l(u,v)\| \leq \frac{1}{8}(l_1^2 M_1 + 2l_1 l_2 M_2 + l_2^2 M_3),$$
>
> 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 again reduce the computations of M_1, M_2 and M_3 to finding zeros of polynomials and solve them using techniques from elimination theory [MD92]. However, in practice this method typically oversamples the surface as the degrees of the polynomials are very high and the bounds become loose (often by a factor of two or more). Therefore we instead use a simple estimation based on the geometry of the transformed control points in screen space. To the number of steps, n_u, computed for the size criterion (using the method detailed in the previous section) we add:

$$K \times \max \left\{ \begin{array}{l} ||\mathbf{F}_u(\epsilon_u, 0) - \mathbf{F}_u(0,0)|| \\ ||\mathbf{F}_u(\epsilon_u, 1) - \mathbf{F}_u(0,1)|| \\ ||\mathbf{F}_u(1,0) - \mathbf{F}_u(1-\epsilon_u, 0)|| \\ ||\mathbf{F}_u(1,1) - \mathbf{F}_u(1-\epsilon_u, 1)|| \end{array} \right\}$$

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

$$K \times \max \left\{ \begin{array}{l} ||\mathbf{F}_v(0, \epsilon_v) - \mathbf{F}_v(0,0)|| \\ ||\mathbf{F}_v(1, \epsilon_v) - \mathbf{F}_v(1,0)|| \\ ||\mathbf{F}_v(0,1) - \mathbf{F}_v(0, 1-\epsilon_v)|| \\ ||\mathbf{F}_v(1,1) - \mathbf{F}_v(1, 1-\epsilon_v)|| \end{array} \right\}$$

to n_v. These formula use the variance in derivative at the patch corners to estimate the curvature for the complete patch. This works well in practice because these numbers provide a fair estimate of the curvature of the surface for most real life models as most surfaces do not have highly varying curvature.

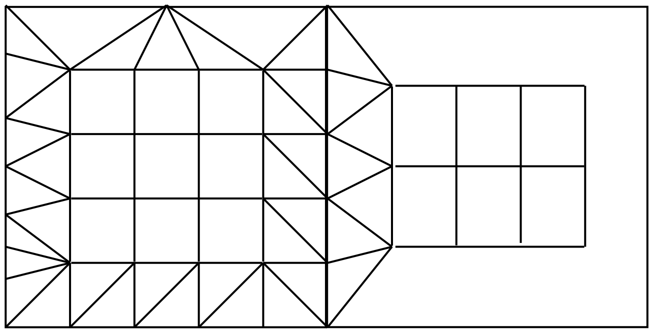

Figure 4.4: Filling at boundaries

4.4 Crack prevention

Since the sampling size for each patch is evaluated independently, different samplings of two adjacent patches are possible. This would result in cracks in the rendered image (Figure 1.5). To address this issue [FMM86, RHD89] suggest that the amount of tessellation at the boundary be based solely on the boundary curve, and a strip of filling triangles be generated at the boundary as shown in Figure 4.4. However, this method works only if the common boundary curves of the two adjacent patches have exactly the same parametric representation in terms of the control points. One common counter-example occurs when one of the patches is subdivided into two, resulting in a T-joint at the common boundary (Figure 4.5).

4.4.1 T-joints

At T-joints, there is no way to prevent cracks without using information about the adjacent patches.

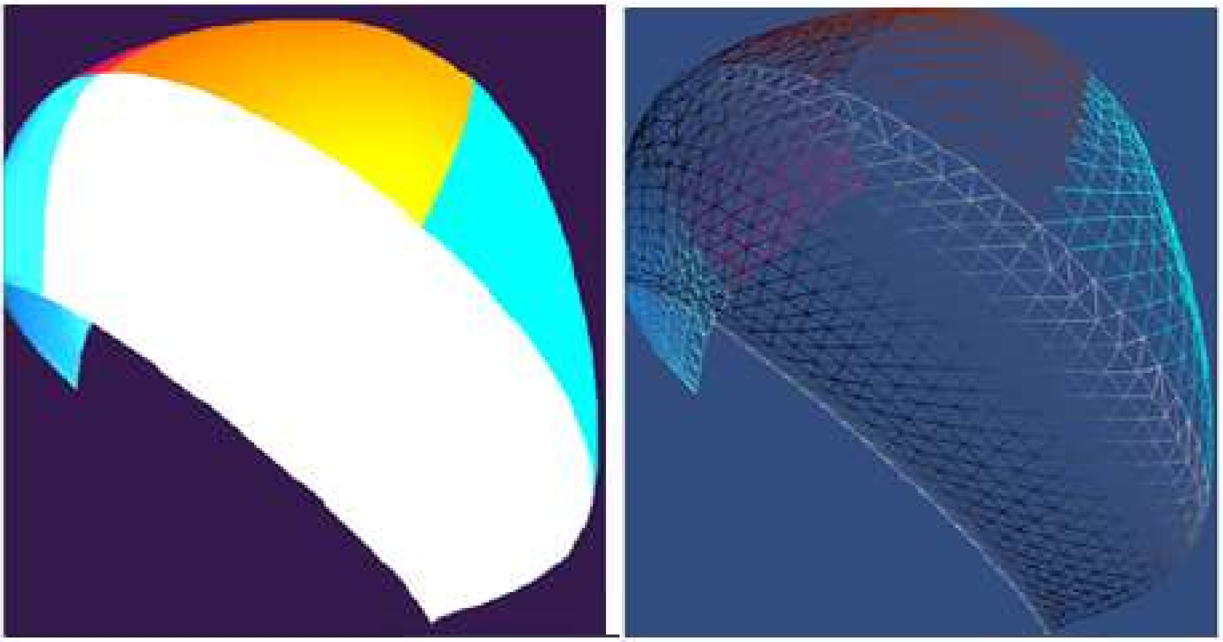

Figure 4.5: T-Joint

To illustrate this fact, suppose we calculated the required tessellation for a boundary curve of a patch $\mathbf{F}$. We can always subdivide one of the patches adjacent to this boundary, say $\mathbf{F}'$, into two sub-patches, say $\mathbf{F}'_1$ and $\mathbf{F}'_2$. Now it is straightforward to reparametrize each of them in such a way that the tessellation points on the boundaries of $\mathbf{F}'_1$ and $\mathbf{F}'_2$ are different from $\mathbf{F}'$, and hence from $\mathbf{F}$.

Our algorithm computes the adjacency information between the patches in the pre-processing phase. We assume that the patches share the same boundary curves geometrically. The algorithm computes the bounding boxes of the boundary curves and sorts them along their projections on the X, Y and Z axes to compute the overlapping pairs. For each pair of overlapping boxes the algorithm checks whether the two curves form a common boundary as follows. Let the two boundary Bézier curves be $C_1(t)$ and $C_2(t)$.

The curves are common only if

1. There exist t_0 and t_1 such that both $C_1(0) = C_2(t_0)$ and $C_1(1) = C_2(t_1)$, and

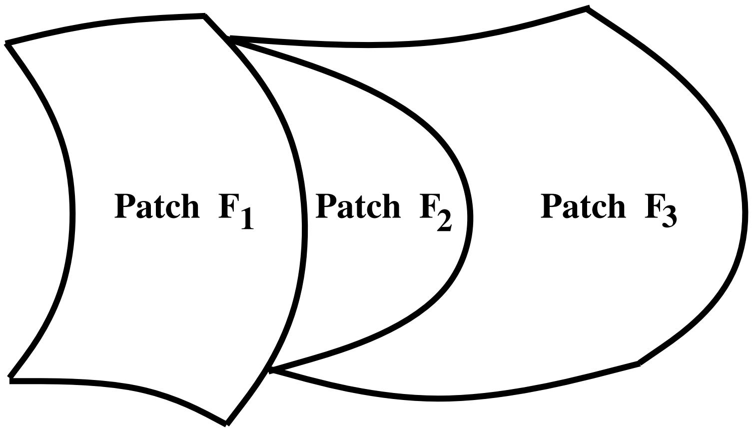

Figure 4.6: Degenerate boundary pairs: Patches $\mathbf{F}_1$ and $\mathbf{F}_3$

2. At least one of t_0 and t_1 lie in [0,1]. (Otherwise there is no overlap between parts of curve in parametric range [0,1]. Note that Bézier curves are defined only in [0,1].)

In degenerate cases these two conditions may be satisfied for boundary pairs that are not common. For example in Figure 4.6, Patch $\mathbf{F}_3$ will be declared adjacent to Patch $\mathbf{F}_1$. Assuming that the surface is a simple manifold, i.e. it is like a small disk in the neighborhood of each point, exactly one of the apparently adjacent patches is actually adjacent, Patch $\mathbf{F}_2$ in this case. Such conflicts are resolved by considering a third point (arbitrarily chosen) on the boundary curve. We cannot have three points common to two patches that do not have a common boundary curve (and do not overlap). In our example, only patch $\mathbf{F}_2$ has a third point common with $\mathbf{F}_1$, while patch $\mathbf{F}_3$ does not. Note that the third point must be geometrically different; parametric difference is not sufficient. It is possible that $\mathbf{F}_1(u_1, v_1) = \mathbf{F}_1(u_2, v_2)$ even though $(u_1, v_1) \neq (u_2, v_2)$.

Location of the points on the adjacent curve reduces to an *inversion* problem:

> Given a point $\mathbf{p}$ and a curve $\mathbf{f}$, find the parameter value t such that $\mathbf{f}(t) = \mathbf{p}$.

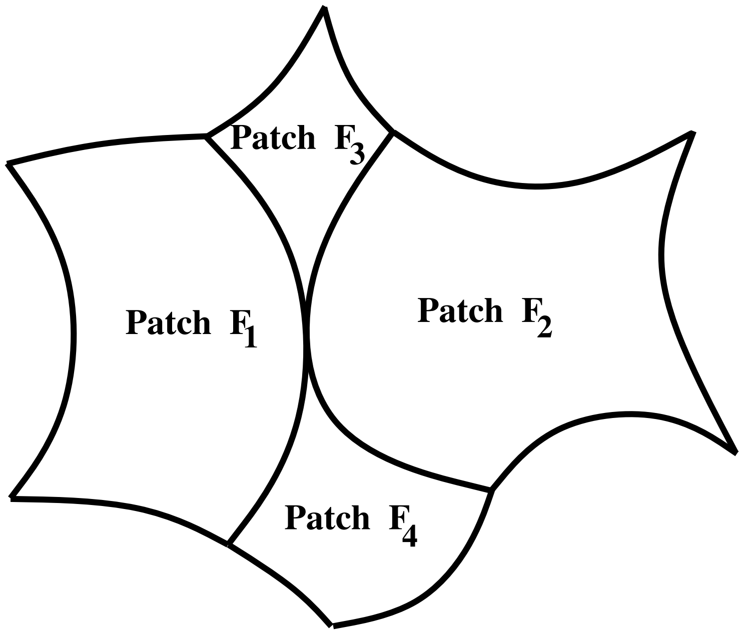

Figure 4.7: Tangent boundary pairs

Patch $\mathbf{F}_2$ is tangent to Patch $\mathbf{F}_1$. Patches $\mathbf{F}_3$ and $\mathbf{F}_4$ complete the adjacency for Patch $\mathbf{F}_1$ without including Patch $\mathbf{F}_2$.

Again using techniques from elimination theory, we first solve for t using the equation for the $\mathtt{x}$ coordinate:

$$X(t) = \mathbf{p}_\mathsf{x}$$

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

We associate one of the adjacent patches, say $\mathbf{F}_1$ (chosen arbitrarily) with the common boundary between $\mathbf{F}_1$ and $\mathbf{F}_2$. If we have two different representations for the boundary curve, we store the representation of the boundary curve of $\mathbf{F}_1$ with $\mathbf{F}_2$. To calculate the bounds on the curves and tessellate them, we use this stored representation which is the same for both $\mathbf{F}_1$ and $\mathbf{F}_2$, resulting in common samples.

Clearly, this algorithm does not detect cases when one boundary is tangent to another. Fortunately we do not need to compute such adjacencies. The topological consistency of the approximation is still maintained if the point of adjacency is always used in the tessellation. That this actually occurs can be seen by noting that on each side of the tangency, there exists a patch with one corner at the point of tangency (see Figure 4.7). Since we pick all corners as sample points, the tessellations are consistent.

4.4.2 Trimming Curves

Just as with T-joints, trimming curves admit no solution without the use of adjacencies. The problem, again, is the lack of a consistent representation. A trimming curve is defined in the parametric domain of its surface. Two Bézier curves, $\mathbf{f}_1$ and $\mathbf{f}_2$, that form a common boundary between patches $\mathbf{F}_1$ and $\mathbf{F}_2$, and have the same geometry in object space, can have arbitrarily different representations in the respective domains of $\mathbf{F}_1$ and $\mathbf{F}_2$, as shown in Figure 1.9. Since our step size determination is dependent on the representation of the curve, different tessellations of the same curve may result.

Our solution involves computing the space representation of each trimming curve. For solid models generated using CSG we use a seven-dimensional curve fitting method described in [KKMN95]. This method bounds error in the three object space dimensions, the two parametric dimensions of $\mathbf{F}_1$'s domain and the two parametric dimensions of $\mathbf{F}_2$'s domain at the same time. The method outputs a space representation of the trimming curve which is common to $\mathbf{F}_1$ and $\mathbf{F}_2$.

For each NURBS trimming curve, $\mathbf{f}$, on a patch, $\mathbf{F}$, we maintain its representation in the domain of $\mathbf{F}$, $\mathbf{f}_d$, and its representation in object space, $\mathbf{f}_s$. We use $\mathbf{f}_s$ to compute the step size of the curve, call it n_c. The sample points in object space are:

$$\mathbf{f}_s\left(t_i\right),\ t_i = \frac{i}{n_c},\ 0 \leq i \leq n_c$$

Notice that the domain triangulation (described in the next chapter) must still use $\mathbf{f}_d$. The sample points on the domain are:

$$\mathbf{f}_d\left(t_i\right),\ t_i = \frac{i}{n_c},\ 0 \leq i \leq n_c$$

While generating the object triangles, instead of evaluating $\mathbf{F}\left(\mathbf{f}_d^u(t_i), \mathbf{f}_d^v(t_i)\right)$ for each sample point $(\mathbf{f}_d^u(t_i),\ \mathbf{f}_d^v(t_i))$ on the domain of patch $\mathbf{F}$, we evaluate $(\mathbf{f}_s^x(t_i),\ \mathbf{f}_s^y(t_i),\ \mathbf{f}_s^z(t_i))$

Since the curve fitting routine ensures that

$$\|\mathbf{f}_s(t_i) - \mathbf{F}(\mathbf{f}_d^u(t_i), \mathbf{f}_d^v(t_i))\|,\ t_i \in [0,1] < \epsilon,\ \text{for } \epsilon \text{ close to } 0.$$

we are able to generate valid triangular approximations to the surface using our method. Further, since $\mathbf{f}_s$ is common to both patches $\mathbf{F}_1$ and $\mathbf{F}_2$, we guarantee a consistent tessellation of the surface.

Our method assumes the availability of the space representation of the trimming curves. Many modelers do not generate this information and do not guarantee that $\mathbf{F}_1 \circ \mathbf{f}_1 = \mathbf{F}_2 \circ \mathbf{f}_2$, if $\mathbf{f}_1$ and $\mathbf{f}_2$ represent the boundary curve between $\mathbf{F}_1$ and $\mathbf{F}_2$. In such cases we evaluate $\mathbf{F}_1 \circ \mathbf{f}_1$ and store it with both patches for use as the space representation of the boundary curve. Theoretically, for some surface parametrizations this may generate triangles intersecting each other. In practice such parametrizations are rare.

Chapter V

Triangulation

5.1 Overview

In order to effectively use the triangle rendering pipelines on current graphics systems, we approximate trimmed Bézier surfaces with triangles and display these triangles employing standard triangle rendering hardware. Apart from making efficient use of hardware for rendering triangles, our method is able to easily employ techniques such as anti-aliasing, shadow generation, texture mapping and radiosity computation, which improve the quality of the final image and are known to work well on polygons [FVFH93, Wat95]. Although the actual surface approximation resides in the three-dimensional object space, it is much easier to perform the triangulation in the two-dimensional parametric domain of a patch. Each domain triangle $[(u_1, v_1), (u_2, v_2), (u_3, v_3)]$ maps to the object triangle $[\mathbf{F}(u_1, v_1), \mathbf{F}(u_2, v_2), \mathbf{F}(u_3, v_3)]$, which is then rendered.

In this chapter we present algorithms to perform the domain triangulation. We first pick uniformly spaced samples using the step size computed in Chapter IV. The domain triangulation algorithm outputs the connectivities between the sample points. We ensure that the triangle size property is satisfied and no cracks are generated. The time complexity of our algorithm is $O(n \log^* n)$.

5.2 Constraints on triangulation

The triangulation problem can be stated as follows:

> Find a triangulation, ∇, given G_0, an $n_u \times n_v$ orthogonal grid of points and S_i, $0 \le i < q$, a set of q closed piecewise linear sequences. Sequence S_i consists of n_{t_i} points, and $\sum_0^{q-1} n_{t_i} = n_t$. All points and line segments lie in $\mathcal{D}$, the domain of the surface. The triangulation, ∇, must satisfy the following constraints:
>
> 1. The region bounded by the closed trimming curve formed by the input segments is covered by ∇.
> 2. Each triangle, T, in ∇ is bounded by the closed trimming curve formed by the input segments.
> 3. The maximum triangle-edge length is less than δ.
> 4. Each input point is a triangle vertex.
> 5. Each input segment is a triangle edge.
> 6. No triangle edges intersect.
> 7. No triangle vertex lies on a triangle edge, except at its endpoints. (A grid point coincident with a curve sample is not input to the triangulation routine.)
> 8. No Steiner points, i.e. no additional points, may be added to ∇. This requirement prevents cracks from appearing in the triangular approximation. Creation of extra sample points on a boundary can lead to inconsistent tessellation. We later show how Steiner points may be added without violating the consistency.

In order to ensure that triangle edges are less than Δ apart on the screen, we restrict the distances between adjacent sample points on the trimming curves to be less than $\frac{\Delta}{2}$, and the grid step size to be less than $\frac{\Delta}{\sqrt{2}}$ in screen space. Correspondingly, we choose domain step sizes to be, respectively, $\frac{\delta}{2}$ and $\frac{\delta}{\sqrt{2}}$. The step size of $\frac{\delta}{2}$ on trimming curves allows us to generate triangle $(c(j), c(j+1), c(j+2))$ for consecutive sample points, $c(j)$, $c(j+1)$ and $c(j+2)$, of the curve c. The step size of $\frac{\delta}{\sqrt{2}}$ on the grid allows us to use the grid diagonals as triangle edges. Using the mapping described in Chapter IV we compute the domain triangle edge size δ as a function of Δ.

A practical consideration guiding our triangulation algorithm is our need to compute a large number of triangulations, each of a moderately small size. The input size for each surface patch, $(n_u n_v + n_t)$, mostly remains in the neighborhood of 20-30. Hence we need an algorithm with small set-up cost and a low asymptotic complexity with a small constant.

5.3 Non-trimmed patches

It is straightforward to compute the domain triangulation of non-trimmed patches. Suppose the required grid size is $n_u \times n_v$, and the number of samples on the boundaries are, respectively, n_i, $i \in [0, 3]$. The grid consists of n_u *u-lines* perpendicular to the u axis and n_v *v-lines* perpendicular to the v axis of the domain (Figure 5.1). The *u-line* at $u = k$ has the value k. A *u-line*, u_1 is greater than u_2 if its value is greater than that of u_2. The part of the domain between a *u-line* $= \delta_u \mathtt{u}$ and *u-line* $= \delta_u(\mathtt{u}+1)$ is the *u-strip* `u`. *v-lines* and *v-strips* are analogously defined. The triangulation is computed as follows:

For each grid *cell* (`u`,`v`), $(\mathtt{u}, \mathtt{v}) \in [1, n_u - 2] \times [1, n_v - 2]$, generate two triangles by adding a diagonal (chosen arbitrarily):

1. $[\ (\mathtt{u}\delta_u, \mathtt{v}\delta_v),\ ((\mathtt{u}+1)\delta_u, \mathtt{v}\delta_v),\ (\mathtt{u}\delta_u, (\mathtt{v}+1)\delta_v)\ ]$ and

2. $[\ (\mathtt{u}\delta_u, (\mathtt{v}+1)\delta_v),\ ((\mathtt{u}+1)\delta_u, \mathtt{v}\delta_v),\ ((\mathtt{u}+1)\delta_u, (\mathtt{v}+1)\delta_v)\ ]$,

where $\delta_u = \frac{1}{n_u}$ and $\delta_v = \frac{1}{n_v}$. In addition, at each boundary, triangulate the samples on the boundary and the internal grid as shown in Figure 4.4.

5.4 Trimmed patches

The problem of domain triangulation increases in complexity once trimming curves are associated with patches. We use the general principle of filling at the trimming curves to maintain crack-free tessellations. We again perform the triangulation of grid cells except, unlike the domain boundaries, the trimming curves are, in general, not parallel to lines on the original $n_u \times n_v$ grid.

Before the triangulation, each trimming curve is tessellated as described in Chapter IV. We classify the grid cells as *0-cells* if they are fully trimmed out, *1-cells* if they are fully retained, and *0-1-cells* if they are intersected by a trimming curve. Note that we do not compute intersections of cells with the analytic representation of the trimming curve, but only with its tessellation. The *1-cells* are triangulated, as in the previous section, by adding a diagonal. The *0-cells* are discarded. Most of the extra computation involves the *0-1-cells*. The retained portion of these cells are, in general, bounded by PSLGs (See Section 2.6). The domain triangulation algorithm for trimmed patches is described next.

5.4.1 Triangulation algorithm

Input:

- Grid sizes n_u and n_v
- The piecewise linear tessellation c_i for each closed trimming curve c_i (Curve c_i has n_i points. We refer to the j^{th} point on the i^{th} curve as $(c_i(j)^u, c_i(j)^v)$.)

Output:

- A valid triangulation of the trimmed domain

Algorithm:

1. **For each** trimming curve c_i:
    - **For each** segment $s = (c_i(j), c_i(j+1)),\ j \in [0, n_i]$

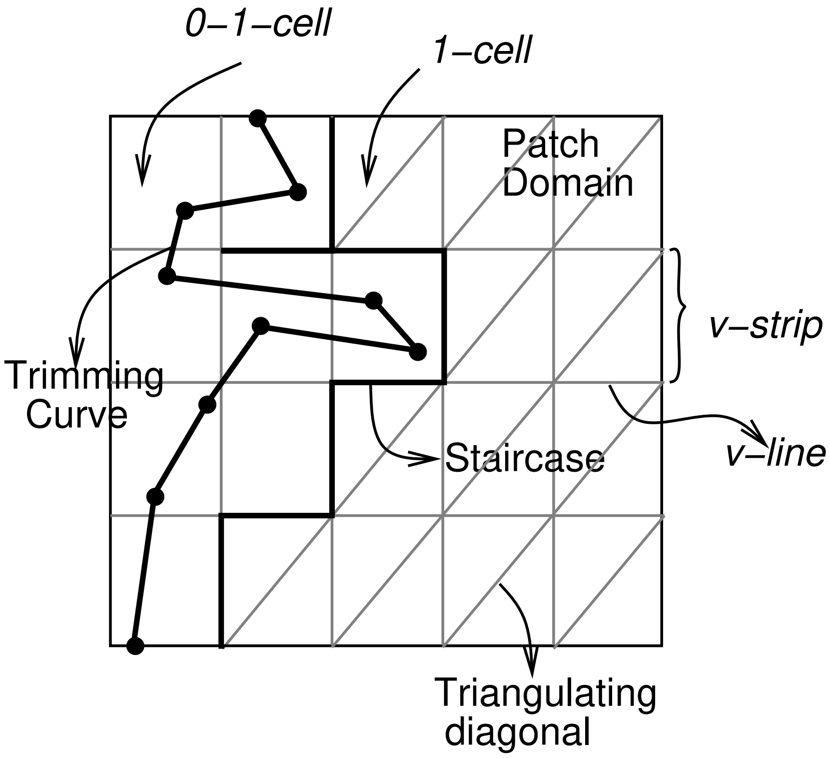

Figure 5.1: Staircase

(a) Mark the cells intersected by s as *0-1-cells*.

(b) While s is contained in the current *u-strip*, maintain the maximum and minimum *v-lines* intersected in the *u-strip*. These *v-line* values are used later to determine the extents of the *staircase* within the current *u-strip*.

(c) If s crosses a *u-line*, store the intersection and the maximum and minimum *v-lines* of the previous *u-strip*.

(d) If $c_i(j+1)^u > c_i(j)^u$, an intersection with a *u-line* is of type MAX, as s trims out the domain immediately above it (see Figure 5.2). Similarly, if $c_i(j+1)^u < c_i(j)^u$, the intersection is of type MIN. If $c_i(j+1)^u = c_i(j)^u$, the type is same as that of the previous segment.

2. For each *u-line*, sort all stored intersections. If consecutive MIN intersections exist in the sorted order, delete all but the maximum of those. Similarly, if consecutive MAX intersections exist in the sorted order, delete all but the

minimum.

3. This results in a sequence of $MIN - MAX$ pairs of *v-strip* values (which were earlier stored with intersections) for each *u-strip*. Each pair corresponds to a range of *1-cells* delimited by *0-1-cells*.

4. Generate triangles for *1-cells* using a diagonal.

5. The delimiting *0-1-cells* form a set of staircase-like polygonal chains with a non-zero offset from the trimming curves as shown in Figure 5.1. This is a PSLG.

6. We trapezoidize the PSLG using [Sei91] and generate monotone polygons.

7. Finally we triangulate the monotone polygons using greedy triangulation [FM84].

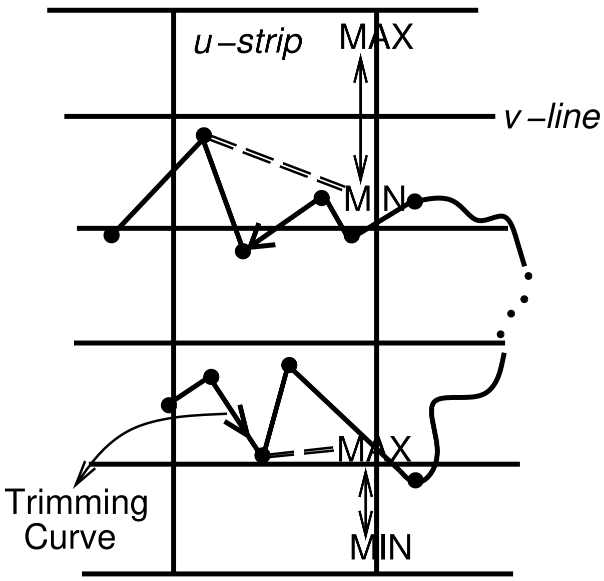

Figure 5.2: Generation of PSLG

The problem with the algorithm described above is that it generates triangles with edges greater than δ even if the adjacent samples points are less than $\frac{\delta}{2}$ apart. This results mostly from the greedy triangulation which is prone to generating long triangles. The following section describes techniques to ensure that the triangle edges are no larger than δ.

5.4.2 Triangle edges

We use two inexpensive heuristics to reduce the number of triangle edges that intersect more than two grid cells. The first heuristic adds barriers in form of line segments inside *0-1-cells* which no triangle edges may cross. The second reduces the degree of a vertex in the triangulation of a monotone polygon.

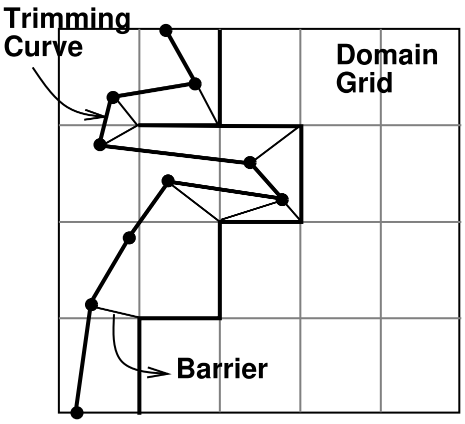

Figure 5.3: Barriers added to the PSLG

1. For each *0-1-cell*, we add an edge to the nearest point in the cell. In most cases this is a diagonal of the PSLG (see Figure 5.3). Since no edge of a triangulation of the PSLG may cross any of its diagonals, no triangle edges cross more than one cell when such a diagonal exists. This, in turn, implies that triangle edges are no larger than δ.

2. We set a limit on the number of triangles incident on a vertex. While performing the greedy connection described in Section 2.6:
   - Connect no more than four vertices to the current base.
   - Once four vertices have been connected,
     - Push the current base on to a stack

    - Move up the chain picking a new base.

- Once the top is reached, recursively triangulate the vertices on the stack.
- Repeat this process only a constant number of times (at most four in our case).
- At the last iteration remove the connectivity restriction, and perform the standard greedy triangulation.

This simple heuristic restricts the maximum degree of any vertex to be four to five on average and does not produce the long and thin triangles the standard greedy algorithm does. This is useful for incremental triangulation described in the next chapter.

Clearly, these two methods do not theoretically eliminate all occurrences of long edges. However, all long edges are internal to the patch. Only trimming curves lie on the boundary and all segments on the boundary are guaranteed to be less than $\frac{\delta}{2}$. Subdividing each internal edge maintains the triangle size property without introducing any cracks.

Chapter VI

Incremental Tessellation

6.1 Overview

Our goal to render tens of thousands of Bézier patches at ten frames per second implies that we render a patch in less than 10 microseconds. While the algorithms presented in previous chapters achieve a speed-up of more than 5 on our models over the state of the art, this speed-up is not sufficient to meet our goal. We cannot generate satisfactory triangular instances of tens of thousands of Bézier patches every tenth of a second. Fortunately we do not have to generate a new triangulation every frame. Typically, the change in the screen position of the model between successive frames is small. As a result, in most frames, the change in sampling density, n_u, n_v or an n_{t_i} for a given patch is small, if not zero. We exploit this coherence by performing a minimal amount of computation to calculate the new satisfactory triangulation starting from the one used in the previous frame.

At each frame we store the vertices of the polygons generated (and the surface normals) in memory. As the new bounds are computed, we perform a few extra evaluations of points and normals on the surface, if more samples are needed. If we need fewer samples as compared to the last frame, some triangles are removed from the list. For models with 30,000 patches we typically need to store less than 200,000 vertices, requiring about 8 megabytes of memory. Thus the memory requirements are not high for current graphics systems. With more available memory, some triangles

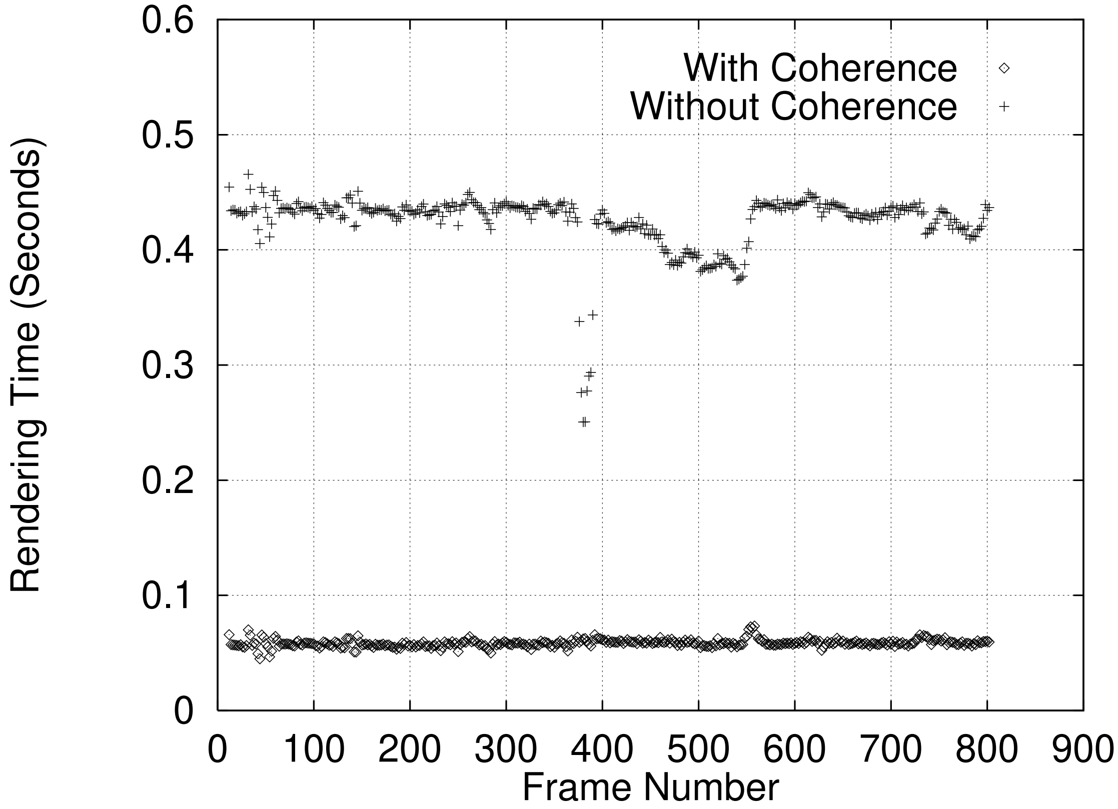

Figure 6.1: Effect of Coherence

that are not immediately needed may also be stored and re-used in case they are needed in the future.

With the use of coherence, we are able to render thirty thousand patches in about a tenth of a second on Pixel-Planes 5. The graph in Figure 6.1 shows the effect of coherence on frame rate. The graph corresponds to a short inspection of the car model (shown on the color plate) on Pixel-Planes 5. We recorded the wall-clock time when the rendering of each frame was complete. The difference in successive times has been shown in the figure, with the frame number graphed on the X axis and the time (in seconds) on the Y axes. Notice that apart from the speed-up of about 7-8, the coherence-based rendering has fewer glitches and allows a more consistent frame rate. The glitch in the rendering near frame 560 occurred when most of the model went off-screen and had to be re-evaluated once it came back as the cached triangles had been discarded. The glitch around frame 380 in the non-coherent rendering occurred when a large part of the model was concentrated on a small part of the screen. On average, the coherence-based scheme requires the re-tessellation of only 15-20% of the model every frame.

6.2 Sampling

In this section we discuss the updates of the sampling size, and show how to relax the stringent quality requirements (see Section 4.2) achieving significant speed-up with little loss of image quality. We group Bézier patches into clusters, and determine the sampling density for full clusters first.

6.2.1 Cluster sampling

A CAD model typically is organized as a number of solids each consisting of a number of Bézier patches. Since displaying CAD models is our driving application, a clustering of patches is naturally available in terms of one cluster per solid. We maintain a tightly enclosing bounding box, B_i, around each cluster, C_i. In the first frame of the simulation we perform sample size computation for each patch as described in Chapter IV. Moreover, we compute and store the area $\mathcal{A}_s(B_i)$ of the screen projection for each bounding box B_i. In subsequent frames:

1. Recompute the screen area of the bounding box, call it $\overline{\mathcal{A}_s(B_i)}$.

2. If the difference $|\overline{\mathcal{A}_s(B_i)} - \mathcal{A}_s(B_i)| > K\mathcal{A}_s(B_i)$, compute the required sampling for each patch in cluster C_i and set $\mathcal{A}_s(B_i) = \overline{\mathcal{A}_s(B_i)}$. K is a user-defined constant.

3. Otherwise, re-use the previous triangulation for all patches in C_i.

We have found $K = 0.1$ to work well in practice, since our bounds typically result in triangles about 10-15% larger than the target size (Section 4.2). Clearly, there is no guarantee that this will indeed be the case for all patches. Consequently we may violate the quality constraints sometimes. Fortunately, for display purposes, we can tolerate such errors. Occasional drawing of triangles marginally larger than the required size goes unnoticed. Of course, when the change in $\mathcal{A}_s$ is large, we must re-evaluate the sampling requirements for each patch. We make some optimizations in that computation also, which we discuss in the next section.

6.2.2 Coherent patch-sampling

In this section we demonstrate how to manage samples along a curve, particularly along the u dimension of the domain. The same algorithm is repeated along the v dimension of the domain and along the parameter of each trimming curve. We still use the algorithm in Chapter IV to compute the required number of steps n_u, but the samples are no longer placed uniformly along the u parameter. We intersperse new samples among the old ones. There are three important implications:

1. Only a few samples, rather than all, need to be re-evaluated or discarded.
2. Due to curve-based sampling management, we still introduce complete *u-lines* or *v-lines* rather than independent sample points.
3. The coherent sampling is not uniform any more. It is a hybrid of adaptive and uniform tessellation.

Given the sample size for the last frame, $\overline{n}_u$ and that for the current frame, n_u, we want to update the triangulation. Let $\|\overline{n}_u - n_u\| = \epsilon_u$.

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

> Of the candidate intervals we seek to subdivide, we first choose the one across which the change in the magnitude of the derivative vector, i.e.

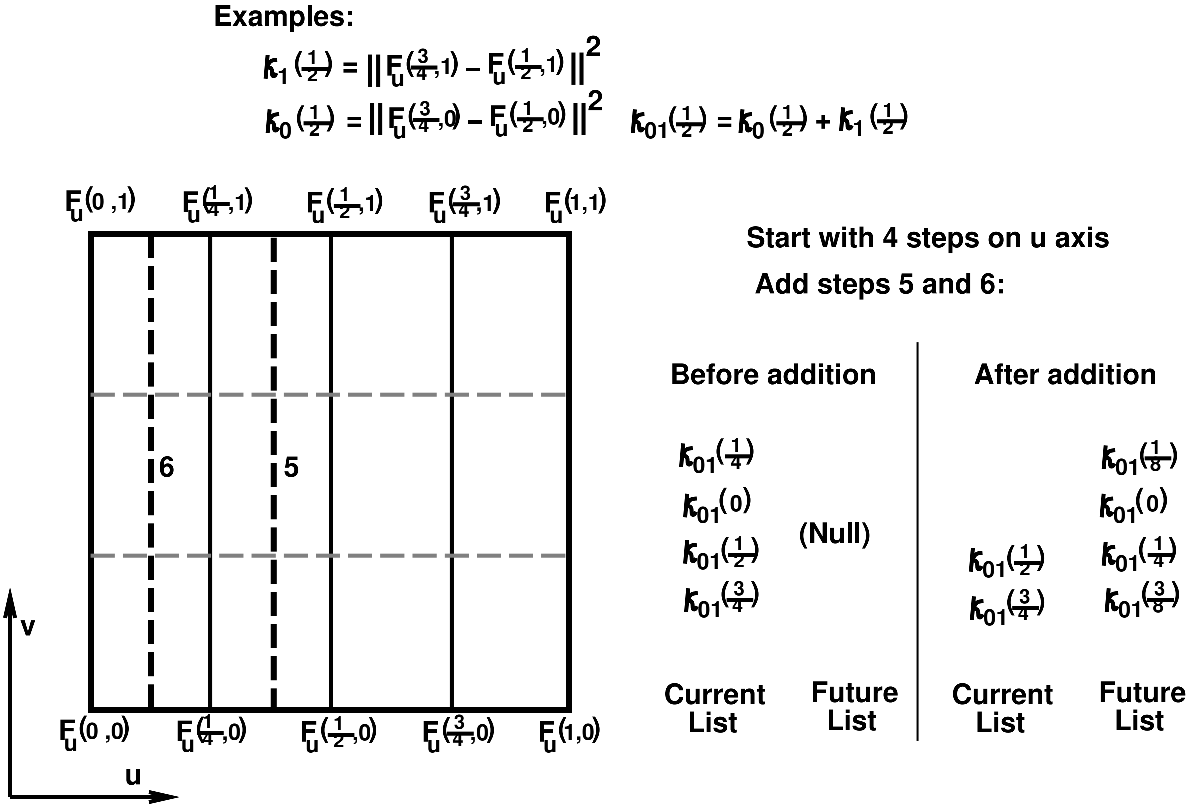

Figure 6.2: Example of incremental tessellation

average local curvature, is the maximum. The next sample is placed in the interval of the next largest change, and so on.

Formally,

Let $\hat{n_u} = 2^{\lceil \lg n_u \rceil}$ be the next power of 2 for n_u. Thus there are at most $\frac{\hat{n_u}}{2}$ intervals to choose from, and of the $\frac{\hat{n_u}}{2}$ sample points lying in the range $\left[\frac{\hat{n_u}}{2}+1, \hat{n_u}\right]$, the i^{th} point is added at $u = \frac{1}{\hat{n_u}} + \frac{2i}{\hat{n_u}}$. The chosen i is based on the derivative vectors:

For samples at u_1, u_2 on the isoparametric line $v = v'$, let $\kappa_{v'}(u_1, u_2) = \|\mathbf{F}_u(u_2, v') - \mathbf{F}_u(u_1, v')\|^2$. We drop u_2 when it is understood to be the sample at smallest u greater than u_1, i.e. the one next to u_1 (see example in Figure 6.2). We store the sum, κ_{01}, of $\kappa_0(u_1)$ and $\kappa_1(u_1)$ for each evaluated sample point in [0,1). Of these, the $\frac{\hat{n_u}}{2}$ candidate intervals are maintained sorted. We add the next sample at the i corresponding to the maximum element of the list.

We maintain two sorted lists of κ_{01}'s — a current list and a future list. The newly subdivided intervals do not again become candidates for subdivision until the required sampling size becomes greater than $\hat{n_u}$. The two lists are merged and a new list is created when $n_u = \hat{n_u}$, i.e. it becomes a power of two. The algorithm, in short, is:

1. If $CurrentList$ is empty
   - Set $CurrentList := FutureList$
   - Create empty $FutureList$
2. Subdivide the interval corresponding to head($CurrentList$). Delete the head.
3. Compute the derivative at the new sample point. Compute new κ's.
4. Insert the left and right half intervals into $FutureList$ (in their sorted positions).

In case $n_u < \overline{n}_u$, we need to discard some samples. This process is just the inverse of adding samples: the last sample point added gets deleted first.

Notice that even though we compromise the uniformity of tessellation, we always decompose the domain into rectangles whose edges are parallel to the u and v axis. Thus when we introduce an additional sample line along the u or v dimensions, all the points on the surface are evaluated using the generalized Horner's rule, which still takes linear time. This technique works well in practice because it includes some benefits of adaptive tessellation, increasing tessellation where the curvature is high. At the same time, the overall efficiency of uniform tessellation is not compromised.

6.3 Incremental triangulation

Once we have added (or deleted) the new (old) sample lines on the grid or points on the trimming curves, we must update the domain triangulation. It is easy to see that the class of inputs to our new triangulation is limited. All inputs can be classified into addition or deletion of features:

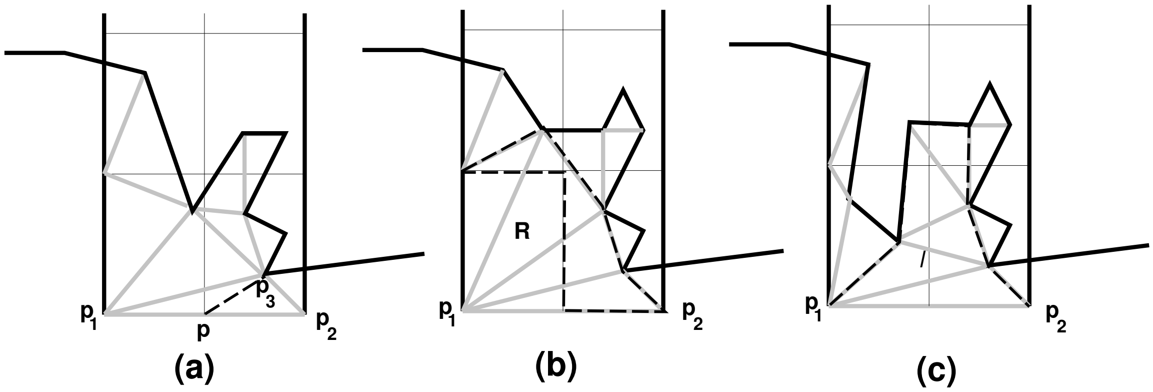

Figure 6.3: Grid update

1. Add a grid line.

2. Delete a grid line.

3. Split a curve segment into two.

4. Merge two consecutive curve segments.

This allows us to specialize the incremental triangulation to handle these cases. The algorithm can be explained in terms of two basic cases:

1. Addition and deletion of grid-lines.

2. Addition and deletion of curve segments.

In either case, at worst, we need to triangulate a PSLG with only a few points. The cases are explained in Figures 6.3 and 6.4. The old triangulations are shown in light color and the new ones are marked with dashed lines. We refer to the grid-lines, curve segments and sample points as the triangulation features.

6.3.1 Re-triangulation Algorithm

The basic idea of the re-triangulation algorithm is:

- To add a new feature, delete all features intersected by the new feature, and re-triangulate the hole along with the new feature.

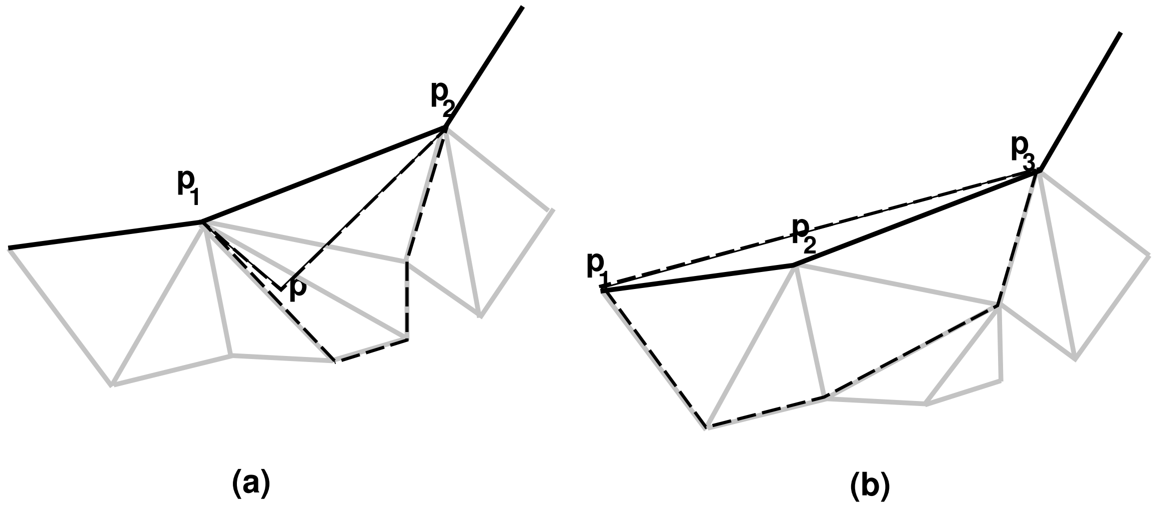

Figure 6.4: Update of trimming curve samples

- To delete a feature, delete all features intersecting the deleted feature, and re-triangulate the hole (*sans* the deleted feature).

If a grid line is added between two grid lines, we potentially need to re-triangulate all regions where it intersects the staircase. Additionally it may intersect complete cells of G_0 which are trivially subdivided. For each staircase intersection we need to insert a point, a sequence of points, or a sequence of rectangles into the current triangulation.

- Case: Point p needs to be inserted (Figure 6.3(a)).

    1. p lies on an edge p_1p_2 of triangle T. Connect p to the apex, p_3, of T.
    2. Connect p to the points p_i adjacent to p_1 that are also visible from p. Replace each such triangle $p_1p'p_i$ by triangle $pp'p_i$.
    3. Repeat vertex substitution for p_2. (Partitioning triangles in this fashion allows us to balance the degrees of p_1, p_2, and p.)

- Case: A sequence of points, i.e. a line, l, needs to be inserted (Figure 6.3(b)).

    1. Delete all triangles intersecting l.
    2. Partition the vertices of deleted triangles into two sets, one on each side of the line. Each side forms a polygonal chain (shown in dotted line).

3. Triangulate each side with the points on l.

- Case: A sequence of rectangles need to be added (Figure 6.3(c)).

  The operations are similar to the case of adding a line. Instead of triangulating each side with the line, we triangulate the left side with the left side of the sequence of rectangles, and the right side with the right.

Processing of grid-line deletions is just the inverse of addition process. In short, all triangles incident on the points of the deleted line are deleted. This results in two polygonal chains on either side of the deleted line. The two chains are triangulated together. For the case of segment update, the cases are similar:

- Case: A segment p_1p_2 is split into p_1pp_2 (Figure 6.4(a)).

  1. Delete all triangles intersected by p_1p and pp_2.
  2. The vertices of the hole form a PSLG. Re-triangulate the PSLG.

- Case: Segments p_1p_2 and p_2p_3 are merged.

  1. Delete all triangles adjacent to the point p_2.
  2. The vertices of the hole form a PSLG. Re-triangulate the PSLG.

Note that the update of segments may cause a change in the grid. As described, the algorithm handles each case separately. This is only for ease of understanding; in practice all updates can be combined.

6.3.2 Data structure

The three basic operations of the incremental algorithm are

1. Locate triangles intersected by a feature.
2. Form polygonal chains.

3. Triangulate a PSLG.

4. Update triangulation.

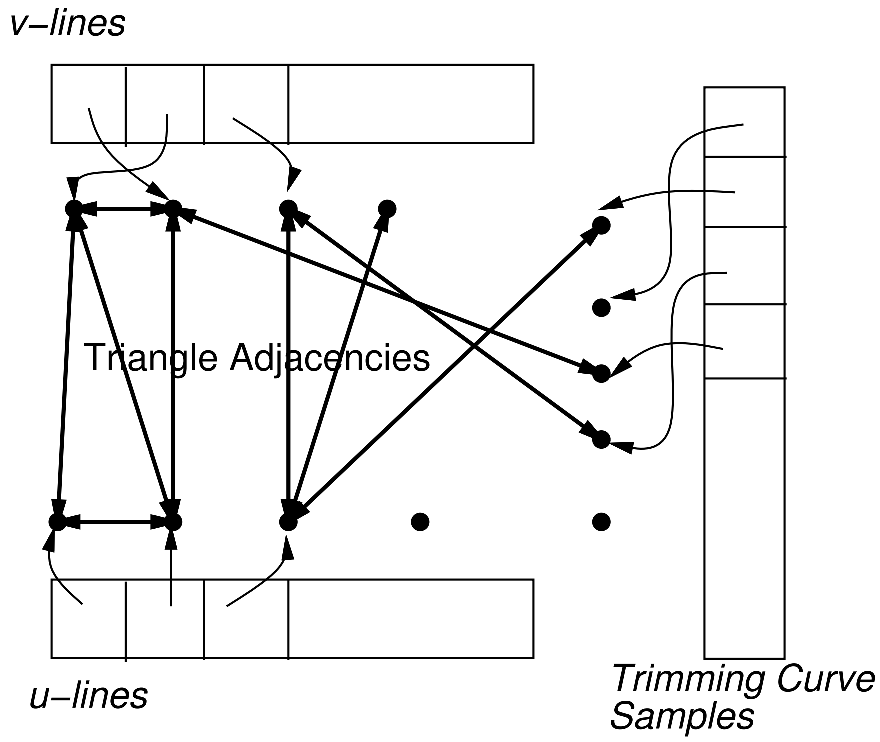

Figure 6.5: Data structure for incremental triangulation

To support these operations, we use an adjacency-based data structure (Figure 6.5) that can be quickly queried to compute the set of points in the triangulation that are adjacent to a feature. The data structure maintains the following information:

1. Set of vertices, sorted by the G_0 grid-cell they lie in.

2. For each vertex, the list of adjacent vertices (in the triangulation), sorted by angle.

3. Visit-flag marker for each triangle (which we use to prevent the generation of duplicate triangles when traversing the data structure).

The common operations are performed as follows:

- **Locate triangles:** The triangles intersected by a grid-line g are adjacent to points that lie on grid-lines g_1 and g_2 lying on either side of g, and the points lying inside the G_0-cells bounded by lines g_1 and g_2. The triangles intersected by a segment are those incident on the vertices adjacent to the segment.

- **Form chains:** Chains are formed due to deletion of triangle edges or vertices lying on a grid-line, or a segment. We traverse the vertices on the deleted feature, forming a list of triangles adjacent to each vertex. For each vertex, this list is sorted by the angular order about it.

- **Update adjacencies:** This is a simple data-structure problem. We just need to insert elements at the right position in a sorted list.

Chapter VII

Analysis of Surface Triangulation

7.1 Overview

This chapter analyzes our surface tessellation algorithm and explains the choices it makes. Any polygonization-based surface rendering algorithm has two main stages:

- Polygon generation
- Polygon rendering

Thus such algorithms first need to allocate resources to generate these polygons. A satisfactory tessellation must be computed, and vertices and normals evaluated. The performance of polygon rendering is system-dependent and is typically a function of the number of polygons and the sizes and distribution of these polygons on screen.

It is possible to compute a very fine polygonization a priori and render all polygons at each frame. In this case, almost no time is spent in polygon generation (some time is spent in loading polygons into memory) and most of the time is spent in rendering. However, the number of polygons needed for close-up (zoomed) views of some surface patches can be extremely high (more than a thousand) and to interactively render models consisting of tens of thousands of patches, we require hundreds of megabytes of storage, and the capability to render hundreds of millions of polygons per second. We can reduce the demand on the polygon rendering pipeline by computing different *levels of detail* of each patch and, at each frame, choosing one of these approximations as a function of the viewing parameters. In general, these methods suffer from geometric robustness problems, rendering artifacts and high memory requirements.

On the other hand, we can compute, on-line, the minimum number of polygons required for a smooth image as a function of the viewing parameters (for each frame). The resulting algorithm is based on adaptive sampling and takes considerable time in the polygon generation phase for each frame. As a result, it can be too slow for interactive performance on large-scale models.

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

- *Minimize total time:* If the two stages are performed sequentially on the same processor, we only need to minimize the sum of times spent in each stage. This is typically the case when rendering without specialized hardware.

- *Balance individual time:* If the two stages are pipelined, we need to balance the time spent in generating polygons with that in rendering them, while minimizing the time spent in each stage. This case, being the norm for interactive graphics, is the one of interest to us.

In this chapter we explain our choice of uniform sampling and compare our tessellation algorithm to some earlier techniques. By resorting to exact computation of bounds we are able to generate better sampling. At the same time, by performing our expensive operations off-line and devising techniques to efficiently map pre-computed bounds to screen space at rendering time we obtain a more efficient algorithm.

7.2 Sampling size determination

Our goal is to find the correct balance between triangle generation time and triangle rendering time. There are two main parts of triangle generation:

1. Determine the samples needed for a smooth image.

2. Evaluate the samples and compute the triangles approximating the surface

In order to determine the sampling size we either exactly compute or estimate surface properties like derivatives, curvature and screen-space area. We defined certain quality criteria in Chapter IV, but in general, it is not straightforward to measure quality, which is a function of many subjective factors. Even if we assume our quality constraints, it is NP-hard to compute the optimal triangulation or to even determine the minimum number of triangles required to satisfy the constraints. Thus, from a practical point of view, over-sampling is unavoidable, once the quality constraints have been established. On the other hand, a correct evaluation of bounds results in fewer triangles than an over-estimation. At the same time an estimation typically is less computationally expensive than the exact bound computation. Thus there exists a trade-off between the time spent to compute the bounds and the number of samples generated. There are two such trade-offs:

- Certain bound computations are performed to estimate the change in on-screen surface properties between frames. For example, suppose we use the change in the screen area of a patch to determine if re-tessellation may be needed. Suppose we use algorithm $\mathcal{A}_1$ to estimate the area. Even if the estimate itself is loose, the decision to re-tessellate is likely to be correct if the error in $\mathcal{A}_1$ does not vary significantly between frames.

- If absolute bounds are used, the constraints are more stringent on their computation. We must achieve enough accuracy so that the time to generate the triangles and the time to render them well match the time to compute the bounds.

Bound computation is the most expensive operation in our implementation and is done every frame even if the actual tessellation changes only about 15% of the time. However, as shown in Section 7.4 our method produces fewer triangles than earlier techniques while spending less time in bound computation.

7.3 Adaptive sampling

A number of different algorithms for adaptive subdivision have been proposed recently. As opposed to uniform sampling methods, that place samples only on a grid, adaptive sampling methods may place the samples almost anywhere on the domain. Forsey et al. [FK90] present a simple recursive subdivision based scheme that tests the size and flatness of each sub-patch and approximates it with a rectangle once the rectangle is closer to it than a user-specified tolerance. Sheng et al. [SH93] present an adaptive tessellation algorithm for trimmed patches. This algorithm starts with a Delaunay triangulation of points on the patch. It then subdivides each edge longer than a user-specified value in object space. Klein et al. [KS95] similarly use Delaunay triangulation. Their algorithm incrementally computes the farthest (in terms of Euclidean distance) patch point from the current approximation and inserts that point into the triangulation. The process terminates when this farthest point is closer than some user-specified tolerance. Vigo et al. [VB93] also start with a triangular surface approximation. Their algorithm recursively subdivides each triangle with an edge longer than a user-specified tolerance. Vlassopoulos [Vla90] proposes an algorithm that operates in terms of line segments in the domain. This algorithm starts with a uniform grid and adds horizontal or vertical segments in the parts of the domain where the approximation has a high deviation from the surface or normal deviation. Bajaj [Baj90] presents an adaptive sampling scheme based on surface curvature. This scheme is designed to polygonize surfaces in three dimension or higher. Bajaj recommends polygonizing implicit surfaces by parametrizing them first. Bajaj et al. [BR94] simplify the sampling process by first identifying all degenerate points and curves on the surface and subdividing surfaces along these features.

Although adaptive subdivision schemes produce fewer triangles, none of the algorithms listed above are able to render more than a few hundred Bézier patches in real-time. The main reason is that they generally require too much time to perform the tessellation: more than a minute for some patches [Vla90]. We performed an empirical test to study the requirements on a real-time adaptive tessellation algorithm.

We stored a number of sequences of view points used in a user-run of our display system. For each view point of each sequence, we computed, off-line, an adaptive tessellation of each patch using the following algorithm:

1. Transform the control points of the patch.

2. Approximate the patch by a rectangle defined by its four corner points.

3.
   - If the maximum deviation of the patch from the approximating rectangle is more than 10 pixels, say at the parametric point (u_1, v_1), subdivide the patch into four sub-patches at isoparametric lines $u = u_1$ and $v = v_1$ and recursively tessellate each sub-patch.
   - If the diagonal of the approximating rectangle is longer than 50 pixels, subdivide the patch into four sub-patches at isoparametric lines $u = 0.5$ and $v = 0.5$ and recursively tessellate each sub-patch.

4. Store the (u, v) values of the tessellation of the patch, when all its sub-patches have been satisfactorily tessellated.

Thus, on-line, only a few triangles are generated and no tests of "satisfactoriness" of tessellation need to be performed, but the points on the surfaces and their normals are evaluated. We recorded the time spent in triangle generation. The total number of processors used in an experiment was determined by the amount of memory needed for the model. The evaluation time (including the transformation time) was compared with the overall time, including the rasterization. As Table 7.1 shows, the evaluation stage was already the bottleneck in some cases. This means that any more computation that is proportional to the number of triangles generated would considerably slow down the system. It also implies that any time spent in computation of uniform tessellation must be insignificant. In fact, as shown in Section 6.1, with the exploitation of frame-to-frame coherence we are able to achieve just that. While the flatness of patches seems to vary little between frames, it is hard to find a test that exploits this fact. Since these tests are the most expensive operation of adaptive tessellation, we are not able to achieve similar speed-ups for adaptive tessellation.

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

Table 7.1: Complexity of surface evaluation

On Pixel-Planes 5 (one renderer) the processor generating the triangles is mostly busy.

The following are some of the reasons that uniform tessellation algorithms generally out-perform the adaptive tessellation algorithms:

- Simplicity of the algorithm. Uniform tessellation involves a precomputation of bounds and evaluation only.

- Simple evaluation algorithm based on modified Horner's rule of average complexity $O(n)$ as opposed to $O(n^2)$ as shown in Chapter II.

- No good and simple algorithms for quick determination of the flatness of parts of a patch.

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

- Flatness of real-life surfaces. In practice, most large-scale models consist of many relatively flat patches. This is indeed the case after converting NURBS models into Bézier patches. Adaptive subdivision does well on patches with highly varying curvatures. On such models the uniform tessellation may over-sample the patch.

7.4 Uniform sampling

Uniform subdivision methods generally outperform [FMM86, KS95] adaptive methods. Most uniform subdivision algorithms first estimate the step size to use for tessellation and use results from approximation theory [Pre75]. In the context of computer graphics, Lane et al. [LR81] present methods to estimate the maxima, minima and roots of a Bézier polynomial. Filip et al. [FMM86] apply and extend these results to compute the step size for uniform tessellation. In particular, they derive the deviation bounds reported in Chapter IV.

Rockwood et al. [RHD89] report one of the first successful attempts at real-time rendering of trimmed NURBS surfaces, and we draw profusely from their method. [RHD89] decomposes NURBS surfaces into Bézier patches and uniformly tessellates each Bézier patch. It also uniformly tessellates each trimming curve. The sampling criterion is based on the triangle edge size. In order to simplify the triangulation, [RHD89] decomposes each trimming curve into monotonic segments and introduces degenerate triangles between sample points on the trimming curve and its intersection with the uniform surface grid.

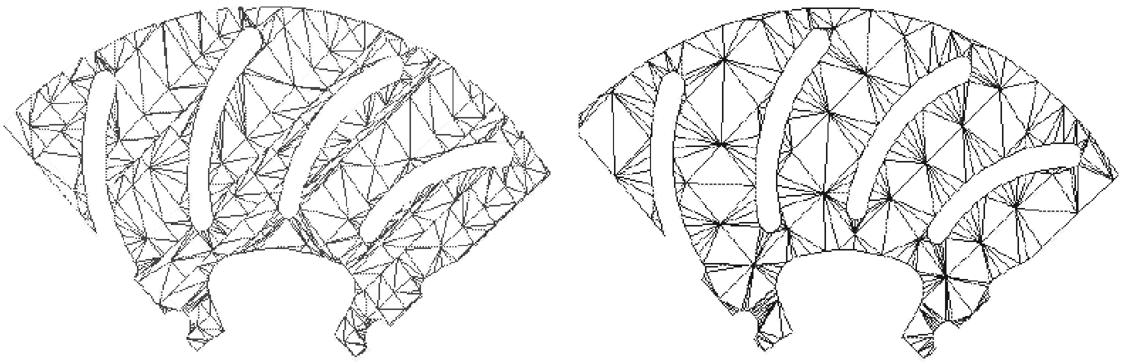

Figure 7.1: Comparison of [RHD89] tessellation to ours (on the right)

We adapt [RHD89] boundary triangulation algorithm for crack prevention. In particular, we make sure that the sample points on a boundary curves coincide on both its adjacent patches. We, however, avoid degenerate triangles and do not introduce the intersection points; such extra points may produce cracks in the tessellation.

We also do not need to break up the trimming curves into monotonic segments by adopting a more complex triangulation algorithm. As a result, we obtain a reduction in the number of triangles produced and an improvement in the uniformity of these triangles. A comparison of the two triangulations on a brake-rotor is shown in Figure 7.1. Since the [RHD89] algorithm performs the computation of sampling size entirely in screen space, it is computationally more expensive. Moreover, in order to improve its efficiency, it uses simple algorithms to estimate its bounds (Equation IV.2), which can be rather loose for rational patches.

We use the idea of Abi-Ezzi et al. [AES91] and decompose the bound computation into an off-line and an on-line algorithm. They evaluate their bounds in object space and map them to screen space. Our screen-space mapping is slightly less complicated, and hence more efficient, but is less general. We also improve on the tightness of their bounds by spending more time in the off-line component and computing the exact maxima of derivatives using techniques from elimination theory (as discussed in Chapter IV). [AES91] uses the standard mean value theorem for its bound computation, which is not applicable to many surfaces. We modify the bound computation (see Chapter IV) so that it is applicable to vector-valued functions, which surface functions are.

We empirically compared our bounds with those of Rockwood and those of Abi-Ezzi. For each model, these comparisons were performed over a large number of user-driven model inspections run with different user-specified screen-tolerance values. We collated the averages of the number of triangles generated for each model. The degrees of the patches in these models were between two and three in u as well as v. Some models (e.g. Dragon) contained no rational patches to make Rockwood's bounds unduly loose. For a given tolerance, our bounds result in about 31% fewer triangles than Rockwood's and about 20% fewer than Abi-Ezzi's, without sacrificing visual fidelity. Table 7.2 shows the relative number of triangles produced for a few models. Color plate A displays the wire-frames and shaded images of the pencil and goblet models computed using the three methods. Since we use tighter bounds, the variation in sizes of our triangle is less than that produced by [RHD89] or [AES91] algorithms.

Model	Number of Patches	Our Algorithm		[RHD89] Algorithm		[AES91] Algorithm	
		# Tris.	Ratio	# Tris.	Ratio	# Tris.	Ratio
Goblet	72	535	1	790	1.48	659	1.23
Pencil	570	4720	1	6875	1.45	5810	1.23
Dragon	5354	19220	1	22500	1.15	22755	1.18

Table 7.2: Comparison of the number of triangles generated for a given tolerance

7.5 Direct NURBS tessellation

Luken et al. [Luk93, LC93] present an algorithm to polygonize trimmed NURBS surfaces. They generate general polygons, not triangles. They do not decompose trimmed NURBS into trimmed Bézier patches either, but tessellate NURBS directly. They divide each u-span and each v-span into uniform steps. As a result they normally generate more polygons than needed. Moreover the evaluation of points and normals on a NURBS surface requires more operations than Bézier evaluation. The Bézier form incurs only a one-time cost of conversion from NURBS to Bézier. These Luken algorithms assume that a NURBS surface forms a complete solid, and hence make no attempt to prevent cracks between NURBS surfaces. Piegl et al. [PR95] also tessellate NURBS directly. They use a surface deviation tolerance to guide their step size. Their algorithm guarantees bounds on the deviation in object space, as opposed to screen space.

Chapter VIII

Parallel Processing

8.1 Overview

Part of the rendering speed-up on modern graphics systems is achieved by parallelism. Indeed, our target architectures, namely SGI machines and Pixel-Planes 5, both incorporate multiple CPUs. Pixel-Planes 5 GPs (Graphics Processors) are connected via a communication network (Figure 1.11), whereas the CPUs on SGI communicate via the memory subsystem which is shared by all CPUs (Figure 1.10). We have achieved real-time update of our triangular approximations only by effectively utilizing parallelism for tessellation. For example, assuming re-triangulation of approximately 20% of the model every frame, to render a model with 30,000 Bézier patches, we must compute the sampling density for 30,000 patches and update the triangulation of 6,000 patches in less than a tenth of a second. This requires hundreds of millions of operations per second, well beyond the capability of a single i860 GP. This chapter reports our parallelization technique.

For any parallel implementation it is important to determine

1. What part of the computations is performed at each processor?

2. When and how do processors exchange information?

Since most high-end graphics systems support coarse-grain parallelism, that is primarily the model our algorithms are designed for. Decomposing the problem is quite straightforward. Each patch, or cluster of patches, forms an independent unit of work (called *work-unit*) that we map to an individual processor. Such mapping eliminates any need for processor communication. Each processor is independently able to generate the triangular approximation of patches assigned to it. While the duplication of common evaluation of vertices at the boundary of two patches can be avoided by allocating a full solid to one processor, our dual trimming curve representation ensures that doing so is not necessary. Even if adjacent patches are mapped to different processors, they can be tessellated independently without any introduction of cracks, as discussed in Section 4.4. Once all triangular approximations are generated, the triangles are collated and passed down to the rasterization unit.

However, a naive decomposition of the problem leads to severe load imbalances. There is no way to pre-determine the cost of tessellating each work-unit, as it can vary significantly during a simulation. When the user zooms in to a small part of the model, that part occupies a significant part of the screen, and hence a significant part of the total tessellation cost. If that part of the cost is not fairly distributed amongst all processors, some processors become bottlenecks. If we randomly distribute work-units to processors, say in a round-robin manner, load imbalances of more than 1:50 are not uncommon. Since the load varies dynamically, any guarantee of bounded load imbalances can only be achieved by re-allocating work-units. In our case, this re-allocation means transfer of the patches and trimming curves, the pre-computed bounds and the cached triangles. In systems that do not have enough available communication bandwidth, this data movement itself can induce significant overhead. For such systems, we present a heuristic for one time work allocation that reduces load-imbalance on average. On the other hand, for systems that do allow efficient data movement, for example the shared-memory system of SGIs, we present a dynamic load balancing algorithm that incurs little overhead. Two important features of this algorithm are:

1. It uses no blocking locks and thus does not spend time waiting to get into critical regions.

2. It induces no overhead on any processor busy tessellating surface patches.

By resorting to dynamic load balancing we are able to achieve a $10-15\%$ speed-up over static load balancing, and by eliminating locks we achieve an additional 15% speed-up. Section 8.5 reports comparative statistics of our different load balancing algorithms. This chapter also presents a speed-up trick, called *greedy rendering*, that allows us to display out-of-date triangles in order to maintain interactive frame rates.

8.2 Static load balancing

A number of load balancing algorithms reduce the problem to graph partitioning [HL95, KK95]. These algorithms assume the existence of a load graph. The vertices and edges of a load graph have weights associated with them. The goal is to find sub-graphs of the load graph such that each sub-graph satisfies certain properties. We represent each surface patch by a vertex of the load graph. The vertices and edges of this graph are assigned weights such that weight of a sub-graph estimates the rendering cost for patches corresponding to its vertices. In order to construct the load graph, we first estimate the processing cost of each surface patch. The rendering cost of a spline primitive is a function of parameters such as:

- the degree of surface
- the complexity of the trimming curves
- the screen size of the primitive

Although, the dependence of this cost on the viewing parameters makes it difficult to find an assignment which is optimal for all frames, primitives on the same part of the screen tend to have a similar load. This observation of spatial coherence suggests the distribution of proximate primitives to different processors. We factor this information into our edge weights. We construct a complete graph with n vertices, one for each patch. The weight assignment is as follows:

- **Vertex weight**: The estimated rendering cost of a surface patch. It is a function of surface degrees, m and n, the degrees of the trimming curves, n'_{t_i}, and the size of the object space bounding volume of the patch, S_B:

$$W(v) = S_B \times m \times n \times \sum_i n'_{t_i}$$

- **Edge weight**: The inverse of the geometric distance between the patches corresponding to the two vertices it connects. We use the minimum distance between the two corresponding bounding volumes S_{B_1} and S_{B_2}:

$$W(v_1, v_2) = \min |p_1 p_2|, \text{ where } p_1 \in S_{B_1} \text{ and } p_2 \in S_{B_2}$$

- **Subgraph cost function**: For each edge, $e = (v_1, v_2)$, we calculate $\mathcal{C}(e) = W(e)[W(v_1) + W_(v_2)]$, where W is the weight function. The cost of a sub-graph $\mathcal{G}$ is $\mathcal{C}(\mathcal{G}) = \sum_{e \in \mathcal{G}} \mathcal{C}(e)$

Our goal is to partition the graph into p disjoint subgraphs of almost equal weight, where p is the number of processors, such that the cost of each subgraph is minimized. We minimize the function $\sum_{i=1}^{p} \mathcal{C}(\mathcal{G}_i)$ using recursive descent. We start with a decomposition of the load graph. We improve the decomposition by moving vertices between sub-graphs. The algorithm terminates when no vertex movement further reduces the objective function. We use the following heuristic to guide the descent:

> If two nearby primitives are on the same processors, the cost function of the subgraph for that processor is high because the weight of the edge connecting the primitives is high. The minimization algorithm moves one of the primitives to a different processor with high probability.

In practice, this algorithm yields about a 15% speed-up over random work allocation and a 10% speed-up over round-robin allocation. The round-robin algorithm distributes patches in a round-robin manner and balances the number of patches allocated to each processor.

8.3 Primitive re-distribution

As mentioned earlier, in any given frame the processor load may be imbalanced. This means that some processors may finish the tessellation of their work-unit and become *idle* while others are still *busy* tessellating patches. This work-unit is dynamic in the sense that the cost to render the same set of patches changes with time. For systems with efficient inter-processor communication, patch re-distribution results in a well balanced load.

In this section we first present a simple scheme for re-distributing work-units. This algorithm maintains a global queue of work-units, arbitrating access to the queue using *locks*. We then present the *load stealing* algorithm which improves this scheme by eliminating processor waits and scales well with the number of processors.

8.3.1 Global queue (with locking)

We maintain a global *queue* of work-units that each processor accesses. Each element of the queue (Figure 8.1) corresponds to N_p patches. For each frame, each processor executes the following loop:

```
While queue is not empty {
        Lock queue
        Remove element from the front of queue
        Unlock queue
        Tessellate the patches corresponding to element
}
```

The granularity, N_p, of the queue elements affects the processor utilization. In our experience, N_p should be a small fraction of N, the total number of patches to be rendered. Since, a $1 : 100$ load imbalance can sometimes occur for static schemes, we use $N_p = \frac{N}{100P}$, where P is the number of processors.

A problem with this method is the fixed choice of granularity. Different values of N_p are required to achieve optimal speed-up for different environments and models.

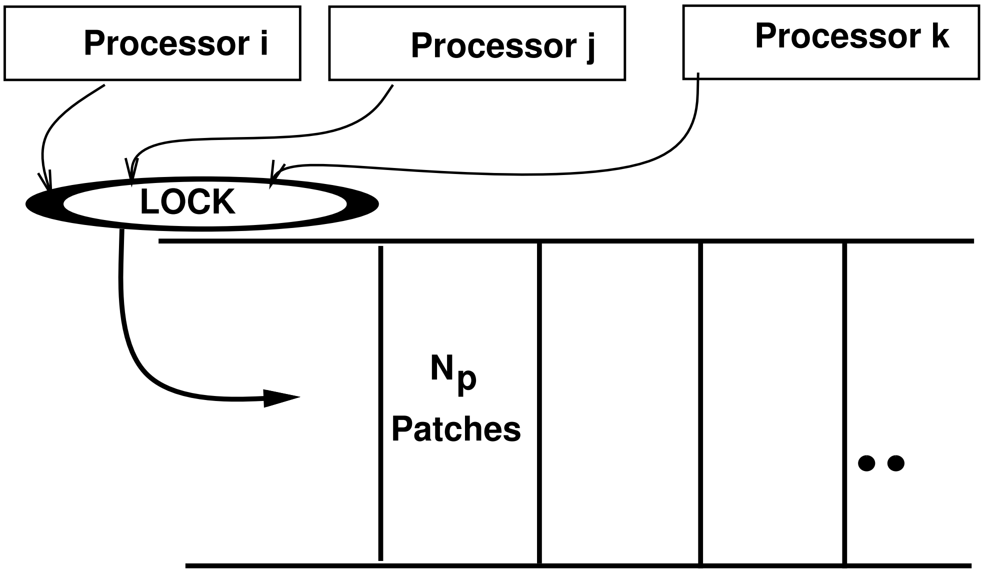

Figure 8.1: Global queue with locking: a potential bottleneck

In addition, while this algorithm performs better than the static allocation algorithm, the overhead of lock maintenance for mutual-exclusion prevents us from making full use of dynamic load-balancing. The load stealing algorithm eliminates this overhead. At each frame, each processor starts with an approximate work-load. As processors become idle, they steal patches from busy processors, computing the tessellation of those additional patches.

8.3.2 Load stealing

We no longer maintain a global work queue. Each processor p maintains its current work-load in the local queue $ActivityList[p]$. $ActivityList$ of each processor is visible to every other processor. Patches allocated to p for the current frame form the elements of $ActivityList[p]$. At the beginning of simulation, this allocation is same as the static allocation described in Section 8.2. At each frame, each processor runs its tessellation loop and updates its allocation.

Tessellation loop

Due to temporal coherence, the rendering cost of a primitive does not vary significantly between successive frames. Hence the *ActivityList* for a processor in the previous frame is a good starting guess for a balanced *ActivityList* for the current frame. The basic idea of our algorithm is to impose no additional overhead on busy processors. All the extra computation is done by the idle processors. Here is the basic tessellation loop:[1]

`While (` $Next$ `=` $Front$ `of` $ActivityList$ `is not End of list ){` :1
    `Update` $Front$`;` :2
    `tessellate (SURFACE` $Next$`)` :3
    `update triangles for` $Next$:4
`}` :5
`Find a busy processor` p_b`, Share its load.` :6

In this algorithm, possible concurrent access to shared variables occur at line 6. The first contention at line 6 is between idle processors. Multiple processors may "find" the same p_b. To reduce such contention, each processor maintains a list of other processors in a random order, and checks processors' *ActivityList*s in that order. Of course, this does not guarantee mutual exclusion. The access is arbitrated by **non-blocking locks**. Each processor has a lock, *LockList*, associated with it. Before an idle processor p_i checks the *ActivityList* of p_b ($i \neq b$), it secures a non-blocking lock on $LockList[p_b]$. Non-blocking lock ensures that if p_i cannot acquire $LockList[p_b]$, there must exist another idle processor p_j currently sharing $ActivityList[p_b]$. A processor p_i, when idle, executes the following loop until all processors become idle:

`for` p_b `in` $RandomList$ `{` :61
    `lock` $LockList[p_b]$:62
    `if lock not acquired, go on to next in` $RandomList$:63
    `if (`$Front$ `of` $ActivityList[p_b]$ `is not End of list)` :64

[1] To simplify the notation, we don't index variables by the processor id, when it is clear from context.

Share load with processor p_b :65

unlock $LockList[p_b]$:66

Perform Tessellation loop :67

} :68

Lock-free implementation

Once the idle processor used for sharing a busy processor's load is fixed, the only possible concurrent shared variable access can occur at line 65. This contention is between the idle processor p_i and the selected busy processor p_b. Since it involves a busy processor, we do not want to resolve it by using any type of locks or imposing any overhead on the busy processors. The contention is resolved by letting p_i directly update the $ActivityList$ of p_b, asynchronously. It is possible that p_b reads the old $ActivityList$, and tessellates some patches that are taken off its $ActivityList$ by p_i. We avoid this duplication by letting p_i re-read the current position of p_b after deleting elements from its $ActivityList$. It does not tessellate any patch of new $ActivityList[p_i]$, that p_b has already tessellated. The resulting algorithm is:

Delete the second half of unprocessed $ActivityList[p_b]$:651

Add it to $ActivityList[p_i]$:652

Read new $Front[p_b]$ (Note: p_b's variable) :653

Mark the Tessellation loop to start after new $Front$. :654

There still exists a race condition: p_b could have read its $Front$ (line 1) just before p_i updates its $ActivityList$ (at line 651), and not yet written the new $Front$ (line 2) when p_i reads it back (at line 653). If the cost of tessellating each primitive is not high, we can let the two processors duplicate the effort of tessellating one primitive when this race condition occurs, since it is quite rare. Note that line 653 appears after line 652 by design, since it reduces the probability of a race condition, assuming all processors are equally fast. However, the race condition can be completely eliminated using the following modification.

Busy processors write an additional binary shared variable, $ListTransient$. Each busy processor p sets its $ListTransient$ to 1 before reading its $ActivityList$, and resets it to 0 after having updated its $Front$. Now, we let the idle processor busy-wait[2] for $ListTransient[p_b]$ to be 0 before reading the new $Front$ of p_b.

Proof of correctness

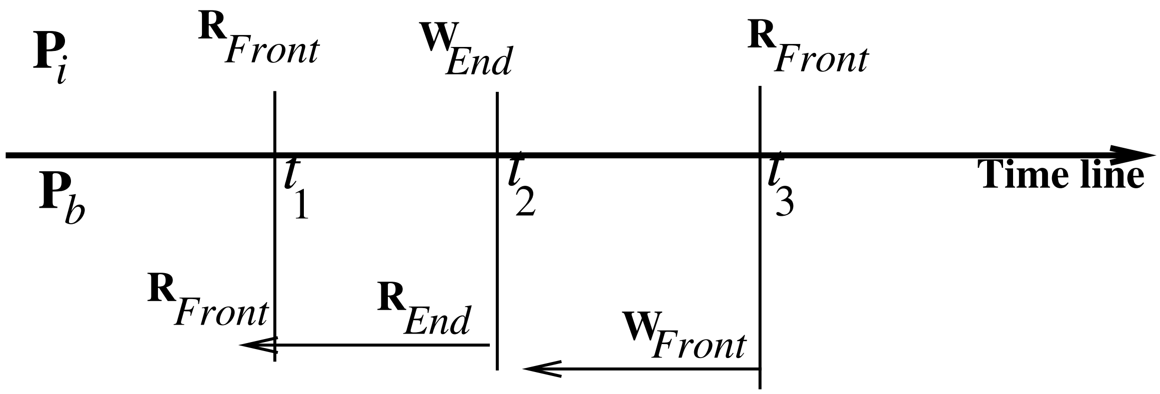

Figure 8.2: Timeline showing consistent behavior

The load-stealing algorithm guarantees the following:

(a) For a given frame, each element of the queue is read at least once.

(b) each element is read exactly once.

Guarantee (b) is not required for correctness, but improves the efficiency of tessellation. It is easy to see that (a) is true, since at every instant an element lies in at least one of the queues of p_b and p_i. It is possible, though, that some elements lie in multiple queues for a time-interval. To see that (b) still holds, consider the time line shown in Figure 8.2.

Suppose the idle processor, p_i, updates the queue of the busy processor, p_b, at time instant[3] t_2. At t_2, p_b may have read element in p_i's $ActivityList$, since it continues

[2]For multiple processors, busy-waiting is the right solution, because it does not induce a context-switch.

[3]Individual memory reads and writes happen atomically.

processing asynchronously. But the next time p_i checks its $ActivityList$, it must see the new queue, and stop. Thus the last element of p_i's queue that p_b reads must be read before t_2. This implies that the $ListTransient$ flag is set at t_2. p_b reads the position of p_i's $Front$ after t_2, say at t_3. This implies that $ListTransient$ must be reset at t_3, which, in turn, implies that p_b must have updated the $Front$ of its queue before t_3. Thus, at t_3, p_i knows exactly which elements of its queue p_b ever reads.

In theory, it is possible for p_i to get starved using our algorithm. In practice, it does make progress, specially since p_b spends considerable time in tessellating the patch corresponding to the $Front$ of its queue. p_i does not need to wait while p_b performs tessellation.

8.4 Greedy rendering

A fundamental component of real-time graphics is to have the image appear on screen on time. The quality of such an image may sometimes not be of primary concern. In the most general sense, such applications are referred to as *time-critical* rendering. We present a method for allocating the time spent in each frame in which all triangles must be sent down the rendering pipeline.

8.4.1 System architecture

Figure 8.3 shows our greedy rendering system architecture. A triangle pushing processor is allocated for the sole purpose of sending triangles down the rendering pipeline. This processor[4] runs asynchronously with the tessellators. It has a *view* of the current set of triangles to be rendered.

This triangle pushing processor executes the following loop:

```
Compute and set transformation matrix                         :P1
For each triangle in view                                     :P2
```

[4]In general, multiple pushers may be allocated, but the SGI system we used allows only one host processor to access the graphics pipeline.

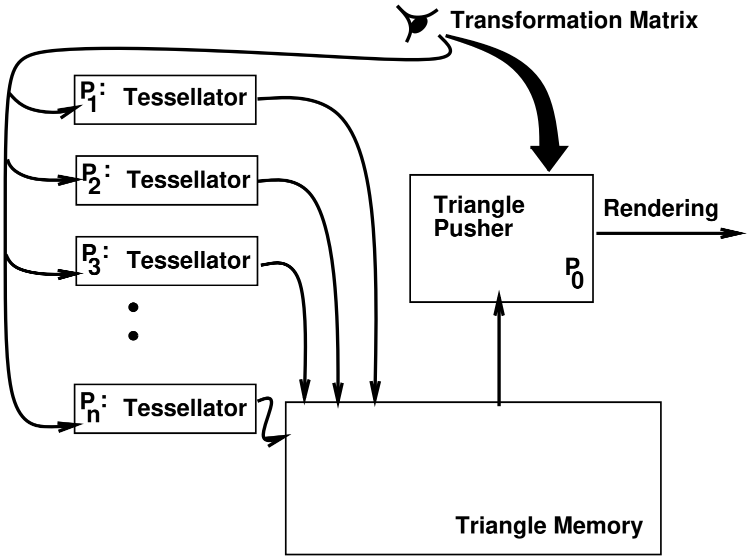

Figure 8.3: Greedy rendering technique

```
    Send triangle to the pipeline.                                :P3
```

The triangle pusher never waits for any tessellators to finish. The tessellators asynchronously generate triangles and update the pusher's *view*. The tessellators, however, are frame synchronous: a tessellator proceeds to frame number $i + 1$ only if all patches have been tessellated for frame i. The tessellators follow the algorithm presented in Section 8.3. Once a frame is complete, all tessellators read the then current view-matrix from a shared global variable, and use this matrix to generate new triangles. Clearly, if tessellation is slow, as is commonly the case, we almost always render a sub-optimal number of triangles, as the tessellators *lag* behind the pusher by a few frames. The tessellators mostly generate triangles with an old view-matrix. In our experience this is not a major problem. The tessellators typically lag by no more than $3-4$ frames. Since the view-matrix does not change significantly in $3-4$ frames, using even an "old" approximation lead to a reasonably smooth image.

8.4.2 View update

It is important to update the *view* in such a fashion that the pusher never mixes old and new triangles. In addition, patches are not independent entities. The triangles for adjacent patches must match up at the patch boundaries to prevent cracks. We partition the adjacency graph of patches into connected components. The update of all members of a connected components occurs simultaneously. For each component c there exists a shared variable $ComponentTris(c)$. $ComponentTris(c)$ is a pointer to a list of triangle addresses. Each address corresponds to the triangles of a patch, which are stored contiguously in memory. The tessellator that generates triangles for patch i of component c, stores the address of new triangles in $Newlist[i]$. Once the new triangles have been generated and the pointer list set up, the tessellator that generates the last triangle of the group writes the address of the new list in $ComponentTris(c)$.

`Generate Triangles and store at address` a

$Newlist[i] = a$

`if for all patches in the component` $NewList[i]$ `is non NULL`

    $ComponentTris(c) = NewList$

To ensure that the pusher does not read triangles from a different list, it reads ComponentTris(c) just once per frame, and saves it locally. Thus it either renders all new, or all old triangles. We modify line P2 to reflect this:

$List = ComponentTris(c)$

`for each patch` i `in component` c

    `Render all triangles at` $List(i)$

8.4.3 General environments

Our algorithm for *view* management works well for environments consisting of a number of small solid models, resulting in a number of small components. In other environments we propose a more general technique:

1. Tessellate patch boundaries separately from the patch interior

2. Generate boundary strips between the boundary and patch interior

3. For each boundary, generate a boundary strip, N_i, between the new interior tessellation and the new boundary tessellation, and a strip, O_i, between the new interior tessellation and the old boundary tessellation.

4. If an adjacent boundary has not been tessellated for the current frame, use O_i. If the adjacent boundary has been tessellated, use N_i. Again, the second processor to tessellate a boundary updates the address for both processors. Thus boundary tessellations are always consistent.

In practice, we have found it much more efficient to render the images with cracks, and let the cracks 'fill up' when the user stops, and the tessellation catches up.

As a final note, let us emphasize that this method is not necessarily bound to multiple processor environments. For a given application, even single processors can allocate triangle-pushing time per frame. For example, suppose it takes t_p seconds to process each triangle. For each frame the triangle pushing thread maintains the cardinality, m, of the set of triangles available for rendering in that frame. Suppose further that the desired frame rate is R frames per second. The system must allocate no more than $\frac{1}{R} - mt_p$ seconds for triangle generation in any frame.

8.5 Performance

On Pixel-Planes 5, we implemented our static load balancing scheme presented in section 8.2 and compared its performance with round-robin distribution. With a configuration of 25 GPs, our load-distribution scheme shows an average speed-up of 10%. We evaluated our algorithms on a number of models, including various parts of the submarine storage and handling room. Table 8.1 shows the increase in average processor utilization using the graph partitioning algorithm, and the actual speed-up gained.

We also implemented our algorithms on a Silicon Graphics Onyx with 4 host CPUs and Reality Engine II graphics. The memory (and address space) is shared by all processors. On this system, we implemented the "static", "global queue", and

Model	# Patches	Round-Robin Distribution		Graph Partioning	
		Frames/Sec	Utilization	Frames/Sec	Utilization
Torpedo	1201	10.80	72.6%	11.74	80.1%
Pivot	4101	15.41	61.9%	15.95	81%
Ship	3392	19.05	65.3%	19.86	78%

Table 8.1: Performance of static load-distribution algorithms

"load stealing" load distribution schemes, and compared their performances. For experiments on large processor configurations we used up to 16 CPUs on a parallel Challenge machine, which has a similar architecture. Figure 8.4 shows three graphs with models of 5302, 10,604, and 15,906 trimmed NURBS patches respectively. Each graph shows the change of the tessellation rate[5] with different number of processors. The graphs show that the global queue scheme provides 5% to 10% improvement over the static scheme, whereas the load stealing scheme provides another 10% to 20% improvement over the global queue scheme. In addition, as the number of processors increases, the load stealing scheme shows the best scaling behavior. The speed-up is close to linear.

Model	# Patches	Tessellations/Sec	Updates/Sec	Average Lag
Dragon	5354	4.20	19.96	4.75
Car 1	5053	4.84	19.97	4.12
Car 2	8693	2.20	8.55	3.88

Table 8.2: Performance of the greedy rendering algorithm

If a large number of processors are not available, or if the size of the model is large, the tessellation rate drops to under 5 frames per second. With greedy rendering, we

[5] Since greedy rendering proceeds asynchronously, the frame-update rate is hardly affected. Only the lag changes with the tessellation rate.

are consistently able to render at the speed of about 4 times the actual tessellation rate. Table 8.2 shows the performance of the greedy algorithm on a number of models. Column 5 shows the average number of frames the tessellators lag behind the triangle pusher, and represents the speed-up gained by the greedy algorithm. This speed-up comes at a minimal loss in image quality. The Color plate C shows the images corresponding to the worst lag in a user sequence for three parts of the submarine storage and handling system. As can be seen, even in the worst case, the lagging images are of reasonable quality. Thus this technique achieves about 5 times speed-up with little visual erosion.

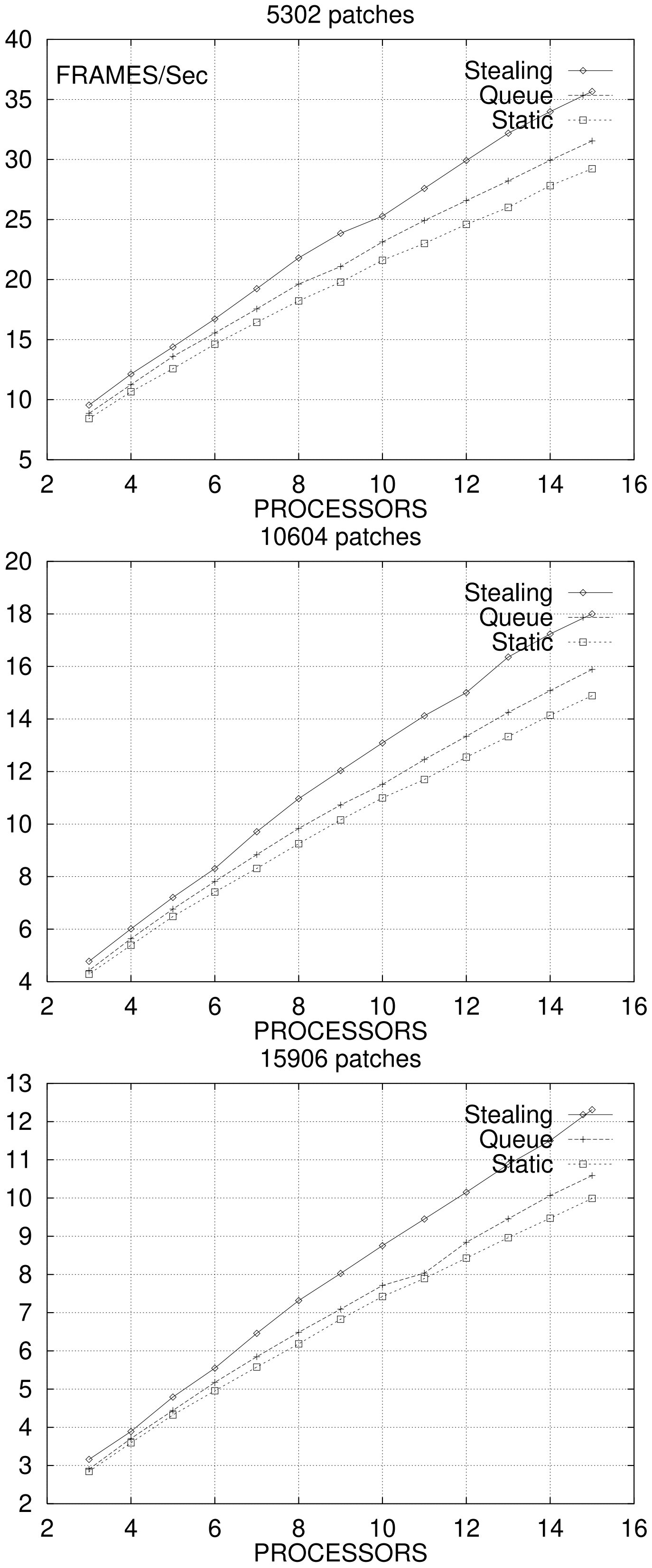

Figure 8.4: Performance comparison of three load-balancing techniques

Chapter IX

On triangular and multisided domains

9.1 Overview

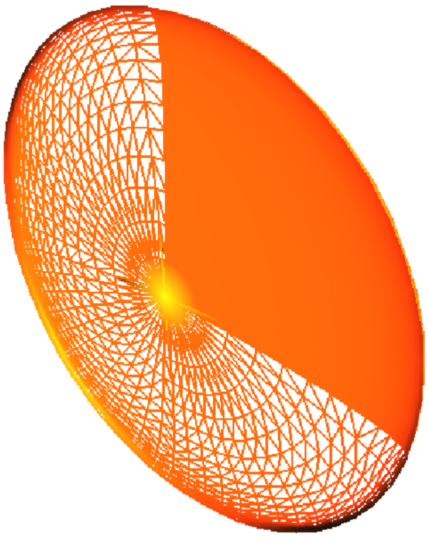

Figure 9.1: The teapot bottom: a triangle-like patch

In most of this dissertation we assume a tensor product NURBS surface, one that is parametrized over a four sided domain: $[0, 1] \times [0, 1]$. Although a wide variety of surfaces can be represented parametrically and may be parametrized over a four sided domain it is often more convenient to use a different shape for the domain.

The rectangular parametrization often induces unnecessary degeneracies on many surfaces that have different inherent shapes. For example, the bottom of the Utah teapot, represented as a set of Bézier patches, consists of four Bézier patches as shown in Figure 9.1. The problem is that each of these patches is three sided. A four sided parametrization is obtained by shrinking one of the four boundary curves, say $v = 0$, to a point. This implies that the partial derivative, $\mathbf{F}_v$, of the surface vanishes at that point. This, in turn, implies that the pseudo-Gauss map contains the origin. In addition, the uniform triangulation in the vicinity of such points tends to result in skinny triangles. A triangular domain provides a smoother and better behaved parametrization for such a surface.

In this chapter we show how our results can be extended to 3-sided and other n-sided domains. For simplicity of presentation we assume that the domain is a regular n-gon.

9.1.1 Bézier triangles

Bézier surfaces defined over a triangular domain are often called Bézier triangles. In fact, de Casteljau [de 59, de 63] first devised Bézier triangles, which were later generalized to tensor-product surfaces. Sabin [Sab77] first generalized Bézier to triangular B-splines, The theory of triangular NURBS has been further developed recently by Dahmen et al. [DMS92]. Bézier triangles are obtained from triangular NURBS surfaces by repeated knot insertion. In this section we present an algorithm to tessellate Bézier triangles.

We use the barycentric coordinates of the domain triangle. In Figure 9.2, the triangle $\mathbf{abc}$ represents the domain triangle, each point on which is represented by three coordinates, (u, v, w). The coordinates of a point $\mathbf{q}$ are:

$$u = \frac{\mathcal{A}(\mathbf{q}, \mathbf{b}, \mathbf{c})}{\mathcal{A}(\mathbf{a}, \mathbf{b}, \mathbf{c})}, \ v = \frac{\mathcal{A}(\mathbf{a}, \mathbf{q}, \mathbf{c})}{\mathcal{A}(\mathbf{a}, \mathbf{b}, \mathbf{c})}, \ w = \frac{\mathcal{A}(\mathbf{a}, \mathbf{b}, \mathbf{q})}{\mathcal{A}(\mathbf{a}, \mathbf{b}, \mathbf{c})},$$

where $\mathcal{A}$ is the area function. Notice that $u + v + w = 1$, thus there are only two

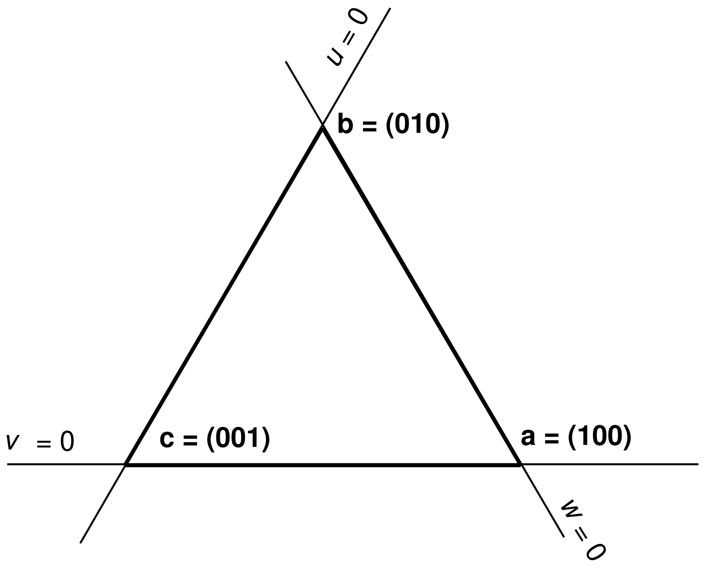

Figure 9.2: Triangular domain and barycentric coordinates

independent variables. A triangular Bézier patch of degree n is given by:

$$\mathbf{F}(\mathbf{u}) = \frac{\sum_{|\mathbf{i}|=n} w_{\mathbf{i}} \mathbf{p}_{\mathbf{i}} \mathcal{B}_{\mathbf{i}}^n(\mathbf{u})}{\sum_{|\mathbf{i}|=n} w_{\mathbf{i}} \mathcal{B}_{\mathbf{i}}^n(\mathbf{u})}, = \frac{\mathbf{P}(\mathbf{u})}{W(\mathbf{u})} \tag{IX.1}$$

where $\mathbf{u}$ is a point on the domain represented by some (u, v, w), $\mathbf{i}$ is a vector of three numbers (i, j, k), and the $\mathbf{p_i}$ form a triangular mesh of control points.

$$\mathcal{B}_{\mathbf{i}}^n(\mathbf{u}) = \frac{n!}{i!j!k!} u^i v^j w^k;\ i + j + k = n$$

The three boundary curves of a triangular Bézier patch, $u = 0, v = 0$, and $w = 0$ are standard Bézier curves. Instead of partial derivatives with respect to the two axes of the rectangular domain, we now compute directional derivatives. The cross product of two directional derivatives produces the normal at a point on the surface. In particular, we are interested in the directional derivatives in two of the boundary directions, along $u = 0$ and along $v = 0$.

The directional derivatives of a Bézier surface are again triangular Bézier surfaces themselves. In general the polynomial derivative in the direction $\mathbf{d} = (d, e, f)$ is:

$$D_{\mathbf{d}}(\mathbf{P}(\mathbf{u})) = n \sum_{|\mathbf{i}|=n-1} [d w_{\mathbf{i+e1}} \mathbf{p}_{\mathbf{i+e1}} + e w_{\mathbf{i+e2}} \mathbf{p}_{\mathbf{i+e2}} + f w_{\mathbf{i+e3}} \mathbf{p}_{\mathbf{i+e3}}]\, \mathcal{B}_{\mathbf{i}}^{n-1}(\mathbf{u}),$$

where $\mathbf{e1} = (1,0,0), \mathbf{e2} = (0,1,0)$ and $\mathbf{e1} = (0,0,1)$.

In particular, the derivative in the direction u is

$$D_u(\mathbf{P}(\mathbf{u})) = n \sum_{|\mathbf{i}|=n-1} (\mathbf{p}_{\mathbf{i}+\mathbf{e2}} - \mathbf{p}_{\mathbf{i}+\mathbf{e3}})\mathcal{B}_{\mathbf{i}}^{n-1}(\mathbf{u})$$

and the derivative in the direction v is

$$D_v(\mathbf{P}(\mathbf{u})) = n \sum_{|\mathbf{i}|=n-1} (\mathbf{p}_{\mathbf{i}+\mathbf{e1}} - \mathbf{p}_{\mathbf{i}+\mathbf{e3}})\mathcal{B}_{\mathbf{i}}^{n-1}(\mathbf{u}).$$

Note the similarity with rectangular surfaces; the pairwise difference of rows of control points of the surface result in the control points of the derivative. We again use the quotient rule for derivatives of rationals.

9.1.2 Sampling

It is easy to reparametrize a triangular domain onto a rectangular domain [IW92] and apply all results described earlier in this dissertation, but that defeats the whole purpose of using triangular Bézier surfaces. A better solution may be to extend the domain by reflection about $w = 0$. The domain becomes rectangular; we can then trim out the extra portion of the domain using a trimming curve. This solution may be acceptable but requires computation and storage of extra information. It also induces the overhead of trimming-curve processing. We instead propose a tessellation algorithm that generalizes to arbitrary sided domains.

Much of our discussion of sampling requirements in the context of tensor product patches holds for triangular domains as well:

1. We sample the interior separately from the boundary curves to avoid cracking.
2. We compute the required step size on the domain so that the screen space quality constraints are satisfied.
3. Our algorithm for computing the sampling size holds for general surfaces and are applicable to triangular Bézier patches as well. As long as we can find a bound on the derivatives, we can compute the step sizes. Instead of two partial

derivatives we now use the derivatives in two directions. We choose $u = 0$ and $v = 0$.

4. Evaluation of points along a parametric line parallel to one of the boundaries can again be optimized. To see how, let us rewrite the polynomial Bézier triangle equation (see Equation IX.1) for a fixed v_0 as:

$$\mathbf{F}(\mathbf{u}) = \sum_{i=0}^{n}\sum_{j=i}^{n} \mathbf{p}_{(i,j,n-i-j)} \frac{n!}{i!j!(n-i-j)!} u^i v_0^j (1 - v_0 - u)^{(n-i-j)}$$

Again, we can re-use part of the computation for all points with $v = v_0$.

9.1.3 Triangulation

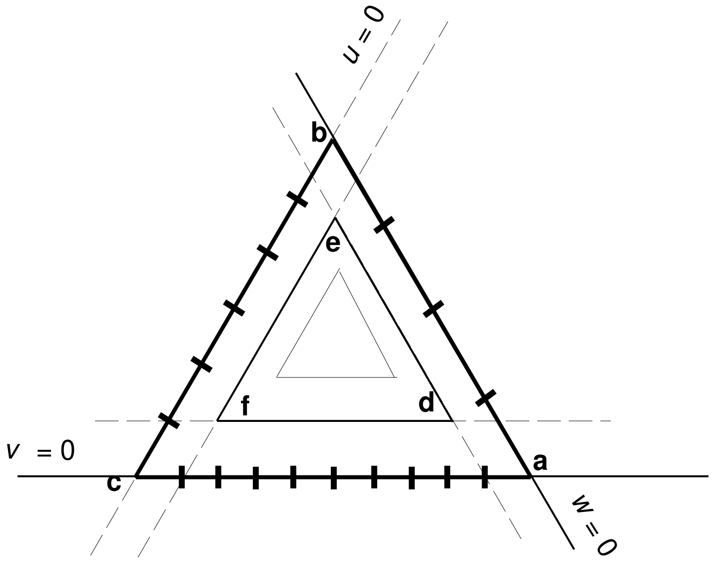

Figure 9.3: Triangulation in the triangular domain

Let the number of steps needed on three boundaries be δ_1, δ_2 and δ_3 respectively. Let the step size needed along the direction $u = 0$ be δ_u and that along $v = 0$ be δ_v. Also denote $\lceil \frac{1}{\delta_i} \rceil$ by n_i. The main steps of the algorithm are as follows:

1. First pick n_i equi-spaced samples on the i^{th} boundary (see Figure 9.3). Stop if each boundary requires just two samples. (Two samples imply a single step.)

2. Form a smaller triangle, nested inside the original triangle, offset from the boundary.

3. Triangulate the region between the two triangles.

4. Update the number of sample points needed on the boundary of the inner triangle.

5. Recurse with the smaller triangle as the new domain.

We must still determine how much to offset the inner triangle and how many samples to choose on the inner triangle.

Consider the curve $v = 0$; the corresponding offset curve is $v = \delta$ where $\delta = \min(\delta_u, \delta_w)$ as shown in Figure 9.3. We find the offset curve for the other two boundaries separately. Note that the boundary $w = 0$ participates in this computation, but subsequent lines $w = k$ do not, since we only compute step sizes along u and v in the interior of the polygon. The offset distance is determined by δ_u and δ_v.

Computation of the number of samples on the new boundary $v = \delta$ is also straightforward. We compute the length of this domain edge, for example the edge of length $v = \delta$ in Figure 9.3 is $1 - 2\delta_v$. Hence we must pick $\lceil \frac{1-2\delta_v}{\delta_v} \rceil$ samples on this curve. We can now produce a filling between $v = 0$ and $v = \delta$ just as we do for the boundary of rectangular patches. Only, in this case we repeat a similar procedure in the interior as well.

The algorithm for processing trimming curves is similar to that for rectangular domains. We trace the curve over the set of nested triangles, each with a certain number of segments on it. We mark each triangle segment the trimming curve crosses. At triangulation time, we generate a staircase offset with the trimming curve. The algorithm presented in Section 5.4 is directly applicable here.

9.2 Multisided domains

Just as some surfaces are inherently triangular, and are better parametrized on a triangle, there are others that are inherently pentagonal, or hexagonal and so on. There exist parametrizations defined over multisided convex domains. Gregory [Gre83] presents some of the pioneering work on multisided patches. Herron [Her79] describes an algorithm to decompose general multisided patches into triangular and rectangular patches. More recently, Loop et al. [LD89] present a unifying framework for multisided patches, or S-patches. They define surfaces by embedding the m sided domain in a $m-1$ dimensional space. The result is a generalization of barycentric coordinates.

Our tessellation algorithm for triangular Bézier patches generalizes quite easily to S-patches. Instead of nested triangles, we use nested m-gons. At the end of the recursive process, we are left with a k-gon, $k \leq m$. We can easily triangulate it, as it is a convex polygon. Notice how this method gives rise to nested rectangles for tensor product surfaces. That is same as the uniform tessellation method described earlier.

Chapter X

Epilogue

10.1 Conclusion

Rendering of increasingly more complex models using limited hardware resources requires high software processing. The fact is, given the screen resolution, only a bounded amount of detail can ever be displayed. Even for large models, usually only a small fraction is visible. Furthermore, not all visible geometry needs to be rendered at significant detail. That is why a rendering system that can display larger and larger models at interactive rates is more than a mere dream. With current graphics systems incorporating powerful general purpose processors, it has become possible to perform compute-intensive operations. The major issues to consider are:

- *Visibility*: Quickly eliminate geometry not visible from the current view point.
- *Value*: Quickly determine the "significance" of different parts of the visible set of geometry.
- *Version*: Maintain multiple conceptual versions of the model. For each part of the model, generate an image with detail commensurate with its value.

If these algorithms scale gracefully, and exhibit logarithmic or sub-linear behavior[1], we can expect the overall rendering to scale gracefully. In this dissertation

[1]Note that we do not need to process each surface at each frame, if the data is hierarchically organized

we have explored each of these three issues and proved the thesis:

> `We can perform interactive display of models consisting of many thousands of spline surfaces on current graphics systems.`

The basic thrust of our research has been to maintain analytic representation of NURBS surfaces and optimize their triangulation in order to exploit fast triangle rendering hardware. Since most geometric design, in fact, involves constructive solid geometry and higher-order surfaces, such representations can be easily available. Off-line polygonization of these models does not guarantee interactive rendering and only leads to loss of analytic information. This dissertation shows that interactive display of spline models is possible without off-line polygonization. Our prototype implementation, SPEED, is able to render more than five thousand Bézier patches more than 10 times a second on an SGI Onyx with Reality Engine II.

We have been able to optimize the tessellation process, and it is no longer an impediment to interactive rendering. Moreover, this tessellation is performed on general purpose processors, and can be parallelized with little overhead. In addition, as the number of patches in a model increases, the size of each patch on screen decreases and thus the cost per patch decreases. Hence our rendering algorithm suffers only sub-linear slow-down with increasing model sizes.

Our method is an advancement over the standard mesh simplification methods which back-engineer several levels of detail from a dense polygonization. We do not compute discrete levels of detail but rather generate almost continuous levels of detail directly from the surface representation. In addition, this *update of detail* can be done asynchronously with the regular rendering pipeline, causing no increase in the rendering time, and incurring only a small loss in visual fidelity. This method also gives users an automatic dial to control the rendering speed *vis a vis* the visual quality.

The higher-order representation of objects allows us to generate at rendering time, an "appropriate" set of triangles, in order to increase both the visual realism and rendering speed at the same time. This a notable advance in our quest for increased realism at increased speed. However, it merely scratches the periphery of an immensely promising, and wide open, realm of research in representations of geometry that allow quick resolution of the three *V*s: visibility, value, and version.

10.2 Future Work

Our desire for increased speed can never be satiated; we always want faster algorithms. Even with the speed-up achieved by this higher-order representation, we cannot render models with, say, more than a hundred thousand Bézier patches at rates of 30 frames a second[2]. We suggest possible directions for future research to improve the results of this dissertation.

- **Higher order representation**: The number of Bézier patches in real models increases rather quickly. Our method produces at least 2 triangles for each Bézier patch. Trimming curves add further complexity. This may result in too many triangles for parts of a model. For example, a small part like the Utah teapot consists of 32 Bézier patches. However, 64 triangles are not required to represent it when its size is, say, a few pixels on screen. We could benefit from a spline simplification algorithm to reduce the number of Bézier patches. Further development of the theme of higher-order representations is another promising avenue for research. For instance, we could find a representation for a collection of patches and tessellate each such collection into a small number of triangles.

- **Visibility**: Efficient visibility processing can significantly impact rendering performance for large models that are spatially well distributed. There is scope for further improvements in visibility pre-computation and occlusion culling.

- **System issues**: For large data-sets, efficient memory management becomes a crucial issue. This is another area of future research. So is improvement in load balancing on distributed memory processors.

[2]Desirable for perception of smooth motion

- **Sampling**: In addition to speeding up the display further, there are other possible improvements to our results. We have used parametric properties, e.g. parametric distances, to compute the sampling for tessellation. Geometric distances, while more difficult to evaluate, should avoid some effects of parametrization and, in general, produce more regular triangulations. On a related note, re-parametrizing surfaces for efficient and high-fidelity display is another open area of research. More investigation into better hybrids between adaptive sampling and uniform sampling techniques is another prospective research area. It would also be useful to further analyze the trade-off between the exactness of bound computation and the number of triangles generated by an algorithm. Unfortunately there is no clear way to characterize the time taken by an algorithm to compute a bound and the tightness of its results.

- **Constant frame rate**: Currently our system is driven by a user-specified quality parameter. On the other hand, for some applications a steady frame rate is quite desirable. It should be possible to choose the parameter so that a user-specified frame rate is maintained.

- **Dynamic models**: We have not directly addressed dynamically changing models in our dissertation. However, such changes are likely to be local in nature and it should be possible to update the corresponding pre-computation using incremental algorithms. Further research is needed to allow real-time modeling.

Bibliography

[AES91] S.S. Abi-Ezzi and L.A. Shirman. Tessellation of curved surfaces under highly varying transformations. *Proceedings of Eurographics*, pages 385–397, 1991.

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

[Ake93] K. Akeley. Reality Engine Graphics. In *Proceedings of ACM SIGGRAPH*, pages 109–116, 1993.

[Ali96] D. Aliaga. Visualization of complex models using dynamic texture-based simplification. In *Proceedings of IEEE Visualization'96*, 1996.

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

[BE92] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 23–90. World Scientific, Singapore, 1992.

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

[BKSS90] N. Beckmann, P. Kriegel, R. Schneider, and B. Seeger. R* tree: An efficient and robust access method for points and rectangles. In *International Conference on Management of Data*, pages 322–331, 1990.

[Bli78] J. F. Blinn. *Computer Display of Curved Surfaces.* Ph.d. thesis, University of Utah, 1978.

[BR94] C.L. Bajaj and A. Royappa. Triangulation and display of rational parametric surfaces. In *Proceedings of IEEE Visualization'94*, pages 69–76, IEEE Computer Society, Los Alamitos, CA, 1994.

[Bro96] F. Brooks. Newport news shipbuilding company interview re ship design process. Technical Report TR96-002, Department of Computer Science, University of North Carolina, 1996. Not available publicly.

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

[CCT92] K. L. Clarkson, R. Cole, and R. E. Tarjan. Randomized parallel algorithms for trapezoidal diagrams. *Internat. J. Comput. Geom. Appl.*, 2(2):117–133, 1992.

[Che95] S.E. Chen. QuickTime vr – an image-based approach to virtual invironment navigation. In *Proceedings of ACM SIGGRAPH*, pages 29–38, 1995.

[Cla76] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of ACM*, 19(10):547–554, 1976.

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

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

[de 59] P. de Casteljau. Outillages méthodes calcul. Technical report, A. Citroen, Paris, 1959.

[de 63] P. de Casteljau. Courbes et surfaces à pôles. Technical report, André Citroën automobilies, Paris, 1963.

[de 78] C. de Boor. *A practical guide to splines.* Springer Verlag, New York, 1978.

[DLW93] T. DeRose, M. Lounsbery, and J. Warren. Multiresolution analysis for surfaces of arbitrary topology type. Technical Report TR 93-10-05, Department of Computer Science, University of Washington, 1993.

[DMS92] W. Dahmen, C. Michelli, and H.P. Seidel. Blossoming begets B-spline bases built better by B-patches. *Mathematics of Computation*, 59(199):97–115, 1992.

[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.

[DP89] L. De Floriani and E. Puppo. A survey of constrained Delaunay triangulation algorithms for surface representaion. In G. G. Pieroni, editor, *Issues on Machine Vision*, pages 95–104. Springer-Verlag, New York, NY, 1989.

[DZ81] M.J. Dehaemer and M.J. Zyda. Simplification of objects rendered by polygonal approximations. *Computer and Graphics*, 15(2):175–184, 1981.

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

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

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

[FKN80] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priority tree structure. In *Proceedings of ACM SIGGRAPH*, pages 124–133, 1980.

[FM84] A. Fournier and D. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3:153–174, 1984.

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

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

[FS93] T.A. Funkhouser and C.H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of ACM SIGGRAPH*, pages 247–254, 1993.

[FVFH93] J. Foley, A. VanDam, S. Feiner, and J. Hughes. *Computer Graphics principles and practice*. Addison Wesley, Menlo Park, California, 1993.

[GGSC96] S. Gortler, R. Grzeszczuk, R. Szelski, and M. Cohen. The lumigraph. In *Proceedings of ACM SIGGRAPH*, pages 43–54, 1996.

[Gre83] J. Gregory. C^1 rectangular and non-rectanguar surface patches. In R. Barnhill and W. Boehm, editors, *Surfaces and CAGD*, pages 25–33. North-Holland, Amsterdam, 1983.

[HDD+93] H. Hoppe, T. DeRose, T. Duchamp, J. Mcdonald, and W. Stuetzle. Mesh optimization. In *Proceedings of ACM SIGGRAPH*, pages 19–26, 1993.

[Her79] G. Herron. *Triangular and Multisided Patch Schemes.* PhD thesis, Dept. of Mathematics, University of Utah, Salt Lake City, 1979.

[HL95] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *Proc. Supercomputing '95*, 1995.

[Hop96] H. Hoppe. Progressive meshes. In *Proceedings of ACM SIGGRAPH*, pages 99–108, 1996.

[Hun78] G. Hunt. *Efficient Computation and Data Structure for Graphics.* PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, 1978.

[IW92] K. Iino and D. Wilde. Subdivision of triangular Bézier patches into rectangular Bézier patches. *Transactions of the ASME*, 1992.

[Jae80] J. Jaech. Surface rendering by computer and a scan-line solution to the cracking problem. Master's thesis, Computer Science Department, University of Washington, 1980.

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

[Kel95] P. Keller. Parallel tessellation of Bézier surface patches using a SIMD processor array. Master's thesis, Computer Science Department, University of North Carolina at Chapel Hill, 1995.

[KK95] 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.

[KKMN95] S. Kumar, S. Krishnan, D. Manocha, and A. Narkhede. Representation and display of complex CSG models. Technical Report TR95-019, Department of Computer Science, University of North Carolina, 1995.

[KM84] P.A. Koparkar and S. P. Mudur. Computational techniques for processing parametric surfaces. *Computer Vision, Graphics and Image Processing*, 28(3):303–322, 1984.

[KM94a] S. Krishnan and D. Manocha. Global visibility and hidden surface algorithms for free form surfaces. Technical Report TR94-063, Department of Computer Science, University of North Carolina, 1994.

[KM94b] S. Kumar and D. Manocha. Interactive display of large scale trimmed NURBS models. Technical Report TR94-008, Department of Computer Science, University of North Carolina, 1994.

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

[KS95] 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.

[Kum95] S. Kumar. A survey of polygon triangulation algorithms. Technical Report TR96-011, Department of Computer Science, University of North Carolina, 1995.

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

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

[LD89] C. Loop and T. DeRose. A multisided generalization of Bézier surfaces. *ACM Transactions on Graphics*, 8(3):204–234, 1989.

[LG95] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Symposium on Interactive 3D Graphics*, Monterey, CA, 1995.

[LH96] M. Levoy and P. Hanrahan. Light field rendering. In *Proceedings of ACM SIGGRAPH*, pages 31–42, 1996.

[LR81] J.M. Lane and R.F. Riesenfeld. Bounds on polynomials. *BIT*, 21(1):112–117, 1981.

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

[MB95] L. Mcmillan and G. Bishop. Plenoptic modeling: An image based rendering system. In *Proceedings of ACM SIGGRAPH*, pages 39–46, 1995.

[MD92] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves. In *Proceedings of Graphics Interface*, pages 232–241, 1992.

[MEP92] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High speed rendering using image composition. *ACM Computer Graphics*, 26(2):231–248, 1992. (SIGGRAPH Proceedings).

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

[NM95] A. Narkhede and D. Manocha. Fast polygon triangulation based on Seidel's algorithm. In A. Paeth, editor, *Graphics Gems V*, Academic Press, 1995.

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

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

[O'R94] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.

[PR95] L. Piegl and A. Richard. Tessellating trimmed NURBS surfaces. *Computer Aided Geometric Design*, 27(1):16–26, 1995.

[Pre75] P. M. Prenter. *Splines and Variational Methods*. Wiley, New York, 1975.

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

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

[Sab77] M. Sabin. *The Use of Piecewise Forms for Numerical Representation of Shape*. PhD thesis, Hungarian Academy of Sciences, Budapest, 1977.

[SAE93] L.A. Shirman and S.S. Abi-Ezzi. The cone of normals technique for fast processing of curved patches. In *EUROGRAPHICS*, pages 261–272, 1993.

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

[Sed83] T.W. Sederberg. *Implicit and Parametric Curves and Surfaces*. PhD thesis, Purdue University, 1983.

[Seg95] M. Segal, 1995. Silicon Graphics, Personal Communication.

[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.

[SH93] X. Sheng and B. Hirsch. Triangulation of trimmed surfaces in parametric space. *Computer-Aided Design*, 24(8):437–444, August 1993.

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

[SLS+96] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching. In *Proceedings of ACM SIGGRAPH*, pages 75–82, 1996.

[SZL92] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen. Decimation of triangle meshes. *ACM Computer Graphics*, 26(2):65–70, 1992. (SIGGRAPH Proceedings).

[TFFH94] S. Teller, C. Fowler, T. Funkhouser, and P Hanrahan. Partitionaing & ordering large radiosity computations. In *Proceedings of ACM SIGGRAPH*, pages 443–450. ACM Press, 1994.

[TS91] S. Teller and C. Séquin. Visibility preprocessing for interactive walkthroughs. *ACM Computer Graphics*, 25(4):61–69, 1991. (SIGGRAPH Proceedings).

[Tur92] G. Turk. Re-tiling polygonal surfaces. *ACM Computer Graphics*, 26(2):55–64, 1992. (SIGGRAPH Proceedings).

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

[Var94] A. Varshney. *Hierarchical Geometric Approximations*. PhD thesis, University of North Carolina, 1994.

[VB93] M. Vigo and P. Brunet. Piecewise linear approximation of trimmed surface. In G. Farin, H. Hagen, and H. Noltemeier, editors, *Geometric Modeling*, 1993.

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

[Wal96] Unc walkthrough group video progress report, 1996. University of North Carolina at Chapel Hill.

[Wat95] J. Watts. A practical approach to dynamic load balancing. Master's thesis, Caltech Computer Science Department, 1995.

[Whi78] J.T. Whitted. A san line algorithm for computer display of curved surfaces. *ACM Computer Graphics*, 12(3):8–13, 1978. (SIGGRAPH Proceedings).

[Whi79] J.T. Whitted. An improved illumination model for shaded display. *ACM Computer Graphics*, 13(3):1–14, 1979. (SIGGRAPH Proceedings).

[Won84] E. Wong. Tessellation of variable order polynomial surface and polyhedral geometries. Master's thesis, Computer Science Department, Rensselaer Polytechnic Institute, 1984.