Partitioning Trimmed Spline Surfaces into Non-Self-Occluding Regions for Visibility Computation *

Shankar Krishnan
AT&T Research Labs
180 Park Avenue, Room E-201
Florham Park, NJ 07932
U.S.A.
e-mail: krishnas@research.att.com

Dinesh Manocha
Department of Computer Science,
University of North Carolina,
Chapel Hill, NC 27599-3175
U.S.A.
e-mail: dm@cs.unc.edu

Abstract

Computing the visible portions of curved surfaces from a given viewpoint is of great interest in many applications. It is closely related to the hidden surface removal problem in computer graphics and machining applications in manufacturing. Most of the earlier work has focused on discrete methods based on polygonization or ray-tracing and hidden curve removal. In this paper we present an algorithm for decomposing a given surface into regions such that each region is either completely visible or hidden from a given viewpoint. Initially, it decomposes the domain of each surface based on silhouettes and boundary curves. To compute the exact visibility, we introduce a notion of *visibility curves* obtained by projection of silhouette and boundary curves and decomposing the surface into non-overlapping regions. These curves are computed using marching methods and we present techniques to compute all the components. The non-overlapping and visible portions of the surface are represented as trimmed surfaces and we present a representation based on polygon trapezoidation algorithms. The algorithms presented use some recently developed algorithms from computational geometry like triangulation of simple polygons and point location. Given the non-overlapping regions, we use an existing randomized algorithm for visibility computation. We also present results from a pre-liminary implementation of our algorithm.
Keywords: Spline Surfaces, Hidden Surface Removal, 3D Computer Graphics, NC Machining, Visibility

*Supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, NSF Grant CCR-9625217, ONR Young Investigator Award and Intel

1 Introduction

The problems of visibility and accessibility computations are fundamental for computer graphics, computer-aided design and manufacturing applications. In particular, hidden line and surface removal algorithms in computer graphics are related to visibility computations [FDHF90, Hor84, SSS74, HG77]. Similarly, accessibility computations in manufacturing applications are based on Gauss maps and visibility sets [Woo94, CCW93, GWT94]. These problems have been extensively studied in computer graphics, computer-aided design, computational geometry and manufacturing literature. In this paper, we are dealing with algebraic surfaces and surfaces defined using rational splines [Far93] that are differentiable.

Given a viewpoint, the hidden surface removal problem deals with computation of the surface boundary visible from that viewpoint. Most of the earlier algorithms in the literature are for planar and polygonal primitives and hidden lines removal [FDHF90, Mul89, SSS74]. In computational geometry literature, many of the hidden surface algorithms simply calculate the entire arrangement of lines (projections of edges and vertices of the objects on the viewing plane). Output-sensitive hidden surface algorithms were developed for special input cases like *c-oriented solids* [GO87], *axis-parallel rectangles* [PVY92] and *polyhedral terrains* [RS88]. Very few algorithms are able to cope with *cycles* (impossible to obtain an ordering among the faces without splitting some of them) efficiently. A randomized algorithm to generate the visibility map was given by Mulmuley [Mul90] for the general case. The algorithm maintains the trapezoidation of the visibility map and updates it by randomly adding one face at a time. The algorithm is (almost) output-sensitive. Extensions of the hidden surface algorithm from planar to curved faces are described in [Mul90]. A survey of most of the recent results in computational geometry regarding object-space hidden surface removal is presented in [Dor94].

When dealing with curved surfaces, most hidden surface removal algorithms must be capable of manipulating semi-algebraic sets [Mul89]. Results from elimination theory and algebraic decision procedures like Gröbner

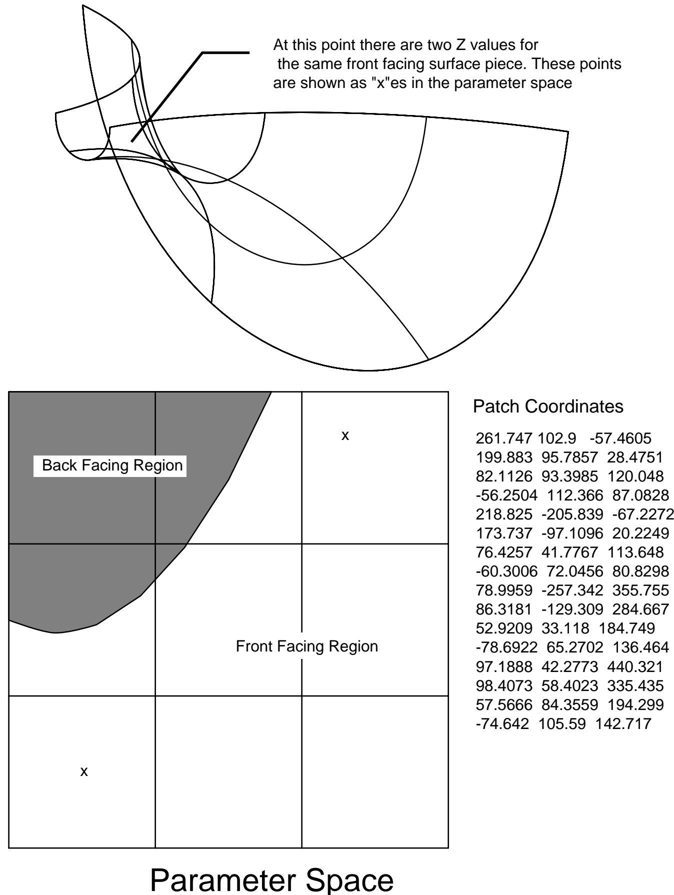


Figure 1: Local Visibility Computations Based on Silhouettes

bases are usually used for this purpose [Can88]. Unfortunately, algorithms based entirely on symbolic manipulation require infinite precision to represent algebraic numbers. Bounds based on gap theorems [Can88] have been used to compute their bit-complexity. However, implementations of these algorithms are very non-trivial and applicability of these bounds in practical situations are still not clear.

Given a model composed of algebraic or parametric surfaces, it can be polygonized and algorithms developed for polygonal models can then be applied. However, the accuracy of the overall algorithm is limited by the accuracy of the polygonal approximation. Other algorithms for visibility computations are based on ray-tracing or scan-line conversion [FDHF90]. These algorithms are slow (may take a few seconds for each patch) and lead to data proliferation. Furthermore, their accuracy is limited by the image or device precision. These techniques are device resolution dependent and many applications in modeling and rendering demand a **device-independent** representation [TW93]. For example, many data standards for 2-D and 3-D models (e.g. Postscript language) use higher order or device-independent representations.

More recently, a hidden curve removal algorithm has been presented for parametric surfaces by Elber and Cohen [EC90] based on *silhouette curves*. A *silhouette curve* is defined as the loci of points on the surface where the normal vector is orthogonal to the viewing direction. In this paper [EC90], they extract the curves of interest by considering boundary curves, silhouette curves, iso-parametric curves and curves along C^1 discontinuity based on 2D curve-curve intersections. Given a curved surface model and a viewpoint, the silhouettes on the model partition it into front facing and back facing regions (as shown in Figure 1). The surfaces obtained after partitioning based on the silhouette computation need not be completely visible, as shown in Figure 1.

We present an algorithm for decomposing a spline surface into non-overlapping regions from a given view-

point. Given a model, we represent it as a series of Bézier surfaces using knot-insertion algorithms. Each Bézier surface is partitioned into non-overlapping regions (these regions can overlap with each other but not with themselves) based on silhouette and *visibility curves* (refer Section 5). Each computed region has the property that it is either entirely visible or invisible in the absence of other surfaces. The visibility curves are computed based on the silhouette and boundary curves. Each non-overlapping region is represented as a *trimmed* Bézier surface bounded by silhouettes, visibility and boundary curves. Given a collection of these trimmed surfaces, we then use a slight variation of an existing randomized algorithm [Mul89] to compute the visible portions for hidden surface removal.

In this paper we assume that the input model is composed of non-intersecting surfaces. Given any arbitrary model, we initially decompose into non-intersecting surfaces using our surface intersection algorithm [KM97]. This algorithm uses a combination of symbolic techniques and results from numerical linear algebra. We have implemented the algorithm for decomposing each surface into non-overlapping regions in finite precision (using 64-bit double precision floating-point arithmetic). The actual performance of the algorithm varies with the viewpoint and surface geometry. On an average it takes about 40-70 milliseconds to decompose one bicubic patch into non-overlapping regions. The main goal of this paper is to present an algorithm for reducing the hidden surface removal problem for spline surfaces to that of polygonal models. Coupled with such an algorithm for polygonal models, we obtain a complete visible surface extraction algorithm for spline surfaces. A preliminary version of this paper appeared in [KM98].

The rest of the paper is organized in the following manner. Section 2 presents background material and reviews algorithms from computational geometry and numerical linear algebra used in the rest of the paper. Section 3 briefly describes an overview of our algorithm. We outline an efficient algorithm for computing the silhouettes based on marching methods in Section 4. We introduce the notion of visibility curves in Section 5 and show that silhouettes and visibility curves partition a general surface into non-overlapping regions. We present algorithms and implementations for computation of visibility curves in Section 6. Section 7 talks about how to apply our decomposition algorithm to accomplish hidden surface removal and gives some details about our implementation. For the sake of completeness, we also present a variation of an existing algorithm [Mul89] for hidden surface removal of our decomposed faces in the Appendix.

2 Background

The overall algorithm for visibility computation uses algorithms from computational geometry and linear algebra. Some of them include trapezoidation of polygons, partitioning a simple polygon using non-intersecting chains, curve/surface intersections and local methods of tracing based on power iterations. We review some of these techniques in this section. Some of the algorithms presented here might not be optimal in terms of worst case asymptotic complexity; we compromised it in favor of simplicity and ease of implementation.

2.1 Triangulating Simple Polygons

We represent trimmed surfaces as well as portions of surfaces obtained after visibility decomposition using non-convex simple polygons in the domain. These non-convex polygonal domains are decomposed into triangles for many geometric operations like intersections and partitioning.

To decompose a simple polygon into an optimal number of triangles we use Seidel's algorithm [Sei91]. It is an incremental randomized algorithm whose expected complexity is $O(N \log^* N)$, where N is the number of vertices (in our application, N is typically between 100 and 200) . In practice, it is almost linear time. The algorithm proceeds in three steps as shown in Figure 2. They are:

- Decompose the polygon into trapezoids in $O(N \log^* N)$ time (Figure 2(a)),
- Generate monotone polygons from the trapezoids in linear time (Figure 2(b)), and
- Triangulate the monotone polygons in linear time (Figure 2(c)).

The trapezoidation of the polygon is useful in two ways. We can find whether a given point is inside the polygon in $O(\log N)$ time by doing binary search on the trapezoids. Moreover, we can obtain a point inside the polygon in constant time. For the purposes of the visibility computation algorithm, trapezoidation is sufficient. However, the triangulation is eventually used for rendering the visible portions. While there are linear time algorithms to triangulate simple polygons, we chose Seidel's algorithm because there was an efficient in-house implementation of the algorithm available to us [NM95].

2.2 Partitioning a Simple Polygon

We use algorithms for partitioning a simple polygon into connected components based on a set of non-intersecting chains. Given a simple polygon P and a number of non-intersecting polygonal chains C, our task is to partition P. We make no assumptions on C except that each chain $c_i \in C$ in itself should partition P. This algorithm is an important component of our overall algorithm. We use it to partition the domain of parametric patches using silhouette and visibility curves. The goal of this algorithm is to subdivide the original domain of a patch into components each of which have a simple boundary (no interior loops). We make use of this fact later in the paper. The problem at hand has been studied extensively in the computational geometry literature and has been solved efficiently (namely, Bentley-Ottmann [BO79] $O((n+k)\log n)$ or Chazelle-Edelsbrunner [CE88]

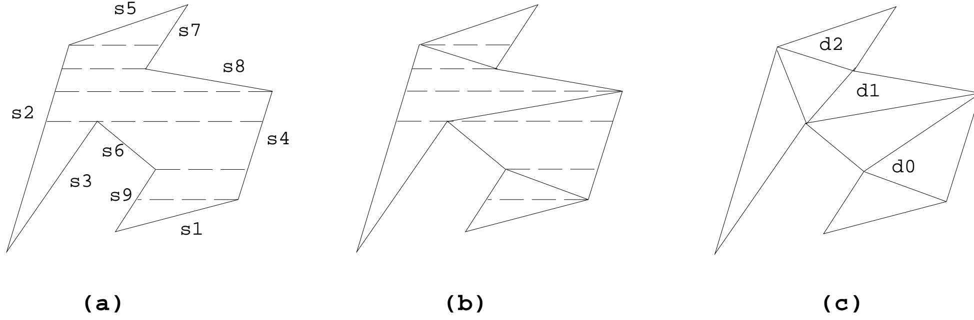


Figure 2: Three stages of Seidel's algorithm

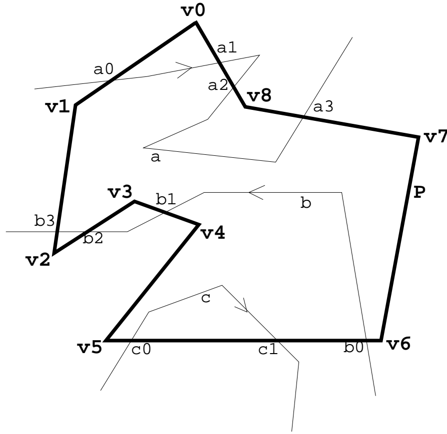


Figure 3: Partitioning a polygon with polygonal chains

$O(n \log n + k)$). Typically, these algorithms compute the arrangement of the line segments and then generate the partitions by walking the faces of the arrangement. We have a implemented a slightly modified version. A feature of our implementation is that along with the partitions, we also obtain the connectivity between them. This is used to verify the correctness of our results.

Let us assume for the sake of simplicity that the chains do not form a loop inside the polygon. This is because partitions surrounding the loop will not have a simple boundary anymore. We treat this case separately in our algorithm. Details are provided at the end of the algorithm description. The main idea in this algorithm is the fact that since the chains are non-intersecting, each of the partitioned region starts and ends at intersection points (with the polygon) of the same chain. Figure 3 shows a simple polygon P and three non-intersecting chains a, b and c. Since the vertices of each chain are given in a specific order, we shall assume that to be the direction of the chain. The algorithm works in two steps.

- Find all the intersection points of each chain with the polygon and number them according to the order in which they occur. We associate three fields with each intersection point - the chain corresponding to each intersection point (*type*), the number of the intersection point within the chain (*rank*), and whether the chain was coming in or going out of the polygon at this point (*in_or_out*). For example, the intersection point a_1 in the figure has $type = a$, $rank = 1$ and $in_or_out = out$ as its three fields. The *in_or_out* field is actually unnecessary because *rank* has that information. However, we use it for ease of description.

- Given all the intersections, we traverse the polygon starting from an arbitrary vertex. We use a stack as a data structure to compute the partitions. Let us assume that we start traversing the polygon from vertex v_0 in an anticlockwise order for the example given in the figure. Given this traversal, we can order the intersection points around the polygon. In the example, the order would be a_0, b_3, b_2, b_1, c_0, c_1, b_0, a_3, a_2, a_1. As we proceed from vertex v_i to v_{i+1} in the polygon, we retrieve all the intersection points of the various chains with the edge (v_i, v_{i+1}) in order. If q is an intersection point, and let p be the point on the top of the stack. To determine if q is *pushed* or p is *popped*, the following condition is checked.

```
if (q.type ≠ p.type)
   push(q);
else if ((q.rank - p.rank == 1) &&
         (q.in_or_out == out) &&
         (p.in_or_out == in))
   pop(p);
else if ((p.rank - q.rank == 1) &&
         (p.in_or_out == out) &&
         (q.in_or_out == in))
   pop(p);
else push(q);
```

If p is popped, then p and q form a partition of the polygon. The corresponding partition is read out and the chain of vertices between p and q is appended to the vertex (this chain will be a part of the partition that involves this vertex) currently on top of the stack (the one that was below p). After traversing the polygon completely once, we would have obtained all the partitions.

At this point, we have partitioned the domain using chains that end at the boundary of the domain. However, any loop that is present inside the domain must lie inside one of the partitioned regions. Each of the loops (starting from the innermost if the loops are nested) themselves form a partition. The remaining part of the region (they have boundaries with multiple components) are broken into simple regions by introducing a simple horizontal cut from the loop to the boundary of the partition or the next outer loop. The horizontal cut is made by choosing a point whose y-coordinate lies between the y extents of the loop and drawing a horizontal through the point.

2.3 Curve/Surface Intersection

Computing the intersection of curves and surfaces is needed to find whether a given surface is occluded. Given a surface patch that is guaranteed to have the same visibility for all its points, we shoot a ray from a point on the surface to the viewpoint and determine if the ray intersects any other surface (between the chosen point on the surface and the viewpoint). If the number of intersections is 0, the surface is visible, otherwise it is not. We use some recent algorithms for these intersections based on eigenvalue computations [MD94]. The algorithm we describe here can be used for any degree curve and its complexity is cubic in the degree of the curve. Therefore, even though the problem of ray-surface intersection is much simpler, our algorithm does not suffer because of its generality.

Given a parametric representation of a surface $\mathbf{F}(s,t)$ ($\mathbf{F}: \mathcal{R}^2 \rightarrow \mathcal{R}^3$) of degree $m \times n$, we compute its implicit representation using resultant methods [Dix08] and obtain a matrix formulation $\mathbf{M}(x,y,z,w)$. The entries of this matrix are linear polynomials in x,y,z,w (of the form $ax+by+cz+dw$) so that the set

$$\{(x,y,z,w) : Det(\mathbf{M}(x,y,z,w)) = 0\} \qquad (1)$$

is exactly the surface in homogeneous coordinates. One main advantage of this method is that we do not have to compute the large determinant which is highly unstable numerically. We substitute the parameterization of the curve, say $\mathbf{G}(u) = (\overline{X}(u), \overline{Y}(u), \overline{Z}(u), \overline{W}(u))$ of degree d, into $\mathbf{M}(x,y,z,w)$ and obtain a univariate matrix polynomial $\mathbf{M}(u)$. The problem of intersection computation is thus reduced to computing the roots of the non-linear matrix polynomial $\mathbf{M}(u)$. The algorithms that are used to compute the roots of a matrix polynomial require a power basis representation. However, our polynomial which is in the Bernstein basis is easily converted to power basis by the transformation $\overline{u} = \frac{u}{1-u}$. The resulting matrix $\mathbf{M}(\overline{u})$ can be represented as

$$\mathbf{M}(\overline{u}) = \overline{u}^d M_d + \overline{u}^{d-1} M_{d-1} + \ldots + \overline{u} M_1 + M_0. \qquad (2)$$

where M_i's are matrices of order $2mn$ with numeric entries. Furthermore, the roots of the matrix polynomial, $\mathbf{M}(u)$, are identical with the eigenvalues of

$$C = \begin{bmatrix} 0 & I_{2mn} & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & I_{2mn} \\ -\overline{\mathbf{M}}_0 & -\overline{\mathbf{M}}_1 & -\overline{\mathbf{M}}_2 & \ldots & -\overline{\mathbf{M}}_{d-1} \end{bmatrix} \qquad (3)$$

where $\overline{\mathbf{M}}_i = \mathbf{M}_d^{-1}\mathbf{M}_i$ [GLR82]. In case $\mathbf{M}_d$ is singular or ill-conditioned, the intersection problem is reduced to a generalized eigenvalue problem [MD94]. Algorithms to compute all the eigenvalues are based on QR orthogonal transformations [GL89].

2.4 Power Iterations

We use marching methods to trace the visibility curves (see Section 5). At each iteration, we pose the problem as an eigenvalue problem and use local methods to compute points on the curve. Power iteration is a fundamental local technique used to compute eigenvalues and eigenvectors of a matrix. Given a diagonalizable matrix, $\mathbf{A}$, there exists an orthonormal matric $\mathbf{X}(= [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n])$ such that $\mathbf{X}^{-1}\mathbf{A}\mathbf{X} = Diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$ and $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$. Given a randomly chosen unit vector $\mathbf{q}_0$, the *power method* produces a sequence of vector $\mathbf{q}_k$ as follows:

$$\mathbf{z}_k = \mathbf{A}\mathbf{q}_{k-1}; \quad \mathbf{q}_k = \mathbf{z}_k / \parallel \mathbf{z}_k \parallel_\infty; \quad s_k = \mathbf{q}_k^T \mathbf{A} \mathbf{q}_k;$$

where $\parallel \mathbf{z}_k \parallel_\infty$ refers to the infinity norm of the vector $\mathbf{z}_k$. s_k converges to the largest eigenvalue λ_1 and $\mathbf{q}_k$ converges to the corresponding eigenvector $\mathbf{x}_1$.

The basic idea of power iterations can be used and modified to obtain the eigenvalue of a matrix A that is closest to a given guess s. It actually corresponds to the largest eigenvalue of the matrix $(\mathbf{A} - s\mathbf{I})^{-1}$. Instead of computing the inverse explicitly (which can be numerically unstable), we use *inverse power* iterations. Given an initial unit vector $\mathbf{q}_0$, we generate a sequence of vectors $\mathbf{q}_k$ as

$$Solve\ (\mathbf{A} - s\mathbf{I})\mathbf{z}_k = \mathbf{q}_{k-1}; \quad \mathbf{q}_k = \mathbf{z}_k / \parallel \mathbf{z}_k \parallel_\infty; \quad s_k = \mathbf{q}_k^T \mathbf{A} \mathbf{q}_k;$$

We use inverse power iterations to trace curves. We formulate the curve as the singular set of a matrix polynomial and reduce it to an eigenvalue problem. Given a point on the curve, we approximate the next point on the curve by taking a small step-size in a direction determined by the local geometry of the curve. We use this

point as our guess and use inverse power iterations to converge back to the curve. The reader is requested to refer to [KM97] to get more details on how we perform curve tracing using inverse power iterations.

3 Overview of the Algorithm

In this section, we briefly describe the algorithm. The details of individual steps of the algorithm are described in later sections. For the purposes of this paper, we assume that the viewpoint is situated at $z = -\infty$ so that the projection on the viewing plane is orthographic. Given a scene composed of non-intersecting Bézier patches and a viewpoint, we perform the following steps for each patch.

- Compute the *silhouette* curves on the patch for the given viewpoint.
- Partition the domain of the Bézier patch as determined by the silhouette curves. The boundary of each partition is made up of the original boundary curves or the computed silhouette curves.
- For each partition so generated,
  - Trace out *visibility* curves (curves in the interior of the partition that have the same projection on the image plane as that of the boundary curves) by following the boundary curves.
  - Partition the domain according to the visibility curves

The set of partitions obtained after executing this algorithm satisfy the property that each partition is non-self-occluding. This fact is proved later in Theorem 2. The main complexity of the algorithm lies in the computation of the silhouette and visibility curves. We use a combination of symbolic and numeric techniques to evaluate these curves explicitly.

4 Silhouettes

Silhouette computation forms an important part of visibility algorithms for curved surfaces. Intuitively speaking, a silhouette curve is the locus of all points on the surface where the normal vector to the surface at that point is perpendicular to the line of sight. We shall restrict our discussion to surfaces whose silhouette (from a given viewpoint) is a curve on the surface. The property of the silhouette curve is that it subdivides the surface into front and back facing regions. However, as shown in Figure 1, silhouettes alone are not sufficient to determine all visible regions. In this section, we describe an algorithm to compute the silhouette curve on a parametric (represented as Bézier [Far93]) patch efficiently. Some assumptions are in order about the kind of surfaces we deal with. We assume that the surfaces are not self-intersecting purely for exposition. In the most general case, we can compute the intersection curve using the algorithm described in [KM95] and partition the surface into non-intersecting pieces.

We assume for the sake of simplicity that the viewpoint is located at $(0, 0, -\infty)$ and that the view direction is towards positive z-axis. It is easy to see that even if this is not the case, one can always achieve it by applying an appropriate perspective transformation to the control points of the parametric surface $\mathbf{F}(s, t)$. We also require that all the surfaces are differentiable everywhere. We formulate the silhouette curve as an algebraic plane curve in the domain of $\mathbf{F}(s, t)$.

4.1 Formulation of the Silhouette Curve

Let $\mathbf{F}(s, t)$ denote the parametric (differentiable) surface and let $\phi_1(s, t), \phi_2(s, t)$ and $\phi_3(s, t)$ denote the mappings from the parametric space to (x, y, z) space.

$$\mathbf{F}(s, t) = \langle X(s, t), Y(s, t), Z(s, t), W(s, t)\rangle$$

$$\phi_1(s, t) = \frac{X(s, t)}{W(s, t)}, \ \ \phi_2(s, t) = \frac{Y(s, t)}{W(s, t)}, \ \ \phi_3(s, t) = \frac{Z(s, t)}{W(s, t)}$$

In the rest of the paper, we shall drop the (s, t) suffixes from all the functions for more concise notation. The $z-$component of the normal at an arbitrary point on the surface is given by the determinant

$$N_z = \begin{vmatrix} \phi_{1_s} & \phi_{1_t} \\ \phi_{2_s} & \phi_{2_t} \end{vmatrix} \tag{4}$$

where ϕ_{i_s} and ϕ_{i_t} denote the partial derivatives of the appropriate function ϕ_i with respect to s and t.

$$\phi_{1_s} = \frac{(WX_s - W_sX)}{W^2}$$

$$\phi_{1_t} = \frac{(WX_t - W_tX)}{W^2}$$

$$\phi_{2_s} = \frac{(WY_s - W_sY)}{W^2}$$

$$\phi_{2_t} = \frac{(WY_t - W_tY)}{W^2}$$

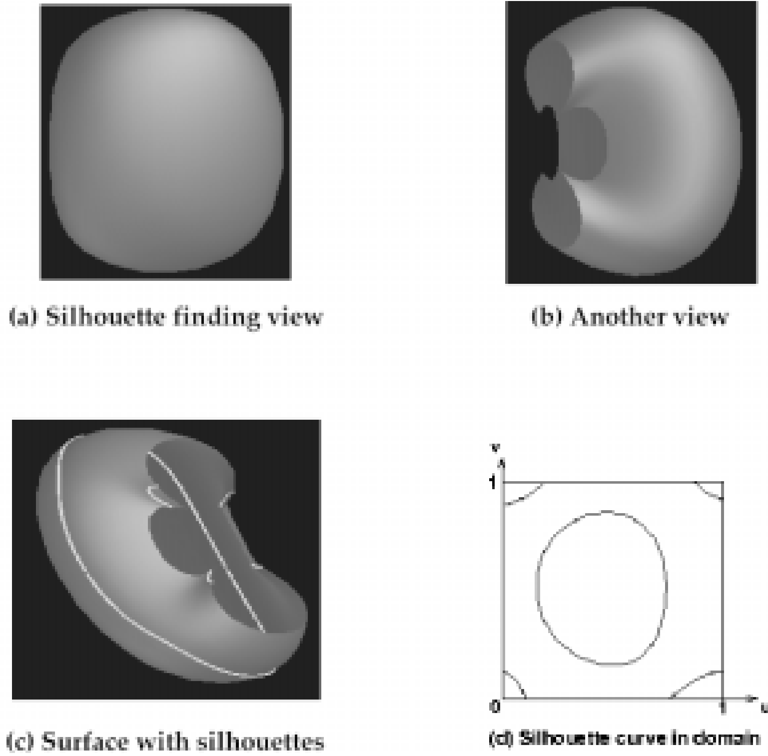
On the silhouette curve, $N_z = 0$. Since $W(s, t) > 0$, we can express the plane curve representing the silhouette as the determinant

$$N_z = \begin{vmatrix} (WX_s - W_sX) & (WX_t - W_tX) \\ (WY_s - W_sY) & (WY_t - W_tY) \end{vmatrix} = 0 \tag{5}$$

Expanding the determinant and rearranging the terms, we can express it as the singular set of the matrix $\mathbf{M}(s, t)$

$$\mathbf{M}(s, t) = \begin{pmatrix} X(s, t) & Y(s, t) & W(s, t) \\ X_s(s, t) & Y_s(s, t) & Z_s(s, t) \\ X_t(s, t) & Y_t(s, t) & Z_t(s, t) \end{pmatrix} = 0 \tag{6}$$

The singular set of $\mathbf{M}(s, t)$ are the values of s and t which make it singular.



(a) Silhouette finding view (b) Another view

(c) Surface with silhouettes (d) Silhouette curve in domain

Figure 4: Loops in silhouette curves

There is a special case here that needs to be addressed here. Earlier, we said that the silhouette curve partitions the surface into front ($N_z < 0$) and back-facing ($N_z > 0$) regions. However, there could be cases when N_z starts positive, touches zero and then becomes positive again. Saddle-shaped regions (oriented appropriately) are examples of these cases. These are not the generic case and small perturbations of the view direction is enough to remove these cases [RE93]. Further, the algebraic formulation of the silhouette curve above does not preclude the possibility of singularities (self-intersections). We believe that singularities of silhouettes (in object space) are also non-generic and a similar perturbation of the view direction should remove them. However, we are not able to prove it or provide a good reference for it. For the rest of this paper, we will assume that the silhouette curve does not contain singular points.

4.2 Silhouette Computation

Let us denote the projected silhouette curve corresponding to $\mathrm{Det}(\mathbf{M}(s,t))$ by $D(s,t)$. If the Bézier patch $\mathbf{F}(s,t)$ is of degree m in s and n in t, the curve $D(s,t)$ has degree at most $3(m+n)$. This is a high degree algebraic curve. Our task is to evaluate this curve (*i.e.,* its topological type) completely and efficiently inside our domain of interest.

Our approach is based on marching along the curve using local geometric properties of the curve. All marching methods require at least one point on every component of the curve inside the domain of interest. We adopt different methods to compute starting points on *open* (intersect the boundary of the domain) and *closed* components (or loops).

To determine starting points on open components we substitute one of the variables s or t with the value 0 or 1. This reduces the silhouette equation to a polynomial equation in one variable and this has only a discrete set of solutions. We find all the *boundary silhouette points* by determining the roots of four univariate matrix polynomials, $\mathbf{M}(0,t)$, $\mathbf{M}(1,t)$, $\mathbf{M}(s,0)$ and $\mathbf{M}(s,1)$. This problem can be reduced to finding the eigenvalues of an associated companion matrix (see Eqn. (3)) [MD94]. We retain only the real solutions that lie within the domain.

A much harder problem is to determine if the silhouette curve has loops inside the domain of the surface, and if so to compute at least one point on each of them. We use the fact that the silhouette curve is an algebraic plane curve that is continuous in the complex domain. Since the coefficients of the curve are real, all complex portions of the curve must occur in conjugate

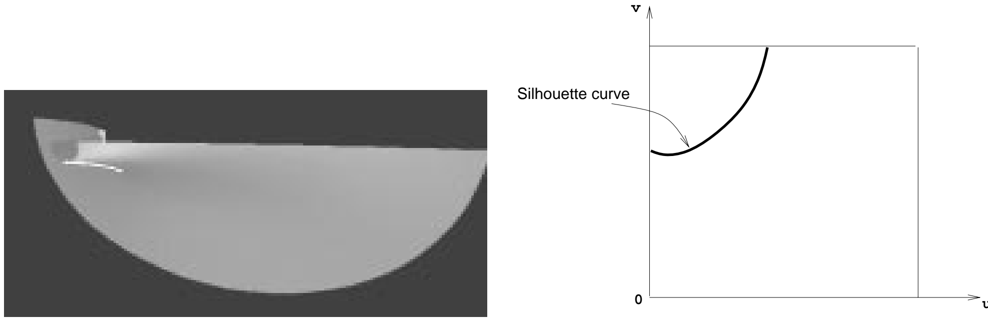


Figure 5: Silhouette curve on the patch and the domain

pairs. We characterize certain special points on loops (turning points) as places where two complex conjugate paths merge to form a real component. Our idea of loop detection is captured by the following lemma which we state without proof.

Lemma 1 *[KM97] If the curve in the real domain* $[0,1]\times[0,1]$ *consists of a closed component, then two arbitrary complex conjugate paths meet at one of the real points (corresponding to a turning point) on the loop.*

By following all the complex paths inside the domain we can locate at least a single point on each loop. Figure 4 shows the presence of loops in silhouette curves.

Given a point on each component of the silhouette curve, marching methods obtain approximations of the next point by taking a small step size in a direction determined by the local geometry of the curve. Based on the approximate value, these algorithms use local iterative methods to trace back on to the curve to evaluate the silhouette curve. We have developed an algorithm based on inverse power iterations (section 2.4) to *trace* the curve. The details of the complete algorithm are presented in [KM97]. Our algorithm evaluates the silhouette curve at discrete steps to create a piecewise linear approximation. The tracing algorithm has been implemented and tested on a variety of examples and has proved to be fairly robust.

Consider a patch $\mathbf{F}(s,t)$ of degree m in s and n in t, and let $N = \max\{m,n\}$. Then the companion matrix (Eqn. (3)), whose eigenvalues correspond to the roots of the matrix polynomial) is of order $3N$. We use the QR method [GL89] to compute all the eigenvalues. The number of floating point operations performed is $8N^3$. In inverse power iterations, the two main operations are the LU decomposition of the matrix and solving the resulting triangular systems. Usually the LU decomposition is computed using Gaussian elimination and it takes about $\frac{1}{3}N^3$ operations (without pivoting). Solving each triangular system costs about $\frac{1}{2}N^2$ operations. As a result, the inverse power iteration takes about $\frac{1}{3}N^3+\frac{k}{2}N^2$ operations, where k is the number of iterations. It turns out that the structure of the companion matrix can be used to reduce the number of operations for LU decomposition ($9N$ operations) as well as solving the triangular systems ($12kN$ operations, where k is the number of iterations).

4.3 Surface Partitioning based on Silhouettes

Figure 5 shows the silhouette curve on an example patch along with the curve on its domain. It is clear from the figure that the silhouette divides the patch into front and back facing regions.

Lemma 2 *Let* $\mathbf{C} \subset \Re^3$ *be the set of silhouette curves on a given surface* $\mathbf{S} \subseteq \Re^3$ *from a given viewpoint. Then* $\mathbf{S}$ *can be decomposed into* $\mathbf{C}$ *and disjoint regions whose boundaries are surface boundaries and/or parts of* $\mathbf{C}$.

Proof: Let p be a point on $\mathbf{S}-\mathbf{C}$ (*i.e.,* $N_z(p) \neq 0$). Define the (open) region $\mathbf{R}_p$ as follows:

$$\mathbf{R}_p = \{s \in \mathbf{S}-\mathbf{C} \mid \exists \textit{ a continuous path from } p \textit{ to } s \textit{ without crossing } \mathbf{C}\}$$

$\mathbf{R}_q$ is defined similarly for a point $q \notin \mathbf{R}_p$, if such a q exists. Now we have to show that $\mathbf{R}_p \cap \mathbf{R}_q \neq \emptyset$. Let us assume the contrary. Then there exists a point $r \in \mathbf{R}_p \cap \mathbf{R}_q$. This implies that there is a continuous path (without crossing $\mathbf{C}$) from p to r and r to q, and hence from p to q. This is a contradiction.

□

From the above lemma we can conclude that $\mathbf{S}$ can be partitioned into a set of regions $\mathbf{R}$ such that the boundaries of each region $\wp \in \mathbf{R}$ consists of original surface boundaries and silhouette curves. The domain of each patch is represented as a simple polygon in counterclockwise order. For example, a complete tensor product Bézier patch has its domain polygon as $\{(0,0),(1,0),(1,1),(0,1),(0,0)\}$. We then use our *partition_polygon* (Section 2.2) routine to subdivide the domain into disjoint regions.

We now state without proof a fundamental theorem from vector calculus called the *global inverse theorem* which provides the basis for our method [Ful78].

Theorem 1 *Let* $\mathbf{F}$ *be a continuously differentiable mapping defined on an open region* $D \in \Re^2$*, with range* $R \in \Re^2$*, and let its Jacobian be never zero in* D*. Suppose further that* C *is a simple closed curve that, together with its interior, lies in* D*, and that* $\mathbf{F}$ *is one-to-one on* C*. Then the image* Γ *of* C *is a simple closed curve that, together with its interior, lies in* R*. Furthermore,* $\mathbf{F}$ *is one-to-one on the closed region consisting of* C *and its interior, so that the inverse transformation can be defined on the closed region consisting of* Γ *and its interior.*

□

The importance of this theorem lies in the fact that properties of the entire region can be argued by looking at the properties of its boundary.

Consider the vector field $\mathbf{M} : \Re^2 \rightarrow \Re^2$ such that

$$M(s,t) = (\phi_1(s,t), \phi_2(s,t))$$

where the $\phi_i's$ were defined in the previous section. Intuitively, the vector field M *projects* a point on the surface $\mathbf{F}(s,t)$ on to the $xy-$plane. We now relate the silhouette curve and the Jacobian of the function M.

Lemma 3 *Let* M *be a mapping from* $\Re^2 \rightarrow \Re^2$ *such that* $M(s,t) = \left(\frac{X(s,t)}{W(s,t)}, \frac{Y(s,t)}{W(s,t)}\right)$*. Then the loci of all points on the surface that have vanishing Jacobians corresponds exactly to the silhouette curve.*

Proof: The Jacobian $J(s,t)$ of M is given by

$$J(s,t) = \begin{vmatrix} \frac{(WX_s - W_sX)}{W^2} & \frac{(WX_t - W_tX)}{W^2} \\ \frac{(WY_s - W_sY)}{W^2} & \frac{(WY_t - W_tY)}{W^2} \end{vmatrix} \qquad (7)$$

However when $J = 0$, we get the same equation as Eqn. (5). That is precisely the equation of the silhouette curve in the domain of the surface. Therefore, the silhouette curves are the only places on the surface where the Jacobian of M vanishes.

□

We denote the set of regions obtained after partitioning by R. Henceforth, we shall be considering a single element of this set $\wp \in R$. Because of the previous lemma, we can conclude that the interior of each $\wp \in R$ have no vanishing Jacobian. Therefore, *global inverse theorem* applies on each of the regions in R.

5 Visibility Curves

In the previous section, we described a method to compute silhouettes. We shall now introduce the notion of *visibility curves* and elucidate their role in determining visibility.

Consider a region $\wp \in R$. Let us denote the boundary of $\wp$ by $\partial\wp$ and the interior of $\wp$, the open set $\wp - \partial\wp$, by $int(\wp)$.

Definition 1 *Given a region* $\wp \in R$*, the visibility curves on* $\wp$*,* $V(\wp)$*, is defined as the locus of points,*

$$V(\wp) = \{p_i | p_i \in int(\wp), \exists p_b \in \partial\wp, M(p_i) = M(p_b)\}.$$

Figure 6: A helical patch with no silhouettes

Intuitively speaking, visibility curves are the loci of all points which lie in the interior of such a partition that have the same projection on the viewing plane as the boundary curves of that partition. It is precisely at these points that the visibility of a patch changes.

Lemma 4 *The visibility curves on* $\wp$*,* $V(\wp)$ *as defined above, indeed, form a set of curves.*

Proof: We shall prove this part by contradiction. Let us assume that there exists an isolated point $p_i \in V(\wp)$. But by definition, there must be a point $p_b \in \partial\wp$ such that $M(p_i) = M(p_b)$. We know that $\partial\wp$ is a continuous closed curve. Consider a small displacement, $\delta \in \Re^2$, of p_b along $\partial\wp$. Since M is a smooth map, there exists an $\epsilon \in \Re^2$ such that $M(p_i + \epsilon) = M(p_b + \delta)$ [Mun75]. In fact, in the limit δ going to zero, $\epsilon \approx J^{-1}(p_i)J(p_b)\delta$, where J and J^{-1} are the Jacobian and the Jacobian inverse of M respectively. Further, we know that $J^{-1}(p_i)$ exists because $p_i \in int(\wp)$. Therefore, p_i cannot be an isolated point, and the visibility curves indeed form a curve.

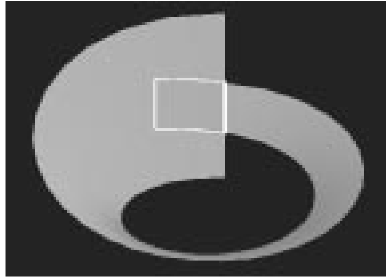
□

Corollary 1 *The set of visibility curves* $V(\wp)$*, induces a partition on the region* $\wp$*.*

Proof: Follows from Lemma 2.

□

Figure 6 shows a helical patch with no silhouettes. Figure 7 shows the visibility curves computed on the same patch. The visibility curves are also shown in the domain of the patch. The patch in Figure 6 is deliberately transformed to provide a better view. Lemma 4 provides a constructive method to find all the visibility curves. Given the boundary curves of each region, we have to determine all the intersections among the various curves comprising the boundary projections (including self-intersections). We are assuming that the boundary



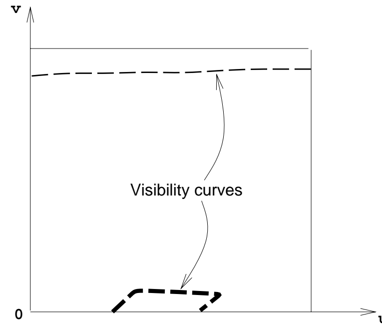


Figure 7: Visibility curves on the helical patch

curves are in general position so that their projections intersect only in discrete points. These points form the endpoints for all the visibility curves inside the region $\wp$. Once all the endpoints are found, we use marching methods to trace all the curves. Details of the method are discussed in the next section.

Given $\wp \in R$, we construct a partition of $\wp$ induced by the visibility curves using the *partition_polygon* routine. Let us denote the set of regions obtained after partitioning by $K_\wp$. If $k \in K_\wp$, ∂k denotes the boundary of the region k and $int(k)$, $k - \partial k$, is the interior of k. Figure 8(a) shows the partition of the helical patch based on the visibility curves. In Figure 8(b), we see the image of $K2$ under M. It is clear that the boundary of this region is not one-to-one (as seen in the case of points a and a'). However, in the next theorem, we will argue that the the interior of each region $k \in K_\wp$ is one-to-one under the map M.

Theorem 2 *Let $k \in K_\wp$. Then $int(k)$ is one-to-one under the projection mapping M.*

Proof: The main idea of the proof is to show that any simple closed curve C in $int(k)$ is one-to-one under M. Since the Jacobian never vanishes in the interior of each region, we use the *global inverse theorem* to conclude that the entire region on and inside the closed curve C is one-to-one. Since choice of C is arbitrary, we can conclude that $int(k)$ is one-to-one. We would like to reiterate our earlier assumption that *the boundary curves are in general position so that their projections intersect only in discrete points.*

Consider an arbitrary curve $C \in int(k)$. Assume that C is not one-to-one under the projection mapping M. Therefore, there exists at least two points p_1 and p_2 such that $M(p_1) = M(p_2)$. Consider a simple path (say $path_1$) from p_1 to a boundary point p_{b1}. Since M is a continuous map, there must be a corresponding path ($path_2$) from p_2. Let us assume that $path_1$ reaches the boundary point p_{b1} first, and that the point on $path_2$ that has the same projection as p_{b1} is p_2'. Since $M(p_{b1}) = M(p_2')$, p_2' cannot be an interior point (otherwise, it is part of the visibility curve). So p_2' is also a boundary point. However, since p_{b1} was an arbitrarily chosen point, all its choices must lead to p_2's on the boundary. But this contradicts our assumption about boundary curves being in general position. Hence any curve C must be one-to-one under M.

Using C and the *global inverse theorem*, we conclude that the interior of every region $k \in K_\wp$ is one-to-one under the projection mapping M.

□

6 Computation of Visibility Curves

In this section, we will describe our method to compute visibility curves. The whole algorithm is split into two parts - (i) finding all the intersections on the projected boundary curves, and (ii) tracing each visibility curve.

6.1 Boundary Intersections

After partitioning based on silhouettes, we obtain regions whose boundaries consist of parts of the original boundary curves of the patch and the silhouette. Let us consider a single region whose boundary is made up of a set of original boundary curves, B, and another set of silhouette curves, L. Each element of B is represented by the corresponding Bézier curve and the interval of parameter values in which it is valid. We also maintain the projection of the boundary curves as a polygonal chain in order to obtain its intersections with silhouette curves. Each element of L and its projection under M is maintained just as a polygonal chain. In order to compute all the intersection points on the projection, we must detect all self-intersections in each element of B and L and intersections between elements. Overall, there are only four basic categories in which all of them fall. We shall discuss each one in detail.

1. **Intersection between two boundary curves:** Basically, this case reduces to finding the intersection points between two Bézier plane curves. Let $\mathbf{f}$ and $\mathbf{g}$ be two plane curves parametrized by u and

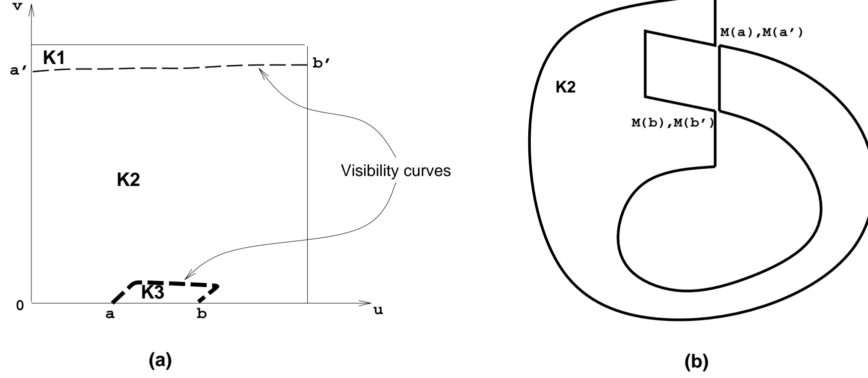


Figure 8: (a) Partitioning based on visibility curves (b) Image of region K2 under M

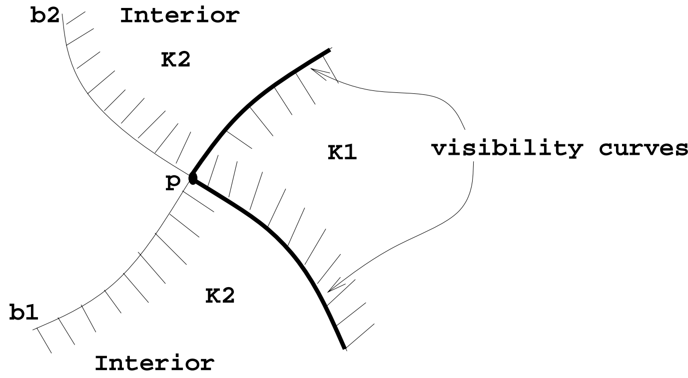


Figure 9: Generation of visibility curves near the boundaries

v respectively. The two equations that give rise to the solution are

$$\frac{X_f(u)}{W_f(u)} = \frac{X_g(v)}{W_g(v)}$$

$$\frac{Y_f(u)}{W_f(u)} = \frac{Y_g(v)}{W_g(v)}$$

where X_f, Y_f, W_f and X_g, Y_g, W_g are the component-wise functions of $\mathbf{f}$ and $\mathbf{g}$ respectively. By eliminating u from these two equations using Sylvester's resultant [Sed83], we obtain a matrix polynomial in v. We can reduce it to an eigenvalue problem of an associated companion matrix [MD94]. After obtaining all the eigenvalues (using LAPACK routines) only the solutions that lie within the intervals are taken. Using this method, all the intersection points are determined accurately and efficiently.

2. **Self-intersections on boundary curves:** Consider a plane Bézier curve $\mathbf{f} = \langle X(s), Y(s), W(s)\rangle$ of degree n. This curve self-intersects if there exist parameter values u and v, $u \neq v$, such that

$$\frac{X(u)}{W(u)} = \frac{X(v)}{W(v)}$$

$$\frac{Y(u)}{W(u)} = \frac{Y(v)}{W(v)}.$$

Since $u = v$ is a trivial solution to the above pair of equations, we eliminate it by dividing each of the equations by the factor $(u-v)$. Thus the equations become

$$\frac{(X(u)W(v) - X(v)W(u))}{(u-v)} = 0$$

$$\frac{(Y(u)W(v) - Y(v)W(u))}{(u-v)} = 0$$

These are two equations each of degree $n-1$ in u and v. By eliminating u from these equations using Sylvester's resultant, we get a $(2n-2) \times (2n-2)$ matrix polynomial of degree $n-1$. We reduce this problem to one of finding eigenvalues of an associated matrix of size $2(n-1)^2$. This gives all the self-intersections on the boundary curve.

3. **Intersection between two silhouette curves:** The equations governing the solution set are

$$D(s,t) = 0$$

$$D(u,v) = 0$$

$$\frac{X(s,t)}{W(s,t)} = \frac{X(u,v)}{W(u,v)}$$

$$\frac{Y(s,t)}{W(s,t)} = \frac{Y(u,v)}{W(u,v)}$$

where $D(s,t) = 0$ is the equation of the silhouette curve defined in Section 4.2 and X, Y, Z and W are the component-wise functions of the patch. These four equations in four variables, typically,

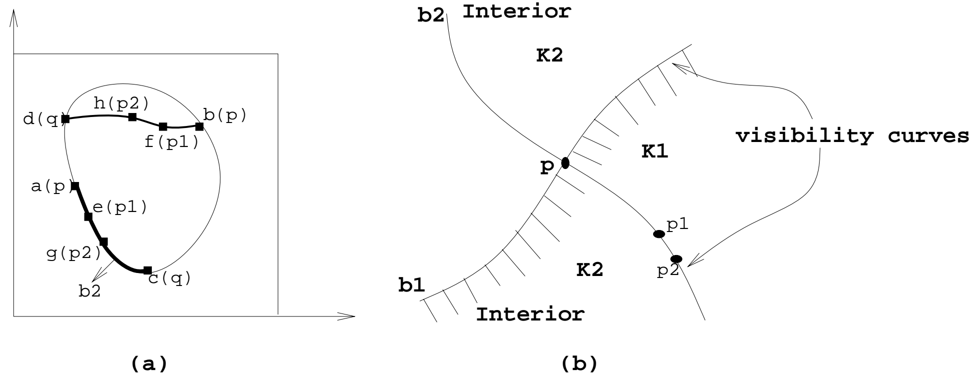


Figure 10: Tracing of visibility curves

give rise to a zero dimensional solution set. However, it is not feasible to solve these four equations directly because of the high degree of the silhouette curve and the algebraic complexity of the resulting system. Therefore, we use piecewise linear approximation of all silhouette curves as the first iteration in our intersection computation. We treat the chain as a set of line segments. Each intersection point obtained is then refined using local minimization methods (we use Powell's method) using the energy equation $E(u,v,s,t)$ given by

$$E(u,v,s,t) = D^2(s,t) \; + \; D^2(u,v) \; + \\ (X(s,t)W(u,v) - X(u,v)W(s,t))^2 \; + \\ (Y(s,t)W(u,v) - Y(u,v)W(s,t))^2.$$

Self-intersections are also found using the same method. We found that in practice this method of finding intersections works well and gives accurate results.

4. **Intersection between boundary curve and silhouette:** Consider a boundary curve **f** parametrized by u. X_f, Y_f, Z_f and W_f are the scalar functions of **F**. X_f, for example, could represent any one of the four functions, $X(s,0)$, $X(s,1)$, $X(0,t)$ and $X(1,t)$. The set of intersection points of this curve with the silhouette satisfy

$$D(s,t) = 0$$

$$\frac{X(s,t)}{W(s,t)} = \frac{X_f(u)}{W_f(u)}$$

$$\frac{Y(s,t)}{W(s,t)} = \frac{Y_f(u)}{W_f(u)}.$$

In this case, we use the approximated version of the boundary curves (as a piecewise linear chain) and apply a similar procedure as the previous case. The minimization equation in this case is

$$E(u,s,t) = D^2(s,t) \; + \\ (X(s,t)W_f(u) - X_f(u)W(s,t))^2 \; + \\ (Y(s,t)W_f(u) - Y_f(u)W(s,t))^2.$$

Accuracy of Silhouette curve: The use of a piecewise linear approximation for the silhouette curves may seem limiting considering that points on this curve are used as starting points for tracing visibility curves. However, the points are not chosen from the linear approximation. We maintain the linear chain only to compute a starting guess to the actual point on the silhouette curve quickly. Given this approximation, we refine it using the analytic form of the silhouette curve and the boundary curve (if given in analytic form). The refinement is carried out until the point is within a user specified tolerance to the actual silhouette curve.

Once all the intersection points are computed, we are ready to trace all the visibility curves.

6.2 Tracing Visibility Curves

Given the starting point of each visibility curve, we are ready to trace it.

Figure 10 shows the tracing of a visibility curve in the domain of a region and in projection space. Points a and b have the same projection point p. Let the curve $b2$ in Figure 10(b) correspond to the portion of the boundary from a to c (see Figure 10(a)). The boundary of the region on the domain is represented as a polygon (obtained after partitioning based on silhouettes). Let us assume that at an arbitrary step of the tracing method we are at point e on $b2$ and at f on the visibility curve. Both e and f have the same projection point $p1$. Let the domain coordinates of f be (f_u, f_v). If we move from e to g on $b2$, the point f must move to a neighboring point, h, on the visibility curve. If (g_u, g_v) are the domain coordinates of g, let $x = \frac{X(g_u,g_v)}{W(g_u,g_v)}$ and $y = \frac{Y(g_u,g_v)}{W(g_u,g_v)}$. X, Y and W are the component-wise functions of the original patch. We would like to find (h_u, h_v) in the local neighborhood of (f_u, f_v) such that

$$\frac{X(h_u,h_v)}{W(h_u,h_v)} - x = 0$$

$$\frac{Y(h_u,h_v)}{W(h_u,h_v)} - y = 0.$$

Eliminating h_v from these two equations results in a matrix polynomial in h_u. The singular set of this polynomial determines h_u and from the corresponding eigenvector , we can find h_v. However, a lot of unnecessary work can be avoided by observing that (h_u, h_v) is in the neighborhood of (f_u, f_v). Using f_u as a guess to h_u and building a corresponding eigenvector out of f_v, we perform inverse power iterations (described in Section 2.4) to obtain (h_u, h_v).

It is possible to use Newton's method to solve the above set of equations. However, as we will see, this method has some problems. Let us assume that point b on the boundary is on a silhouette. We proved that a silhouette point is one where the Jacobian vanishes. Therefore, Newton's method does not perform well close to silhouette points. Inverse power iteration suffers from no such problem.

Tracing terminates when one of the following two cases occur.

- (h_u, h_v) goes out of the region, or
- $(h_u, h_v) = (g_u, g_v)$.

The first case occurs more often. For example, if we trace further from point c, d goes out of the region. Therefore, after each step of the tracing process we have to check for containment of a point inside an arbitrary (simple, but not necessarily convex) polygon. This could be very expensive unless some processing is done on the polygon. We use Seidel's trapezoidation algorithm (created during triangulation) to create trapezoids by horizontal decomposition in $O(N)$ time. Any point can then be determined inside or outside in $O(\log N)$ time.

The second case occurs when the visibility curve hits the boundary curve and then continues along it. This will not be detected in the previous case, and hence, has to be checked explicitly.

After computing all the visibility curves, we calculate all the partitions induced by the visibility curves in each region using the *partition_polygon* routine. Figures 11 and 12 show the partitioning of the patches in Figure 1 and Figure 6 using the visibility curves. Each partition thus obtained is one-to-one under projection. It transforms the original visibility problem into one of n polygon-like surfaces in space. The analysis given so far partitions all parts of a given surface, while only the far regions need to be partitioned. A simple check before tracing visibility can reduce the time and space complexity of the algorithm.

7 Application to Hidden Surface Removal

In the previous sections, we looked at the visibility problem for a single patch. After partitioning each patch based on silhouettes and visibility curves, each region is one-to-one under the projection operation, and can now be treated as a polygon. We shall, therefore, refer to each such region as a *face*. Recently fast randomized algorithms have been developed that can handle this problem for polygonal models [Mul89]. We present

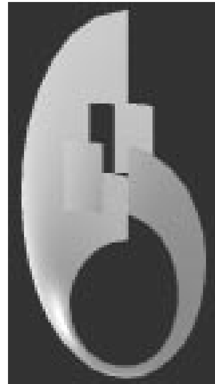


Figure 12: Partitioning of patch in Figure 6 based on visibility curves

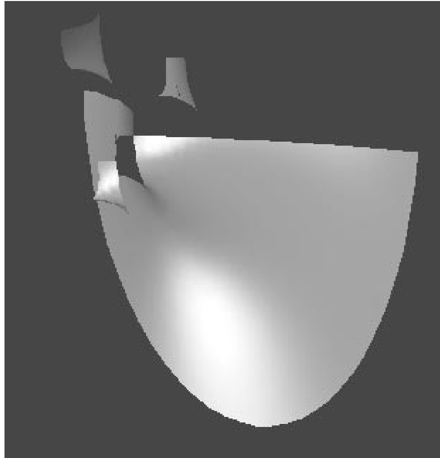
a slight variation of Mulmuley's algorithm for the sake of completeness in the Appendix section.

Our final goal is to output trimmed patches of the scene that are visible from the given viewpoint. We shall assume for simplicity that the faces input to this algorithm are non-intersecting. If they are intersecting, we may have to compute all the pairwise surface intersections [KM97] and split them into non-intersecting faces. It is possible that in many curved surface models (generated using constructive solid geometry or surface fitting algorithms), adjacent patches almost always share sections of boundary curves. This can cause problems when projecting one such curve into the domain of the other and computing curve-curve intersections. However, if any extra information about the adjacency between the various patches in the model is known, we can avoid the computation of such degenerate intersections. Many of the current modelers are capable of producing the adjacency information of such models.

7.1 Implementation and Performance

The algorithm to compute non-overlapping regions using silhouette and algebraic curves has been implemented. The algorithm uses existing EISPACK and LAPACK routines for some of the matrix computations. At each stage of the algorithm, we can compute bounds on the accuracy of the results obtained based on the accuracy and convergence of numerical methods adopted like eigenvalue computation, power iteration and Gaussian elimination. Our implementation uses EISPACK [GBDM77] routines (in Fortran) to compute the eigenvalues of matrices. The algorithm was run on a SGI Onyx workstation with a R4400 CPU with 128 Mbytes of main memory.

We have not implemented the randomized algorithm for performing the general hidden surface removal. Currently, our system takes a set of parametric patches and computes its decomposition into non-overlapping regions. Table 1 shows the performance of our algorithm on certain parametric patches. The time shown for curve



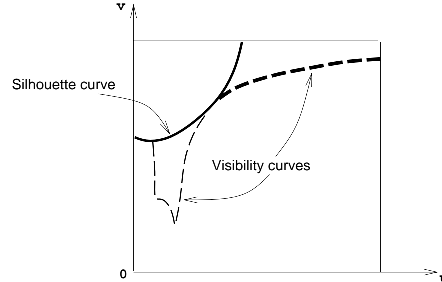


Figure 11: (a) Partitioning of patch in Figure 1 based on visibility curves (b) Visibility curves in the domain

Model	Degree of patch	Curve computation time (in millisecs.)	Running time (in millisecs.)
Fig. 4	3×3	27.8	53.0
Fig. 5	3×3	23.6	42.4
Fig. 6	1×8	17.3	33.2

Table 1: Performance of our algorithm

computation is the total time for silhouette and visibility curve generation, and the column for running time gives the total time taken by the algorithm for curve generation and producing the non-overlapping partitions.

8 Conclusion

We have presented an algorithm for computing the visible portions of a scene composed of curved (parametric) surfaces from a given viewpoint. We have also given a method to compute the silhouette curve efficiently and correctly. We introduced the notion of visibility curves, which are used to partition each patch into non-overlapping regions. The algorithm has been implemented in floating point arithmetic and performs well in practice.

9 Acknowledgments

We would like to thank David Banks, David Eberly, Brice Tebbs and Turner Whitted for all the productive discussions and helpful insights. We would also like to thank Brice Tebbs for providing the patch in Figures 1 and 5.

References

[BO79] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[Can88] J.F. Canny. *The Complexity of Robot Motion Planning*. ACM Doctoral Dissertation Award. MIT Press, 1988.

[CCW93] L. Chen, S. Chou, and T.C. Woo. Separating and intersecting spherical polygons: computing machinability on three, four and five axis numerically controlled machines. *ACM Transactions on Graphics*, 12(4):305–326, 1993.

[CE88] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. In *Proc. 29th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 590–600, 1988.

[Dix08] A.L. Dixon. The eliminant of three quantics in two independent variables. *Proceedings of London Mathematical Society*, 6:49–69, 209–236, 1908.

[Dor94] S. E. Dorward. A survey of object-space hidden surface removal. *International journal of Computational Geometry and Applications*, 4:325–362, 1994.

[EC90] G. Elber and E. Cohen. Hidden curve removal for free form surfaces. *Computer Graphics*, 24(4):95–104, 1990.

[EC93] G. Elber and E. Cohen. Second order surface analysis using hybrid symbolic and numeric operators. *ACM Transactions on Graphics*, 12(2):160–178, 1993.

[EC94] G. Elber and E. Cohen. Exact computation of gauss maps and visibility sets for freefrom surfaces. Technical report CIS #9414, Computer Science Department, Technion, 1994.

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

[FDHF90] J. Foley, A. Van Dam, J. Hughes, and S. Feiner. *Computer Graphics: Principles and Practice*. Addison Wesley, Reading, Mass., 1990.

[Ful78] W. Fulks. *Advanced Calculus: An introduction to analysis.* John Wiley & sons, 1978.

[GBDM77] B.S. Garbow, J.M. Boyle, J. Dongarra, and C.B. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension*, volume 51. Springer-Verlag, Berlin, 1977.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computations.* John Hopkins Press, Baltimore, 1989.

[GLR82] I. Gohberg, P. Lancaster, and L. Rodman. *Matrix Polynomials.* Academic Press, New York, 1982.

[GO87] R. H. Güting and T. Ottmann. New algorithms for special cases of the hidden line elimination problem. *Comput. Vision Graph. Image Process.*, 40:188–204, 1987.

[GWT94] J.G. Gan, T.C. Woo, and K. Tang. Spherical maps: Their construction, properties, and approximation. *ASME Trans. J. Mech. Des.*, 1994. To appear.

[HG77] G. Hamlin and C. W. Gear. Raster-scan hidden surface algorithm techniques. *Computer Graphics*, 11:206–213, 1977.

[Hor84] C. Hornung. A method for solving the visibility problem. *IEEE Computer Graphics and Applications*, pages 26–33, July 1984.

[KM95] S. Krishnan and D. Manocha. Numeric-symbolic algorithms for evaluating one-dimensional algebraic sets. In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, pages 59–67, 1995.

[KM97] S. Krishnan and D. Manocha. An efficient surface intersection algorithm based on the lower dimensional formulation. *ACM Transactions on Graphics*, 16(1):74–106, 1997.

[KM98] S. Krishnan and D. Manocha. Decomposing spline surfaces into non-overlapping regions for visible surface computation. In *Proc. of Indian Conference on Computer Vision, Graphics and Image Processing [to appear]*, 1998.

[Li81] L. Li. Hidden-line algorithm for curved surfaces. *Computer-Aided Design*, 20(8):466–470, 1981.

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

[Mul89] K. Mulmuley. An efficient algorithm for hidden surface removal. *Computer Graphics*, 23(3):379–388, 1989.

[Mul90] K. Mulmuley. An efficient algorithm for hidden surface removal, ii. Report TR-90-31, Univ. Of Chicago, Chicago, Illinois, 1990.

[Mun75] J.R. Munkres. *Topology: A First Course.* Prentice-Hall, 1975.

[NM95] A. Narkhede and D. Manocha. Fast polygon triangulation based on seidel's algorithm. In A. Paeth, editor, *Graphics Gems V*, pages 394–397, Academic Press, 1995.

[PVY92] F. P. Preparata, J. S. Vitter, and M. Yvinec. Output-sensitive generation of the perspective view of isothetic par allelepipeds. *Algorithmica*, 8:257–283, 1992.

[RE93] M. F. Roy and T. Van Effelterre. Aspect graphs of algebraic surfaces. In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, Kiev, Ukraine, 1993.

[RS88] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithms and its parallelization. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 193–200, 1988.

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

[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.

[SSS74] I. Sutherland, R. Sproull, and R. Schumaker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974.

[TW93] B. Tebbs and T. Whitted. Numerical Design Limited, Personal Communication, 1993.

[Woo94] T. Woo. Visibility maps and spherical algorithms. *Computer-Aided Design*, 26(1):6–16, 1994.

Appendix

We now present a slight variation of the hidden surface removal algorithm of Mulmuley [Mul89].

The input to the algorithm is a set of n faces with their boundaries represented as a collection of Bézier curves and piecewise linear chains. We also have the entire face boundary as a closed simple polygon. It will be used during the tracing step. We provide an overview of the algorithm first. Let us represent the set of faces by $H = \{h_1, h_2, \ldots, h_n\}$. Let V^i denote the collection of visible regions after adding i faces in random. The i faces already added are kept in the set Q. We want to compute V^n. After i steps of the algorithm, we maintain V^i. At the $(i+1)^{st}$ step, we pick a random face h_{i+1} from the set $H \setminus Q$. For each face in V^i, we find all the boundary intersections with the face h_{i+1} using the method described in the previous section. Let us consider a specific intersection point (p, q) such that p lies on the boundary of one of the faces in V^i and q lies on the boundary of h_{i+1}. If the $z-$coordinate of p is less than that of q (p lies in front of q), then project the boundary curve on which p lies on h_{i+1}, else the other way around. Tracing is accomplished by inverse power iteration, which was described in both Sections 2.4 and 6. The equations used to trace these curves is exactly the same as those used in Section 6. Tracing is done for all the intersection points. We partition appropriate faces by their projection curves and locate a point r inside each of the partitioned faces. Since all points inside one region is now entirely visible or not, we check only on one point. We shoot a ray from the point r to $-\infty$ in the $z-$direction and find the number of intersections with faces of $V^i \cup \{h_{i+1}\}$. The curve/surface intersection method used is described in Section 2.3. If the number of intersections is 0, this face is added, otherwise it is discarded. All the faces that were not partitioned in step $(i+1)$ are retained in V^{i+1}. After all the n faces are added, we obtain V^n. We shall now provide the pseudo-code for this algorithm. For ease of writing the pseudo-code, we shall assume that we have a routine *project_boundary_curves* that takes two faces as parameters and computes all the projected boundary curves as described above. It returns 0 if there are no boundary curves between the two faces, otherwise, it returns 1. We also have another routine called *partition_face* that computes the partition of the face using the projected boundary curve.

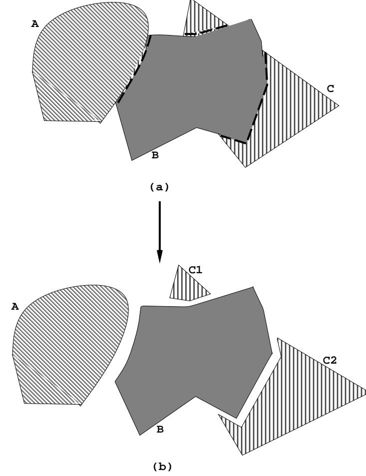


Figure 13: Visibility computation on a set of faces

$Q = \emptyset$;
$H = \{h_1, h_2, \ldots, h_n\}$;
$V^0 = \emptyset$;
for $(i = 1; i \leq n; i++)\{$
  h_i = random face from set $H \setminus Q$;
  $Q = Q \cup h_i$;
  $V^i = \emptyset$;
  $T = \emptyset$; /* stores faces that are partitioned in this iteration */
  for each face $f \in V^{i-1}$ do {
    j = project_boundary_curves(f, h_i, &c);
    if $(j == 0)$ $V^i = V^i \cup \{f\}$;
    else {
      $T = T \cup$ partition_face(f, c);
      $T = T \cup$ partition_face(h_i, c);
    }
  }
  for each face $t \in T$ do {
    (s_x, s_y, s_z) = some point inside t;
    r = semi-infinite ray from (s_x, s_y, s_z) to $(s_x, s_y, -\infty)$;
    if (ray_intersection_count(r) == 0) $V^i = V^i \cup \{t\}$;
  }
}
output V^n;

Figure 13 shows how our algorithm works. Each face boundary is made up of curves and piecewise straight lines. The dark broken lines shown in Figure 13(a) are the projected boundary curves drawn on the objects behind. Figure 13(b) shows the result of the algorithm on the set of faces. The complexity of this algorithm varies according to the order in which the faces are added. Therefore, by adding the faces randomly we reduce the expected running time of the algorithm. The output of the algorithm is a set of visible portions of patches represented as trimmed surfaces. The accuracy of the trimming curve can be adjusted according to the preferences of the user by changing the step size in the tracing method.