Hidden Surface Removal Algorithms for Curved Surfaces *

Shankar Krishnan Dinesh Manocha

Department of Computer Science,
University of North Carolina,
Chapel Hill, NC 27599-3175

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 hidden surface computations on curved surfaces. Given a viewpoint, 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 a simple randomized algorithm for visibility computations from a given viewpoint.

*Supported in part by a Sloan Fellowship, university research council grant, NSF grant CCR-9319957, ONR contract N00014-94-1-0738, ARPA contract DABT63-93-C-0048, NSF/ARPA Science and Technology Center for Computer Graphics & Scientific Visualization and NSF Prime contract No. 8920219

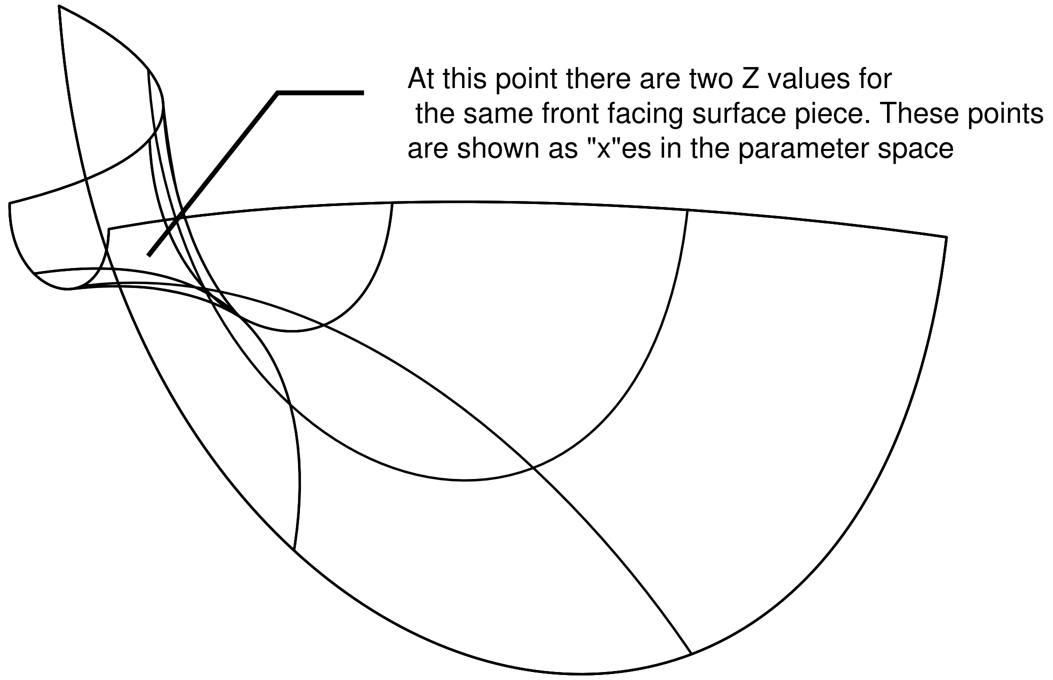
1 Introduction

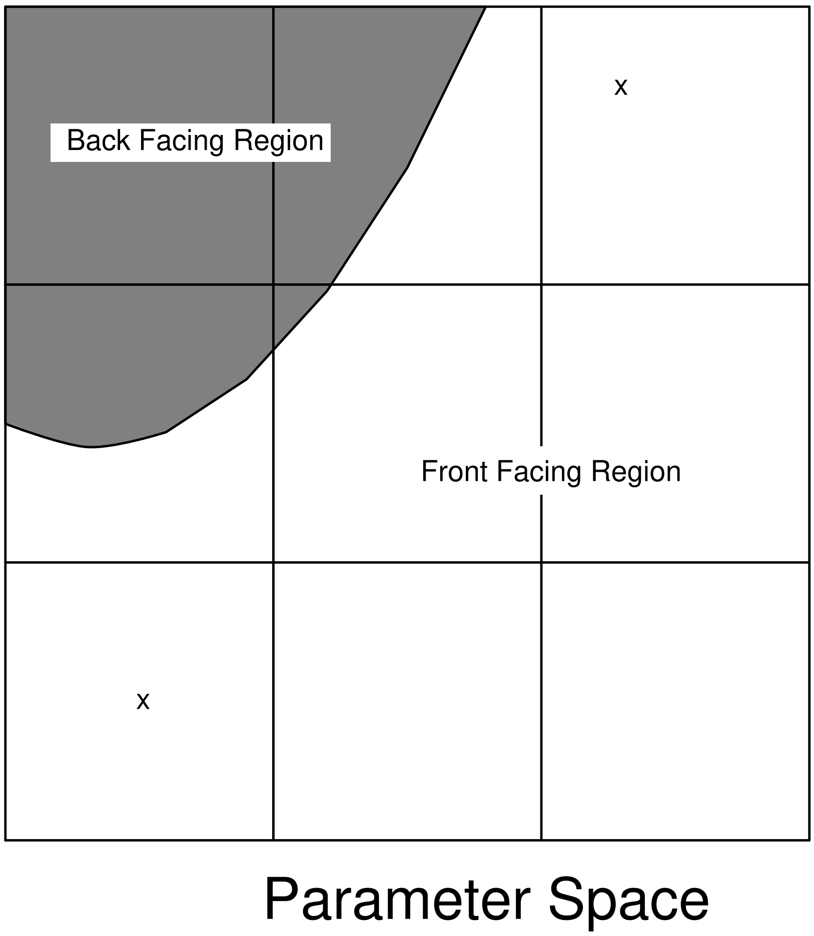
The problems of visibility and accessibility computations are fundamental for computer graphics, computer-aided design and manufacturing applications. In particular, hidden line and surface removal algorithms in computer graphics are related to visibility computations [FDHF90, Hor84, SSS74, HG77]. Similarly, 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 [Mul89] 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. In [Mul91], the expected time was improved to $O(n \log n + q)$, where q is the number of edges in the visibility map. 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 [Mul90]. Results from elimination theory and algebraic decision procedures like Gröbner 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 developed to approximate these numbers using finite precision. 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 techniques for visibility computations are based on ray-tracing [FDHF90]. Not only are the resulting algorithms slow but their accuracy is limited by the image-precision. These techniques are device resolution dependent and many applications in modeling and rendering desire a *device-independent* solution [TW93]. 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 Fig. 1). The silhouettes are composed of points on surfaces where the normal vector is orthogonal to the viewing direction. The surfaces obtained after partitioning based on the silhouette computation need not be completely visible, as shown in Fig. 1. More recently, a hidden curve removal algorithm has been presented for parametric surfaces





Patch Coordinates

261.747	102.9	-57.4605
199.883	95.7857	28.4751
82.1126	93.3985	120.048
-56.2504	112.366	87.0828
218.825	-205.839	-67.2272
173.737	-97.1096	20.2249
76.4257	41.7767	113.648
-60.3006	72.0456	80.8298
78.9959	-257.342	355.755
86.3181	-129.309	284.667
52.9209	33.118	184.749
-78.6922	65.2702	136.464
97.1888	42.2773	440.321
98.4073	58.4023	335.435
57.5666	84.3559	194.299
-74.642	105.59	142.717

Figure 1: Local Visibility Computations Based on Silhouettes

by Elber and Cohen [EC90]. In particular, [EC90] 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. Their intersection algorithms are based on subdivision.

We present an algorithm for hidden surface removal on a set of parametric surfaces from a given viewpoint. We decompose it into a series of Bézier surfaces. Each Bézier surface is partitioned into non-overlapping regions based on silhouette and *visibility curves*. 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 simple randomized algorithm to compute the visible portions. Our algorithm can easily be extended to handle intersecting surfaces. However, for ease of argument we assume that the surfaces are non-intersecting. The algorithm uses a combination of symbolic techniques and results from numerical linear algebra. The algorithm is simple to implement and it has been implemented in fixed precision (using 64-bit double precision floating-point arithmetic). The main aim of this paper is to present a simple implementable algorithm for the hidden surface problem that performs reasonably well in practice.

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. We outline an efficient algorithm for computing the silhouettes based on marching methods in Section 3. We introduce the notion of visibility curves in Section 4 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 5. Finally, Section 6 describes a simple randomized visible surface computation algorithm for multiple non-overlapping surfaces.

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 the most optimal in terms of worst case asymptotic complexity; we compromised this over 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 $O(N \log^* N)$,

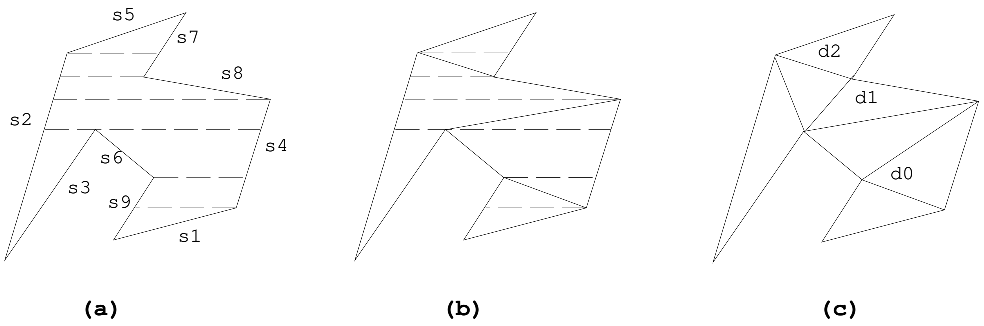


Figure 2: Three stages of Seidel's algorithm

where N is the number of vertices. In practice, it is almost linear time. The algorithm proceeds in three steps as shown in fig. 2. They are:

- Decompose the polygon into trapezoids in $O(N \log^* N)$ time,
- Decompose the trapezoids into monotone polygons in linear time, and
- Triangulate the monotone polygons in linear time.

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.

2.2 Partitioning a Simple Polygon using Non-Intersecting Chains

We use algorithms for partitioning a simple polygon based on 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. Let us assume for the sake of simplicity that the chain does not form a loop inside the polygon. We handle this case as a special case in our algorithm, and hence details are omitted. 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. Fig. 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 (see fig.). 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. This problem can be solved in time $O(N \log^2 N)$ [CEGS94], where N is the total number of segments in the polygon and chains. We associate three fields

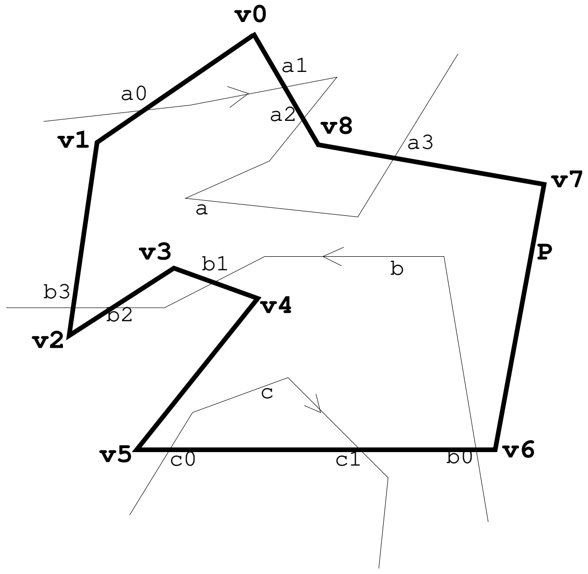


Figure 3: Partitioning a polygon with polygonal chains

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 in some order 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 $\neq$ p.type)
   push(q);
if ((q.rank - p.rank == 1) && (q.in_or_out == *out*) && (p.in_or_out == *in*))
   pop(p);
if ((p.rank - q.rank == 1) && (p.in_or_out == *out*) && (q.in_or_out == *in*))
   pop(p);
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). Some details about reading out the proper partitions once a vertex is popped are omitted. After traversing the polygon completely once, we would have obtained all the partitions.

2.3 Curve/Surface Intersection

Computing the intersection of curves and surfaces is needed to find whether a given surface is occluded. Basically we shoot a ray from a point on the surface to the viewpoint and determine if the ray intersects with any other surface. 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].

Given a surface $\mathbf{F}(s,t)$, we compute its implicit representation using resultant methods [Dix08] and obtain a matrix formulation $\mathbf{M}(x,y,z,w)$. We substitute the parametrization of the curve, say $\mathbf{G}(u) = (\overline{X}(u), \overline{Y}(u), \overline{Z}(u), \overline{W}(u))$ of degree d, and obtain a univariate matrix polynomial $\mathbf{M}(u)$. The problem of intersection computation reduces to computing the roots of the non-linear matrix polynomial $\mathbf{M}(u)$. The polynomial which are in Bernstein basis can be converted to the power basis by the transformation $\overline{u} = \frac{u}{1-u}$. The resulting matrix $\mathbf{M}(\overline{u})$ can be represented as

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

where M_i's are matrices of order $2mn$ with numeric entries. Furthermore, the roots of the matrix polynomial, $\mathbf{M}(u)$, have one-to-one correspondence with the eigendecomposition of

$$C = \begin{bmatrix} 0 & I_n & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & I_n \\ -\overline{\mathbf{M}}_0 & -\overline{\mathbf{M}}_1 & -\overline{\mathbf{M}}_2 & \ldots & -\overline{\mathbf{M}}_{d-1} \end{bmatrix} \tag{2}$$

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]. They compute all the real as well as complex eigenvalues. Algorithms to compute eigenvalues in a subset of the real or complex domain are presented in [MD94].

2.4 Power Iterations

We use marching methods to trace the visibility curves. At each iterations, 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, $\mathbf{A}$, we can find $\mathbf{X}(= [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n])$ 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 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 $\| \mathbf{z}_k \|_\infty$ 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

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

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 stepsize 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.

3 Silhouettes

Silhouette computation forms an important part of visibility algorithms for curved surfaces. 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 fig. 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.

We assume for the sake of simplicity that the viewpoint is located at $(0, 0, -\infty)$. 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 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 $\mathbf{F}(s,t)$.

3.1 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.

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

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

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

$$N_z = \begin{vmatrix} \phi_{1_s} & \phi_{1_t} \\ \phi_{2_s} & \phi_{2_t} \end{vmatrix} \tag{3}$$

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

$$N_z = \begin{vmatrix} (WX_s - W_sX) & (WX_t - W_tX) \\ (WY_s - W_sY) & (WY_t - W_tY) \end{vmatrix} = 0 \tag{4}$$

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

$$\mathbf{M}(s,t) = \begin{pmatrix} X(s,t) & Y(s,t) & W(s,t) \\ X_s(s,t) & Y_s(s,t) & Z_s(s,t) \\ X_t(s,t) & Y_t(s,t) & Z_t(s,t) \end{pmatrix} = 0 \tag{5}$$

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

3.2 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 and can have a number of components (both open and closed) possibly with singularities. Our task is to evaluate this curve 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 eq. (2)) [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 pairs. We characterize certain special points on loops (turning points) as places where two complex conjugate paths merge to form a real component [KM95]. By following all the complex paths inside the domain we can locate at least a single point on each loop.

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. The three main issues concerning tracing algorithms are:

1. Converging back on to the curve.
2. Preventing *component jumping.*
3. Ability to handle singularities and trace through multiple branches.

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 [KM95]. 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 (eq. (2), 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).

3.3 Surface Partitioning based on Silhouettes

Fig.4 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. We shall make this concept more precise using the following lemma.

Lemma 1 *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* $\mathbf{C}$ *partitions* $\mathbf{S}$*.*

Proof: Let p, q and r be points on $\mathbf{S}$. Consider the natural relation $\Im$ induced by $\mathbf{C}$ on $\mathbf{S}$. $p \Im q$ if and only if there exists a continuous path from p to q without crossing any silhouette curve. Let us look at the properties of $\Im$.

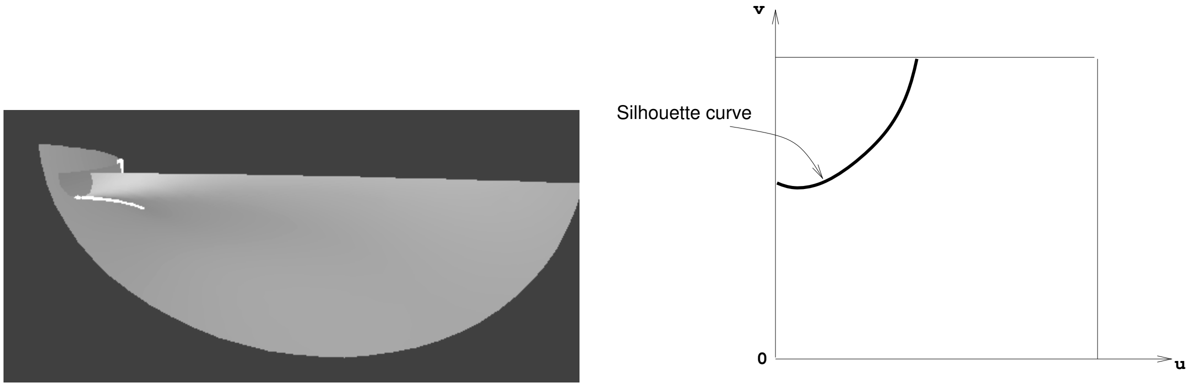


Figure 4: Silhouette curve on the patch and the domain

- $\Im$ is symmetric because $p \Im q \Rightarrow q \Im p$.
- $\Im$ is reflexive because $p \Im p$.
- $\Im$ is transitive because $p \Im q$ and $q \Im r \Rightarrow p \Im r$.

Thus $\Im$ is an *equivalence relation*. Since an equivalence relation induces a partition of the set, **C** partitions **S**.

$\square$

From the above lemma we can conclude that **S** can be partitioned into a set of regions **R** such that the boundaries of each region $\wp \in \mathbf{R}$ consists of original surface boundaries and silhouette curves.

Given the silhouette curve on the domain as a polygonal chain, we initially check for self-intersections (in case the silhouette curve has a singularity) and break it into non-intersecting chains. The domain of each patch is represented as a polygon (not necessarily convex in case of trimmed patches) 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* routine to subdivide the domain into disjoint regions.

4 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 global visibility. 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 2 *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 locus of all points on the surface that have vanishing Jacobians correspond exactly to the silhouette curve.*

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

$$J(s,t) = \begin{vmatrix} \frac{(WX_s - W_sX)}{W^2} & \frac{(WX_t - W_tX)}{W^2} \\ \frac{(WY_s - W_sY)}{W^2} & \frac{(WY_t - W_tY)}{W^2} \end{vmatrix} \tag{6}$$

However when $J = 0$, we get the same equation as eqn. 4. 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.

□

After determining the silhouette curve we proved that it introduces a partition on the surface. Let us denote this set of regions obtained after partitioning as 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 Jacobians. This implies that we can apply the *global inverse theorem* on each of the regions in R.

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,*

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

Lemma 3 *The visibility curves on $\wp$, $V(\wp)$ as defined above, indeed, form a set of curves with the property that each such curve ends in $\partial\wp$.*

Proof: We shall prove both the parts 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 $M(p_i + \epsilon) = M(p_b + \delta)$. 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 $p_i \in int(\wp)$.

Figure 5: A helical patch with no silhouettes

□

The previous argument showed that if we are in the interior of $\wp$, we can always find more points on $V(\wp)$ unless we return to an original point on $V(\wp)$. Therefore, all visibility curves must either extend to the boundary of $\wp$ or end up as loops.

Let p_b be a point on the boundary, and let p_i be the corresponding point in the interior with the same projection. If p_i is a point on the loop, we can show that there exists a simple path between p_i and p_b which crosses a silhouette curve. Therefore, visibility curves cannot exist as loops.

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

Proof: We can prove this fact using the same argument as Lemma 1.

□

Fig.5 shows a helical patch with no silhouettes. Fig.6 shows the visibility curves computed on the same patch. The visibility curves are also shown in the domain of the patch. The patch in fig.5 is deliberately transformed to provide a better view. Lemma 3 provides a constructive method to find all the visibility curves. Given the boundary representation 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 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 can construct a partition of $\wp$ induced by the visibility curves using Corollary 1. 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. Fig. 7(a) shows the partition of

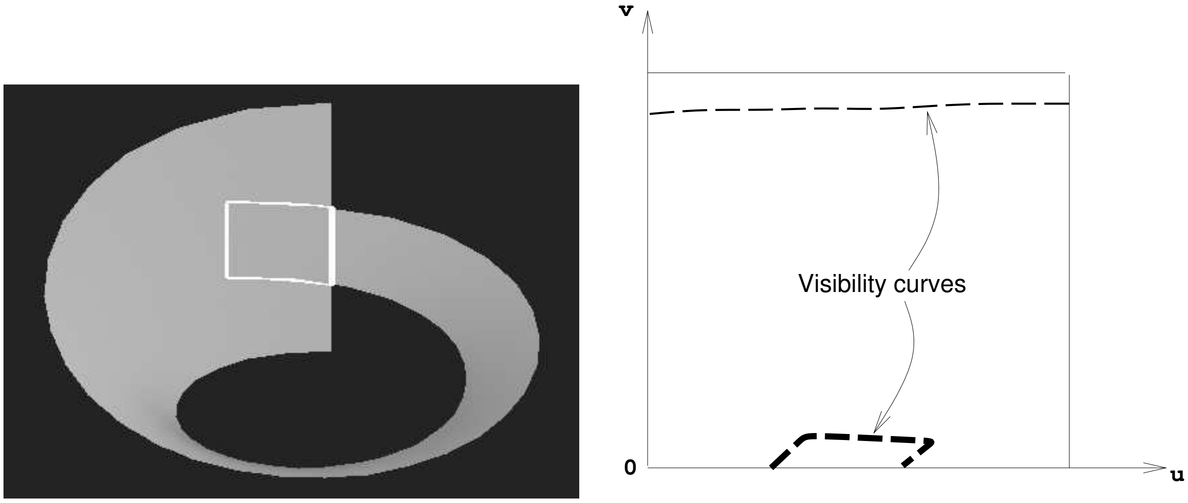


Figure 6: Visibility curves on the helical patch

the helical patch based on the visibility curves. In fig. 7(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 $a\prime$). However, in the next lemma, we are going to argue that the the interior of each region $k \in K_{\wp}$ is one-to-one under the map M.

Lemma 4 *Let $k \in K_{\wp}$. Then $int(k)$ is one-to-one under the projection mapping M.*

Proof: The main idea in the proof is to construct a simple closed curve C in $int(k)$ such that it 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.

It is clear that the closed curve representing $M(\partial k)$ of each region k is not necessarily one-to-one (region $K2$ in fig. 7(b)). Therefore, in order to construct C let us look at the generation of a visibility curve shown in fig. 8. Let $b1$ and $b2$ be the projection of two boundary curves of the region $\wp$ intersecting at the point p. p splits the curve $b1$ into two parts: one that lies locally inside the region whose boundary is $b2$, and the other that lies outside. A similar observation holds for $b2$ as well. The curves that lie in the interior are marked darker in the figure. These are the curves that precisely become visibility curves when projected onto the other curve. These visibility curves partition the region locally near p into at least two regions $K1$ and $K2$ (by corollary 1). The boundary curve of region $K2$ is not one-to-one because there are two points (say a and b) having the same projection point p. However, locally near p, it is the only point that violates the one-to-one property of $K2$. Therefore, by removing the points a and b from the boundary curve and closing the curve using interior points in the neighborhood of a and b, we get rid of point p. Extending this to all intersection points, we get a simple closed curve C which is one-to-one with the property that every interior point of the region lies either on or inside C.

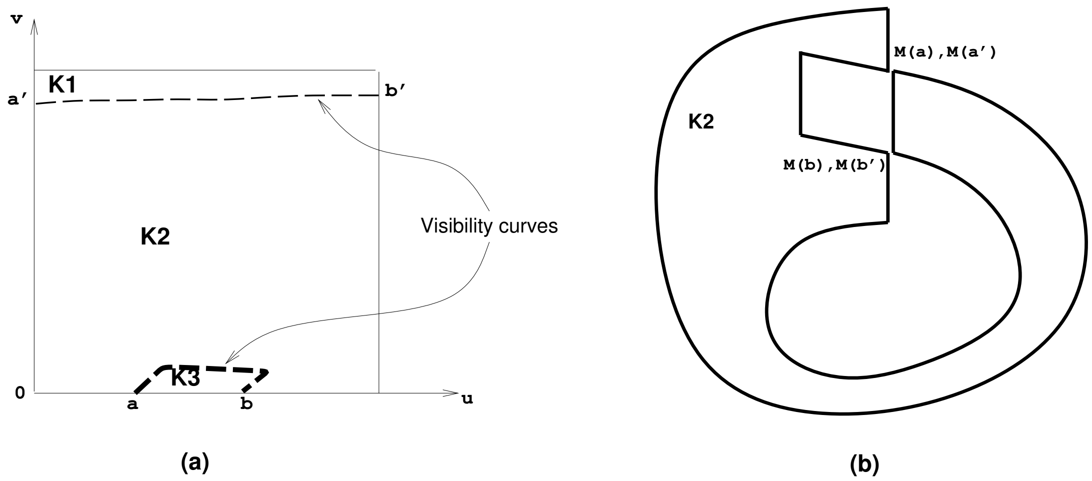


Figure 7: (a) Partitioning based on visibility curves (b) Image of region K2 under M

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.

□

5 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.

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

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

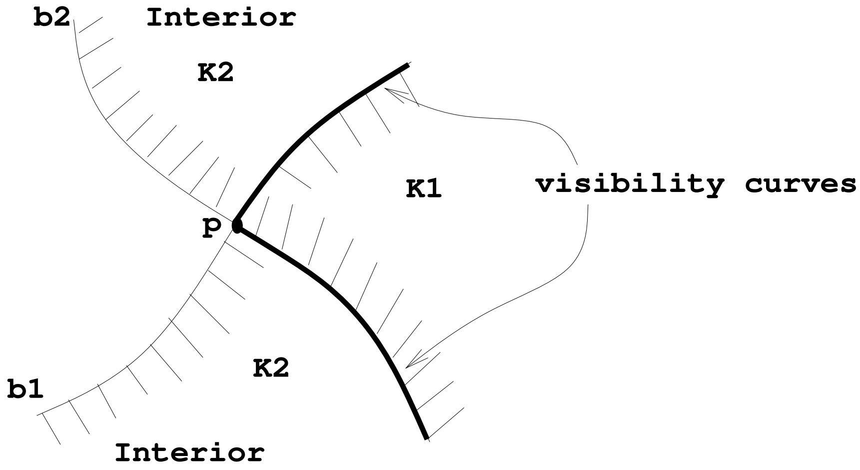


Figure 8: Generation of visibility curves near the boundaries

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

where X_f, Y_f, W_f and X_g, Y_g, W_g are the componentwise 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 [GLR82]. 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 2 and X, Y, Z and W are the componentwise functions of the patch. These four equations in four variables, typically, 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. We use the Bentley-Ottmann algorithm [PS85] to obtain the intersection points in $O((N+k)\log N)$ time, where k is the number of intersection points. This algorithm is close to optimal because in our case k is very small. Each intersection point obtained by this method is refined using local minimization methods like Powell's method using the energy equation $E(u,v,s,t)$ given by

$$\begin{aligned} E(u,v,s,t) = {} & D^2(s,t) + D^2(u,v) + \\ & (X(s,t)W(u,v) - X(u,v)W(s,t))^2 + \\ & (Y(s,t)W(u,v) - Y(u,v)W(s,t))^2. \end{aligned}$$

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

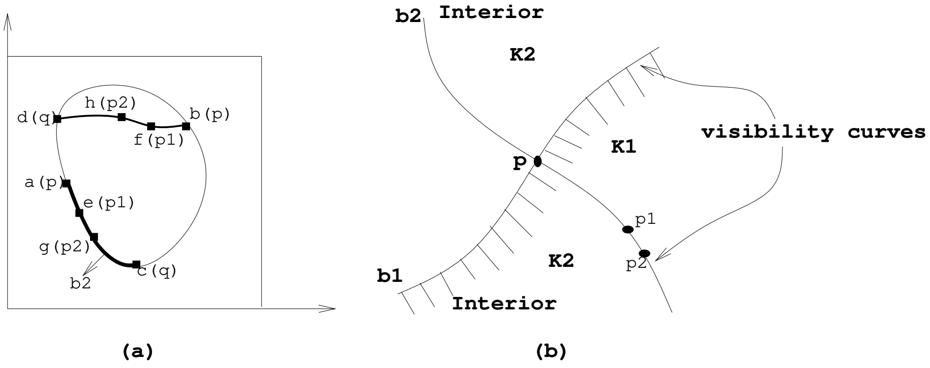


Figure 9: Tracing of visibility curves

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

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

5.2 Tracing Visibility Curves

Given the starting point of each visibility curve, we are ready to trace it. The starting point is represented in the following form

```
struct start_point {
    double xval, yval;  /* x and y value of intersection point */
    double dom_point1_u, dom_point1_v;  /* domain coordinates of first point */
    double dom_point2_u, dom_point2_v;  /* domain coordinates of second point */
    int index_of_first_point;  /* index of the first point in the polygonal region */
    int index_of_second_point;  /* index of the second point in the polygonal region */
}
```

Fig. 9 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 fig. 9(b) correspond to the portion of the boundary from a to c (see fig. 9(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 componentwise 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

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

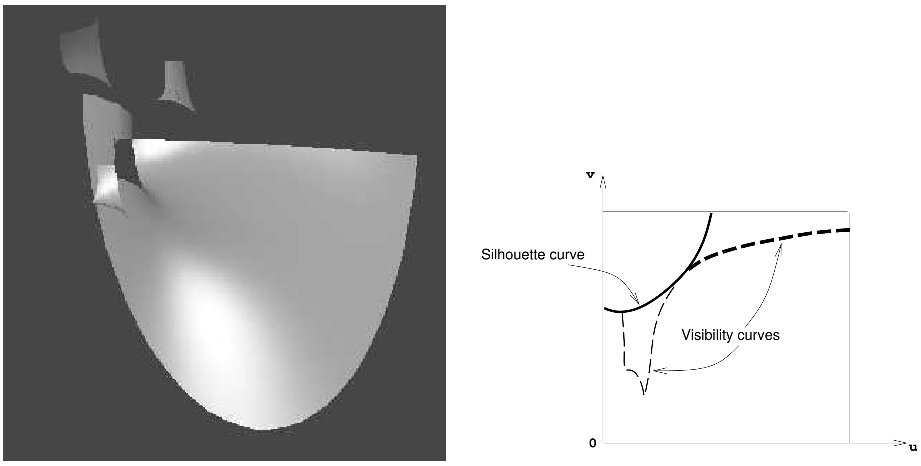


Figure 10: (a) Partitioning of patch in fig.1 based on visibility curves (b) Visibility curves in the domain

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

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) 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.

Figure 11: Partitioning of patch in fig.5 based on visibility curves

After computing all the visibility curves, we calculate all the partitions induced by the visibility curves in each region using the *partition_polygon* routine. Figs. 10 and 11 show the partitioning of the patches in fig.1 and fig.5 using the visibility curves. Each partition thus obtained is one-to-one under projection. It transforms the original visibility problem into one of n polygons on space.

6 Visibility for complex scenes

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]. However, we present a simple algorithm based on the the same projection idea that works well.

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 [KM95] and split them into non-intersecting faces.

The input to the algorithm is a set of n faces, each of whose boundaries is 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 only one of the curves is sufficient because the faces are non-intersecting. Tracing is accomplished by inverse power iteration, which was described in both sections 2 and 5. The equations used to trace these curves is exactly the same as those used in section 5. In some sense, the projected curves are similar to our notion of visibility curves. We split it up this way so that the theoretical argument could be made much easier. We do tracing 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. 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 pseudocode for this algorithm. For ease of writing the pseudocode, 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.

```
Q = ∅;
H = {h_1, h_2, ..., h_n};
V^0 = ∅;
for (i = 1; i ≤ n; i++){
   h_i = random face from set H \ Q;
   Q = Q ∪ h_i;
   V^i = ∅;
   T = ∅;  /* stores faces that are partitioned in this iteration */
   for each face f ∈ V^{i-1} do {
      j = project_boundary_curves(f, h_i, &c);
      if (j == 0) V^i = V^i ∪ {f};
      else {
         T = T ∪ partition_face(f, c);
         T = T ∪ partition_face(h_i, c);
      }
   }
   for each face t ∈ 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, -∞);
      if (ray_intersection_count(r) == 0) V^i = V^i ∪ {t};
   }
}
output V^n;
```

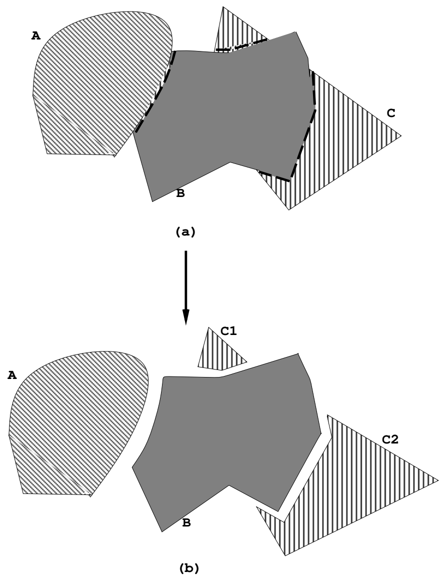


Figure 12: Visibility computation on a set of faces

Fig.12 shows how our algorithm works. Each face boundary is made up of curves and piecewise straight lines. The dark broken lines shown in fig. 12(a) are the projected boundary curves drawn on the objects behind. Fig. 12(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.

6.1 Implementation and Performance

The algorithm has been implemented and its performance was measured on a number of models. 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 high-end SGI Onyx workstation with a single processor configuration. The performance of the algorithm was measured on a number of bicubic (of degree 3 in both parameters) patches. It takes a few milliseconds to produce non-overlapping regions on each patch. We used bicubic patches for most of our experiments because most models used in graphics applications are seldom of higher degree.

7 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.

8 Acknowledgement

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 Fig.1 and 4.

References

[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.

[CEGS94] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. *Algorithmica II*, pages 116–132, 1994.

[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*, 1995.

[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, Dept. Comput. Sci., Univ. Of Chicago, Chicago, Illinois, 1990.

[Mul91] K. Mulmuley. Hidden surface removal with respect to a moving point. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 512–522, 1991.

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

[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.

[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.