A PLANE VIEW OF GEOMETRIC SILHOUETTES

by

Matthew Olson
B.Sc., University of Alberta, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2011

APPROVAL

Name: Matthew Olson

Degree: Doctor of Philosophy

Title of Thesis: A Plane View of Geometric Silhouettes

Examining Committee: Dr. Mark Drew
Chair

Dr. Hao Zhang, Senior Supervisor

Dr. Torsten Möller, Supervisor

Dr. Greg Mori, SFU Examiner

Dr. Bruce Gooch, External Examiner,
University of Victoria

Date Approved: _____

Declaration of
Partial Copyright Licence

Abstract

Geometric silhouettes are arcs on a surface representation that separate front-facing regions from back-facing regions with respect to a given viewpoint. These arcs are in general significantly less complex than the surface itself, and carry a great deal of information describing the surface. In this thesis, we take a *plane* view of geometric silhouettes, defining them in terms of the tangential planes of the surfaces on which they are defined rather than in terms of its local properties. We show that this perspective leads to efficient algorithms as well as a novel characterization of silhouettes based on a *silhouette-generating set*, or SGS.

The low asymptotic complexity of mesh silhouettes, combined with their utility, justifies the development of silhouette extraction algorithms that are sublinear in the size of the input model. Many of these more efficient algorithms are based on tangential-plane representations of the input model. We present a novel silhouette extraction and update algorithm based on the *3D Hough transform*, which combines the advantages of previous tangential-plane representations. We begin by presenting this algorithm on triangle meshes, then extend it to support point-set surfaces. In doing so, we generalize the double-wedge structure underlying mesh-edge silhouettes to an SGS applicable to arbitrary primitives.

While our plane-based data representation allows us to identify silhouettes on distant parts of the input model when their SGSes coincide, it is nonetheless a local approach in 3D Hough space. However, by *aggregating* tangential plane information over the entire input mesh, we can perform a number of global optimizations effectively. We introduce the *tangential distance field* (TDF), a scalar function based on the SGSes of *all* triangles in a mesh. We develop a toolbox of weighting functions which embed different geometric information in the TDF. Depending on the function chosen, we can find a set of optimized origins for our silhouette extraction algorithm, a set of visually informative viewpoints around a given model, or a similarly informative light position based on a given viewpoint.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Silhouettes are a powerful perceptual cue and are deeply ingrained in our notion of shape; thus, it is important to define precisely what we mean when we discuss silhouette problems in computer graphics. We begin by presenting an intuitive view of the problem, and refine our definitions in Chapter 2. Glisse and Lazard [32] present three different intuitive definitions of the silhouette, as shown in Figure 1.1.

The intuitive notion of the silhouette is strongly connected to visibility. Figure 1.1(a) depicts the *visual hull* of the object, which defines the region in the visual field occluded by the object as a whole. Similarly, Figure 1.1(b) depicts the *visible silhouette* or *visible rim* of the object, which extends the notion of occlusion to the object itself (self-occlusion). However, while these "silhouettes" correspond closely to our visual intuitions, their connection with the global problem of visible-surface determination makes them difficult to analyze and extract efficiently in a geometric framework. For that reason, in this thesis we confine

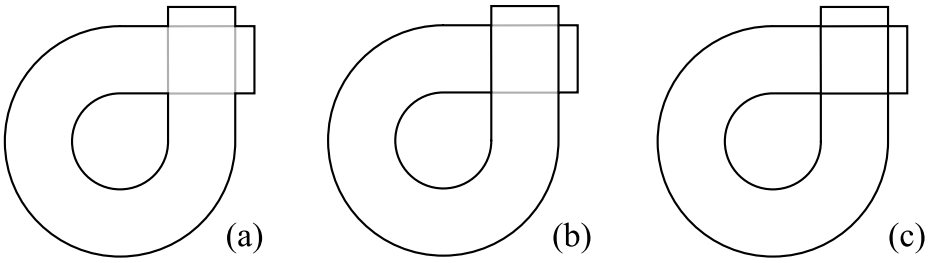


Figure 1.1: The silhouette of an object as (a) the outline of its shadow, (b) the visible rim of an opaque object, and (c) the visible rim of a transparent object.

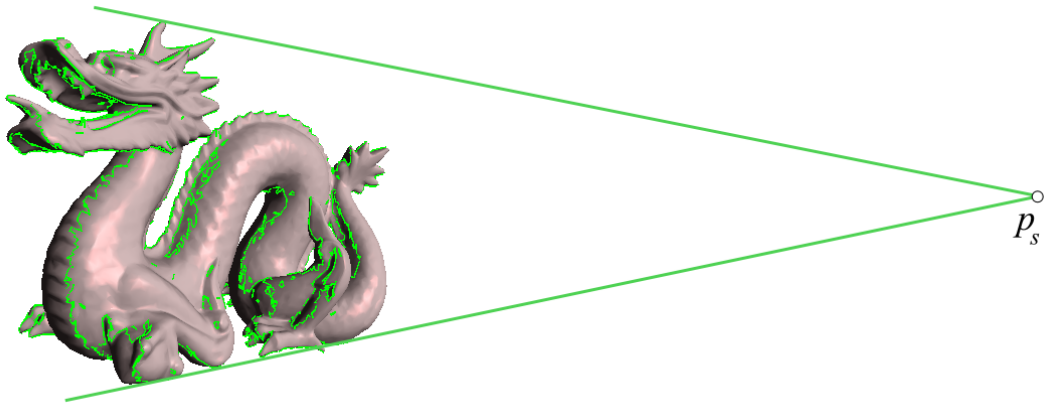


Figure 1.2: The silhouette of an object with respect to p_s, viewed from an oblique angle.

ourselves to the *geometric silhouette* of an object, shown in Figure 1.1(c) and briefly defined as the set of arcs separating front-facing from back-facing regions. (See Section 2.1 for a more rigorous discussion of these concepts.)

Our intuitive conception of geometric silhouettes often leads to confusion when the silhouette is being computed from a point distant from the camera, as in Figure 1.2. For clarity, we denote this point as the *silhouette point* p_s, and emphasize that p_s need not be placed at the camera. If the silhouette point is placed at infinity, we are instead calculating the *orthogonal* silhouette from the *silhouette vector* v_s. We present more detailed descriptions of these concepts in Section 2.1.

In this thesis, we seek to first find and then exploit the silhouette set of a 3D object. We will see that careful attention to the tangential planes of this object is invaluable. In the remainder of this chapter, we give a brief overview of our contributions.

1.1 A plane view of object-space silhouettes

In the computer graphics community, the geometric silhouette of a model is generally considered in terms of the local geometry of the surface. For the purpose of *computing* the silhouette, this is often the most convenient approach, and as part of a randomized algorithm can be made reasonably efficient. The result, however, is a purely local algorithm that risks neglecting many of the structural properties of the geometric silhouette's propagation across a surface as p_s moves.

The silhouette of a model has received attention in other communities, however, and some of these have produced insights which promise to complement its use in geometry processing. We begin by noting that silhouettes are defined in terms of a point p_s and either a *normal vector* (see Definition 2.1.2) or an *edge* (Definition 2.1.3). In either case we see that geometric silhouettes are defined by a *coplanarity relationship* between p_s and a subset of points on the model. This thesis aims to exploit this coplanarity.

We can describe the tangential planes associated with a surface by constructing its *pedal surface*, as done by Sethi *et al.* [76] in the context of shape recognition. The pedal surface consists of the loci of points closest to the origin on each plane tangential to the model, and is closely related to the *3D Hough transform* described in Section 2.3.2. Of special note is that the pedal curve unifies *bitangent points* on the surface; these points are important to algorithms that seek to update a model's silhouette under a moving p_s and cannot easily be obtained by a purely local formulation of the silhouette.

Furthermore, the identification of bitangents in a planar formulation of the silhouette hints at the availability of further global information from structures such as pedal surfaces. Lazebnik and Ponce [56] emphasize that "certain important properties of the visual world are *intrinsically projective*", including fundamental relationships such as visibility and occlusion. While this thesis cannot claim to provide solutions to general visibility problems, we hope that by strongly emphasizing the planar relationships involved in silhouette problems we can provide tools to attack these issues.

1.2 Hough space silhouette extraction

Previous silhouette-extraction algorithms based on supporting plane representations have followed the same general method: first, transform the mesh's supporting planes into points in an alternate space in which they are easily and efficiently organized; next, transform p_s into the same space; and finally, use intersection tests in an acceleration structure to identify the (transformed) supporting planes crossed by the (transformed) p_s. We follow the same method, but use a transformation which addresses the limitations of the existing algorithms.

This transformation is the *3D Hough transform*. It takes planes $\pi : ax + by + cz - d = 0$ (where $a^2 + b^2 + c^2 = 1$) to points $H(\pi) = (ad, bd, cd)$ – these correspond to the closest point on π to the origin. The points produced by applying this transform to the supporting planes of a mesh fall within a sphere containing the mesh and the origin; the same cannot be said of

the similar *geometric dual transform* used by Pop *et al.* [69] (where $D(\pi) = (a/d, b/d, c/d)$). Whereas the 3D Hough transform tends to produce relatively uniform and compact point distributions, which in turn tend to produce balanced and shallow octrees as acceleration structures, the geometric dual transform tends to cluster most of the transformed points near the origin while scattering a small but significant number of points arbitrarily close to infinity when d is small.

Two other transformations have been used for silhouette extraction. The first [8, 33] simply represents each supporting plane by its normal as a point on the Gaussian sphere. These methods are only able to perform silhouette extraction under orthographic projection. The second is the four-component parameter transform [39], which requires more complex data structures and more expensive intersection tests.

Unlike the geometric dual transform, which takes $p_s = (a, b, c)$ to the plane $D(p_s) : ax + by + cz + 1 = 0$, the 3D Hough transform takes p_s to the *sphere* with poles at p_s and the origin. This may be thought of as the loci of the 3D Hough transforms of every plane which passes through p_s. This sphere is our intersection primitive for determining the status of p_s within the half-spaces of our mesh's supporting planes.

To accelerate these intersection tests, we build an augmented octree around the set of points in Hough space. The cells of this octree contain bounding boxes for the data within each cell, and for the *neighbours* of the faces whose transformed supporting planes fall within the cell. The latter bounding box allows us to identify edges on the silhouette with respect to a static point p_s, something that the similar method of [69] cannot do. Like Pop *et al.*, we find changes in the silhouette from frame t to $t + 1$ by identifying transformed points between $H(p_s(t))$ and $H(p_s(t + 1))$. However, we add links from each node in the octree to its adjacent *neighbour* nodes, allowing us to maintain an active front of low-level octree cells and avoid the cost of starting each frame from the root.

1.3 Tangential distance fields and applications

Since we represent the supporting planes of a mesh's triangles by some function of their plane equations, the resulting point set in transform space depends upon the origin chosen. A natural choice of origin is the mesh centroid, or the centre of the mesh's bounding sphere; however, these are far from optimal depending upon the mesh. We will present a method for computing a more efficient set of origins and an algorithm for calculating the silhouette when mesh faces are clustered around several local origins.

The 3D Hough transform of a ray r is the circle which lies in a plane perpendicular to r, passing through r and the origin. As with the 3D Hough transform of a single point, this corresponds to the Hough transforms of all planes that include r. If we vary the origin, the likelihood that the sphere $H(p_s)$ intersects $H(r)$ does not change; however, in practice we test $H(p_s)$ against an *axis-aligned bounding box* (AABB) containing $H(r)$, not against $H(r)$ itself. These AABBs *do* vary significantly with the origin. In pathological cases they may even contain the origin; such boxes are never rejected by intersection tests against $H(p_s)$.

We introduce the *tangential distance field* or TDF as a means of identifying points in space that, if chosen as origins, will produce a transformed point distribution that itself produces more efficient octrees. For each point p_i in space (we consider a bounded region surrounding the mesh), we accumulate votes from the supporting planes π_i of each triangle in the mesh. These votes are functions of the distance between p_i and π_i; for the Hough space origin problem, we choose a bimodal function which weights points "close, but not too close" most heavily. The point p_1 with the highest summed weight is chosen as the first origin, and the process is repeated. For subsequent origins $p_{2..k}$, each face's vote is reduced by its highest previous vote; thus, planes that gave a strong vote to a previous origin have little influence on subsequent origins. We find empirically that two to five origins are sufficient to produce a two-fold to three-fold improvement in the performance of our silhouette extraction and update method.

In addition to finding optimized origins for Hough-space silhouette methods, we apply the TDF method to two other problems: finding a set of viewpoints that provide "best views" of a given triangle mesh, and finding light positions that provide "best illumination" of a mesh from a given viewpoint. In both cases we are able to achieve results comparable to state-of-the-art specialized algorithms, suggesting that TDFs have significant utility for general geometry processing.

1.4 Silhouette extraction on point clouds

The biggest problem facing silhouette extraction from point-set surfaces is the lack of connectivity. Without connectivity information we do not have the notion of separation from which we define silhouettes on triangle meshes – "edges that separate front- and back-facing faces". Furthermore, without connectivity information we must *approximate* neighbour relationships between nearby points, a task which often fails when two surfaces are geometrically close or point samples are coarse or nonuniform.

We begin by constructing a definition for the geometric silhouette of a point-set surface P based only upon the provided samples. We consider each point $p \in P$ to be a sample on an underlying surface S. Under this interpretation, each point has an intrinsic Voronoi cell on S, and the silhouette on S with respect to p_s is the set of all points $x \in S$ such that the normal vector $n_x \cdot v_s(x) = 0$ where $v_s(x) = x - p_s$. Then we say that p is on the silhouette of P with respect to p_s if some $x \in S$ is within p's intrinsic Voronoi cell.

More abstractly, a point $p \in P$ represents all points on S within its intrinsic Voronoi cell. Those points have normal vectors and define tangent planes. The *silhouette-generating set* or *SGS* of p is the set of points contained by the tangent plane of at least one point on S represented by p. When p_s is within the SGS of p, p is on the geometric silhouette of P with respect to p_s.

We cannot construct the SGS of p directly, as we do not have access to S. However, we can *approximate* p's SGS by finding a set of meaningful neighbours $Q = \{q_i\}$ for p and constructing an umbrella of triangles around p with vertices in Q. The supporting planes of those triangles approximates the SGS of p.

Beginning with a small set ($n = 16$ gives good results) of Euclidean nearest neighbours, we do this by considering the Gabriel triangle of p – the smallest triangle with an empty circumcircle containing both p and its nearest neighbour – as an initial estimate of p's tangent plane, and building a locally Delaunay triangulation of its adjacent neighbours. Using two independent parameters to describe the expected sampling characteristics and refining our tangent-plane estimate as we acquire further information, we identify boundaries, sharp edges, and Euclidean nearest-neighbours belonging to geodesically-distant surface sheets.

This approach is limited by the *actual* sampling conditions on the input, and therefore does not always find an optimal umbrella around each point. Furthermore, it is an entirely local approach, and will not provide a globally consistent triangulation. However, it is also less expensive than a full surface reconstruction algorithm, and its local approximations are quite suitable for our purpose. Silhouettes extracted from this algorithm are of high quality and exhibit none of the characteristic artifacts of previous methods like normal thresholding. Further, the one-rings we reconstruct are sufficiently consistent as to support a mesh-based feature-identification algorithm with minimal modifications.

Our algorithm finds a point-set surface's geometric silhouette in terms of the point samples themselves. We would prefer to find a set of closed arcs, such as those available on a closed triangle mesh; however, our lack of globally consistent connectivity makes this

a daunting problem. We can provide an approximate solution by identifying *consensus silhouette edges*: we say that pq is a consensus silhouette edge from p_s if it is present and on the silhouette from p_s in the local umbrella of both p and q. This is sufficient to correctly render almost all silhouette edges in most input models, but does not guarantee that the resulting silhoeutte loops obey the properties identified in [1].

1.5 Discussion

The three individual contributions described above share a focus on the *tangential planes* in the input model, whether they seek to exploit or approximate them. This focus allows us to tackle a well-studied and fundamental problem from a perspective that often bears unexpected fruit. In the next chapter we will see that silhouettes are often defined and used in terms of these tangential planes, laying the foundation for and motivating the contributions of later chapters.

Chapter 2

Background

In this thesis we discuss algorithms for computing, updating, and exploiting the object-space silhouette of a 3D surface model as briefly defined in Chapter 1. Before we build on the intuitions presented earlier to obtain a more principled understanding of the structures involved, we must first define our terms. This is particularly important as the idea of "silhouette" has a strong intuitive meaning to most observers; however, without precise language this intuition often leads to confusion.

Once this foundation is established, we motivate further work in the field by presenting theoretical lower bounds on the complexity of an object's silhouette, arguing that its sublinear size and coherent variation makes it a powerful shape descriptor for geometry processing. We illustrate these properties by presenting existing methods, noting their strengths and weaknesses, and identifying room for improvement. Next, we motivate the silhouette's utility for geometric computing by identifying a number of applications that take advantage of silhouette information on a model or in a scene.

At the end of this chapter, we present a brief overview of the *Hough transform* in the context of image processing. While this powerful conceptual tool is only glancingly related to image-based silhouette extraction, it forms the core of the plane-based transform methods used in the original work presented in chapters to come.

2.1 Silhouettes

Silhouettes on objects are always relative to a given point (under perspective projection) or direction (under orthographic projection). Since silhouettes carry so much perceptual

meaning, we generally describe them relative to the *camera* or *viewpoint*; however, for many geometry-processing applications this need not be the case and may be misleading. We instead use the term *silhouette point* to describe the generating point for a silhouette.

Definition 2.1.1 (Silhouette point, silhouette vector)**.** The *silhouette point* p_s is an arbitrary point in space from which the perspective silhouette of an object can be defined. The *silhouette vector* v_s is the constant equivalent of the silhouette point under orthographic projection. We overload the notation as follows: Under pespective projection, relative to a given point p, the *silhouette vector* $v_s(p)$ is $p - p_s$.

When the silhouette point is in motion, we express its position at a given time t as $p_s(t)$.

On smooth surfaces without boundary, we define the silhouette as the set of points separating front-facing regions from back-facing regions. This corresponds to the transparent visible rim from Figure 1.1; the other "silhouettes" in that figure can be derived from this definition. We can determine the facing of a point on the surface relative to p_s by examining the dot product of its normal with v_s; front-facing points will have a positive dot product, while back-facing points will have a negative dot product. It follows that points on the silhouette will have a zero dot product.

Definition 2.1.2 (Silhouette of a continuous surface)**.** The perspective silhouette of a smooth surface model M with respect to a point p_s is the set of all points $p \in M$, each with normal vector n_p, such that $n_p \cdot v_s(p) = 0$. In the orthographic case we omit p_s and this simplifies to the set of points with $n_p \cdot v_s = 0$.

Most silhouette algorithms operate on polygon meshes, which may be closed but are not smooth. In particular, mesh silhouettes are sets of edges, and the normal vector of an edge is not well defined. In this case we rely upon the separation property of silhouettes for a definition:

Definition 2.1.3 (Silhouette of a polygon mesh)**.** Let M be a polygon mesh such that each edge $e \in M$ has adjacent faces f_a and f_b with normal vectors n_a and n_b respectively. Then the perspective silhouette of M with respect to p_s is the set of all edges $e \in M$ such that $(n_a \cdot v_s(p_a)) \times (n_b \cdot v_s(p_b)) < 0$ for arbitrarily-chosen points $p_a \in f_a$ and $p_b \in f_b$. Equivalently we may set π_a (resp. π_b) to be the supporting plane of f_a (resp. f_b) and say that edge e is on the silhouette if p_s is within the negative half-space of π_a and the positive half-space of π_b, or vice versa.

In the orthographic case we may omit p_a and p_b and say that edge e is on the silhouette iff $(n_a \cdot v_s) \times (n_b \cdot v_s) < 0$.

Now that we have defined the silhouette of two types of closed surfaces, we can address the other components of Figure 1.1.

Definition 2.1.4 (Visible silhouette and contour)**.** Let S be the perspective silhouette of a closed surface model M from a point p_s. Then the *visible silhouette* of M is the set of points $p \in S$ such that for any other point $q \neq p, q \in M$ on the ray $(p_s, v_s(p))$, we have $\|p - p_s\| < \|q - p_s\|$, as shown in Figure 1.1(b). The *contour* of M is the set of points $p \in S$ such that p is the only point in M on this ray, as shown in Figure 1.1(a).

Rather than consider the silhouette as a set of points or edges on the model generated by a given silhouette point, it can be instructive to consider the primitives themselves in terms of the set of potential p_ss that would put them on the silhouette. We call this the *silhouette-generating set* or *SGS* of that primitive:

Definition 2.1.5 (Silhouette-generating set)**.** Let M be a closed smooth surface model. The silhouette-generating set of a point $p \in M$ is the set of all points in p's tangent plane.

Let M be a closed polygon mesh. The silhouette-generating set of an edge $e \in M$ with adjacent faces f_a and f_b (with supporting planes π_a and π_b) is the union of the set of all points in the negative half-space of π_a and the positive half-space of π_b with the set of all points in the negative half-space of π_b and the positive half-space of π_a.

2.1.1 Silhouette complexity

Before we begin to study silhouette algorithms, we should get at least an intuition of the complexity of the silhouette. If the silhouette of a mesh with n faces is $\Theta(n)$, we should forget about asymptotic efficiency in our algorithms and strive for other things, like proper use of cache memory, to see gains in speed. If on the other hand the silhouette is often much *less* complex than the mesh, an asymptotically efficient algorithm, even one with a high constant, is far more appealing than even the leanest brute-force methods. Here we consider silhouette complexity (or length) in terms of the number of edges involved; this will often be related to the geometric length of the silhouette by constraints on edge length.

The silhouette of an arbitrary mesh from an arbitrary point of view is highly variable and difficult to analyze. Most theoretical results are proven only on subsets of the sorts

of models we'd like to work on, and more general results are based on empirical evidence rather than theoretical guarantees. However, there is enough research available to give us hope for the elegant asymptotic perspective.

The simplest case is a mesh approximating a sphere. Kettner and Welzl [49] show that such a mesh with Hausdorff distance ϵ has $\Theta(1/\epsilon)$ edges, and that its silhouette from a random viewpoint at infinity (parallel projection) has $\Theta(1/\sqrt{\epsilon})$ edges.

When we broaden our focus from approximately spherical polyhedra to general (convex) polyhedra, however, the picture becomes much gloomier. Now we find that it is easy to construct a polyhedron with a linear-sized silhouette from some – even from *many* – viewing angles, as described by Alt *et al.* [2]. All is not lost, however: Alt *et al.* also prove that a certain class of polytopes – triangulated polytopes with low aspect ratios, bounded edge lengths, and some ϵ such that all but $O(\sqrt{n})$ incident edges form an angle between ϵ and $\pi - \epsilon$ have silhouettes of complexity $O(\sqrt{n})$, with a constant factor that depends on ϵ. They also establish that if we remove the ϵ constraint and instead insist that the number of almost-collinear edges is bounded, we can achieve silhouettes of complexity $O(n^{2/3})$ in the worst case under parallel projection. These bounds cover only convex polyhedra and operate under constraints far more stringent than any we would like to consider, but they do show that sublinear silhouettes are achievable.

If we consider the *average* silhouette length rather than the worst case, the outlook brightens significantly. Glisse and Lazard [32] are able to show that the *expected* size of a polyhedron's silhouette is $O(\sqrt{n})$ under a much more general set of constraints than those put forth by Alt *et al.*. Glisse and Lazard do not require convex polyhedra or bounds on the polyhedron's aspect ratio: instead, they require that the input mesh has edges of bounded (*i.e.* not arbitrarily small) length, that its *faces* have low aspect ratios, and that it approximates an underlying smooth surface in terms of positions and normals. This excludes inputs such as Schwartz lanterns and certain algebraic surfaces, but encompasses the vast majority of input models.

Where Glisse and Lazard consider arbitrary polyhedra conforming to a theoretical model, McGuire [60] performs an empirical study on 897 triangle meshes downloaded from the internet. His data set contains models of man-made shapes with varying complexity, from a simple tetrahedron to a million-triangle Buddha mesh, and includes a number of models with small cracks or holes that do not satisfy the closed 2-manifold assumption normally put forth. He computes the average silhouette size for these meshes by choosing a large

number of p_ss at widely varying distances from the mesh, and computes an overall empirical complexity estimate of $O(n^{0.8})$. McGuire concludes that the self-similarity and fine surface detail on these artificial subjects accounts for the added complexity (while Glisse and Lazard blame sampling nonuniformity around sharp features), and points to the higher-than-expected exponent as "evidence of a fractal nature for man-made objects". While the fractal dimension of a silhouette curve is of potential future interest, we are content to affirm that silhouette complexity is sublinear in the size of the input surface.

Rather than calculate the full silhouette every frame, we would like to be able to simply find the changes in the silhouette under smooth camera movement (that is, exploiting spatial and temporal coherence). Here, the literature is less helpful. Efrat *et al.* [26] consider this problem, though rather than a single polygon mesh they consider a scene of k *convex* polyhedra with n edges total. They show that the *silhouette arrangement* – the arrangement of all silhouette arcs projected onto the display plane – of such a scene involves $\Theta(kn)$ segments, a bound that increases to $\Theta(k^2n)$ when the viewpoint moves linearly and $\Theta(kn^2)$ when it moves along an algebraic curve. These bounds and the construction on which they are based is somewhat removed from our consideration of silhouette update as a mesh-geometry problem, but they give some insight into the difficulties behind visibility processing and the connection between the visibility and silhouette problems.

2.1.2 Image-space silhouette methods

Image-space silhouette algorithms produce a set of pixels corresponding to the projection of a mesh's visible silhouette edges, rather than identifying edges on the mesh geometry itself. In this way they are primarily suitable for non-photorealistic rendering problems, in which only the visible rim of the object is desired. Image-space methods are also typically quite fast, and some can easily be implemented as fragment shaders on the GPU. They are often robust to errors or inconsistencies in the underlying surface representation: if the rendered image is acceptable, image-space silhouette results will likely be acceptable as well. For example, Deussen and Strothotte [20] perform pen-and-ink rendering of "primitive soup" tree models using an image-space technique.

Pure image-space methods apply edge-detection algorithms such as the Sobel operator to the depth buffer, as shown by Saito and Takahashi [73], or an auxiliary normal buffer as shown by Hertzmann [38]. Local edge-detection methods are easier to implement as fragment programs than global methods like the Hough transform. However, since the

silhouette is only ever present as a collection of pixels in an image buffer, it is difficult to extract silhouette geometry from an image-space method. Even for non-photorealistic rendering applications, where only the pixels of the silhouette are of interest, it is difficult to stylize the results of pure image-space methods.

Hybrid image-space algorithms provide an alternative to the above, with more user control over the resulting rendered silhouettes at the cost of increased complexity. In these methods, extra silhouette geometry is rendered along with the model itself, and depth or stencil testing is used to eliminate any "silhouette pixels" that do not correspond to visible silhouettes. For example, Raskar and Cohen [71] develop a method which first fills the depth buffer with the z values of front-facing polygons, then enlarge back-facing polygons and render them in the desired silhouette colour; the depth test culls away any silhouette pixels which would be occluded by the mesh. By varying the scaling of back-facing polygons, the thickness of resulting silhouette lines can be controlled.

Image-space silhouette algorithms are surveyed in more detail by Isenberg [45]. Since they are poorly suited to most geometry processing applications, we direct the interested reader to that work and cover object-space methods in more detail.

2.1.3 Object-space silhouette methods

Object-space silhouette algorithms operate on the geometry of the mesh itself, rather than on a set of pixels corresponding to the projection of the visible part of the mesh. This makes them inherently more precise than image-space methods, and exposes the global structure of the silhouette rather than limiting the algorithm to the visible subset. However, restricting object-space silhouette results to the visible subset of the silhouette requires either an occlusion culling step or a hybrid algorithm.

The cost of an object-space algorithm depends on the number of operations being performed (in asymptotic and absolute complexity) and the cost of each operation. In computational terms this is dominated by viewpoint facing checks and, for more complex data structures, intermediate bounding volume checks. These operations are similar in cost and can be treated as equivalent. At runtime, memory architecture issues such as cache coherence may have a significant effect on performance; however, this issue has not been emphasized in the existing literature.

The naive approach to silhouette extraction, which we call the *brute force* method, simply iterates through all edges on the mesh and checks their adjacent faces, marking the

edge as part of the silhouette iff it is adjacent to one front-facing and one back-facing face. On a triangle mesh, this will test the facing of each triangle three times – once for each of its edges.

Buchanan and Sousa [12] develop the *edge buffer* data structure. This consists of a bitfield associated with each edge, and a set of pointers into the bitfield array on each face. Rather than iterate through the mesh's edges, the edge buffer algorithm iterates through each face, tests its facing once, and sets all of its associated edge flags. A second pass through the edge buffer, which can be done as part of a rendering step, identifies silhouette edges as those with both front- and back-facing flags set. The edge buffer method shares the brute-force method's $O(n)$ asymptotic complexity, but performs only one-third as many facing checks and requires fewer random memory accesses.

Sander *et al.* [75] take a different approach. In this work, they approximate the fully front- and back-facing regions of a mesh edge by a pair of anchored cones. They then build a search tree by combining edges with similar cone pairs into interior nodes; each interior node of the tree has a pair of anchored cones which approximates the *intersection* of its children's cones. This produces a conservative estimate of viewpoint positions for which no edges in the subtree will be on the silhouette. At runtime, their algorithm simply culls away subtrees whose cones contain the viewpoint; this step is shown empirically to be sublinear in the size of the input mesh. However, they find that the preprocessing step is particularly expensive.

Rather than build a precomputed data structure for silhouette extraction, Markosian *et al.* [59] present a randomized algorithm. Their algorithm begins by selecting a random subset of the mesh's edges and checking their silhouette status. If any silhouette edges are found, the algorithm checks their neighbours, following these recursively until a silhouette loop has been constructed. They assign selection probabilities to edges in proportion to the edge's dihedral angle, so that sharp edges (which are more likely to be on the silhouette) are more likely to be checked than shallow ones.

If the silhouette point's movement is sufficiently smooth, Markosian *et al.* exploit the silhouette's spatial and temporal coherence by first checking every silhouette edge of the previous frame, and performing random local neighbourhood checks around previous silhouette edges. These methods are very effective at detecting movement of existing silhouette arcs, but have difficulty detecting new silhouette loops (which are likely to be small, and therefore are unlikely to be found by random edge checks).

Transform-based silhouette algorithms

One of the key challenges to performing silhouette extraction directly on the target mesh is detecting topological changes in the silhouette. In particular, new silhouette loops created by a small viewpoint change may be arbitrarily far from existing silhouette edges. This problem can be addressed by performing silhouette extraction in a *transform space*, in which primitives that indicate changes to the silhouette are clustered together regardless of their position on the primal mesh. Such transform-based algorithms admit much more straightforward acceleration structures than the dual-cone approximation tree from Sander *et al.* [75].

Both Benichou and Elber [8] and Gooch *et al.* [33] perform orthographic silhouette extraction on the Gaussian sphere. They represent the supporting plane of each polygon in the mesh by its normal vector; thus, an edge is represented by a great-circle arc on the Gaussian sphere between the normals of its adjacent faces, with length equal to its dihedral angle. The silhouette vector is represented in the transform space by a plane passing through the centre of the sphere with normal equal to the silhouette vector. Any edge arc which intersects this plane corresponds to a silhouette edge. To avoid testing every arc against the plane, Gooch *et al.* build a hierarchy of spherical triangles, while Benichou and Elber project the Gaussian sphere onto a cube and decompose the cube's sides into grids. Each arc on the sphere is transformed into a set of line segments on the cube's faces.

Perspective silhouettes present a challenge in that the view vector is not constant; thus, faces' supporting planes cannot be represented only by their normal vectors. Hertzmann and Zorin [39] represent each supporting plane $\pi = ax+by+cz+d = 0$ by the homogeneous point $p = (a, b, c, d)$. They store these points in a set of eight octrees corresponding to the faces of the four-dimensional hypercube. Rather than connect faces adjacent to each edge, their algorithm links faces adjacent to shared vertices. Finally, they use the octree to find the intersection of the dual plane $\pi(x) = p_s \cdot x = 0$ with this surface, where p_s is expressed in homogeneous form. This is essentially an extension of the Gaussian sphere method to homogeneous coordinates.

Rather than focus on static silhouette extraction, Pop *et al.* [69] present an algorithm for efficient silhouette *updates*. They apply the geometric dual transform, which takes a plane $\pi = ax + by + cz + d = 0$ to a point $p = (a/d, b/d, c/d)$, to the supporting planes of every face in the mesh. When the silhouette point moves from frame t to frame $t + 1$, they

transform each point into a plane $p_s(t) \cdot \vec{x} + 1 = 0$ (resp. $p_s(t+1)$). These two planes form a *double wedge* in dual space; dual points in this wedge correspond to faces whose supporting planes were crossed by the viewpoint. Edges which have joined or left the silhouette will be found on one of these faces. Pop *et al.* do not develop a dual-space silhouette extraction method; their algorithm must be initialized by a complete silhouette from another method.

2.1.4 Hardware-accelerated silhouette algorithms

One of the key challenges to the incorporation of silhouette-based geometry processing techniques into real-time rendering systems is the fact that most of the rendering process is performed on the GPU rather than in main memory. The overhead of maintaining parallel sets of geometry in both main and GPU memory, computing silhouettes on the CPU, and transferring silhouette geometry to the GPU for rendering is often considered unacceptable. Thus, there has been significant interest in adapting the above algorithms to the GPU.

Isenberg *et al.* [45] note that image-space methods like the Sobel operator can easily be implemented as fragment programs and incorporated into the rendering pipeline. If only fragments on the visible silhouette of an object are required, this is an efficient and straightforward option; however, it is still subject to the disadvantages of image-space approaches.

Geometry processing on the GPU is characterized by streaming computation: in general, vertices are processed separately and in parallel, making the connectivity information required for silhouette extraction difficult to represent. Furthermore, mesh connectivity has been fixed and immutable until the introduction of geometry shaders by the Shader Model 4 standard [62]; platforms that do not support geometry shaders must include all potential silhouette geometry on the input mesh, incurring additional overhead. Nevertheless, several GPU-based object-space algorithms have been presented.

Card and Mitchell [14] propose a scheme in where each edge in a mesh is represented by an explicitly stored quad. Each vertex in the quad is augmented with normal information for the corresponding edge's adjacent faces. In a vertex shader program, the face normals of these "edge" vertices are tested independently against v_s. If the vertex's edge is on the silhouette, two of the quad's vertices will be displaced along the bisecting normal to generate a thin, visible quad. If the edge is not on the silhouette, those vertices are not displaced and the "quad" (now degenerate) is culled by the depth test. A scalar parameter is used to control the thickness of the resulting lines. The authors show that their algorithm produces more consistent and controllable results than a comparable image-space method.

Brabec and Seidel [10] use a similar approach to compute stencil shadow volumes on the GPU. They augment the data representation with vertex-ordering information, and compute silhouette data in a point-based rendering pass rather than a vertex program, but the essential details of their method are similar. Again, their method generates a quad for each edge and renders it as part of the shadow volume if its edge is found to be on the silhouette.

With the addition of geometry shaders to Shader Model 4-capable GPUs, GPU-based silhouette extraction no longer requires explicit geometry to be precomputed and stored per-edge. However, some method of preserving connectivity information still needs to be incorporated into any algorithm. Sander *et al.* [74] compute an optimized covering of a triangle mesh by *adjacency primitives* in a way that minimizes vertex cache misses. This allows them to generate shadow volume geometry in the geometry shader with significantly less overhead than the DirectX 10 sample program [62]. They also adapt their method to stylized rendering.

Since GPUs reward cache-coherent, easily parallelized streaming algorithms, little work has been done to adapt the precomputed methods at the forefront of CPU-based silhouette algorithms. However, work such as that by Greß *et al.* [34] shows that hierarchical data structures and search algorithms can be implemented on GPUs and provide significant performance improvements.

2.1.5 Silhouettes on non-mesh surfaces

While polygon meshes are at present the most common surface representation for geometry processing, they are not the only option available. Silhouette algorithms for several other surface representations have been presented in the literature, which we review below.

Smooth and free-form surfaces

Silhouette computation on free-form surfaces is generally done by selective subdivision and refinement on a more tractable representation of the surface, rather than on the surface itself. Rather than formulate the problem in terms of separation of front- and back-facing surfaces, free-form silhouette algorithms typically define their target as the set of points whose normal is orthogonal to the view vector, as in Definition 2.1.2. In this way we can see these algorithms as level-set methods on the surface.

Elber and Cohen [27] find silhouette curves on NURBS patches as part of a hidden curve removal algorithm. They extract piecewise-linear approximations of the silhouette curves in the parameter domain of the patch by adaptive subdivision following a quadtree-like scheme, up to a given tolerance. The resulting curve is accurate at its vertices, and is subdivided again by linear bisection in $\mathbb{R}^3$ until the tolerance is met at the midpoints.

Hertzmann and Zorin [39] also propose a method for finding silhouettes of subdivision surfaces. They use a level-set method similar to that of Elber and Cohen to find curves whose tangent planes contain v_s; however, rather than perform this operation in the parameter domain, they operate directly on the surface itself, subdividing where necessary. Note that the silhouette curve generated by their method need not be constrained to the edges of the surface's final tesselation, but is again accurate at its own vertices.

Point-set surfaces

Point-set surfaces, or point clouds, are increasingly popular surface representations, particularly for data acquired from laser or photometric scanners. Finding the silhouette of a point-set surface is therefore a compelling problem, but since these surfaces are not compact it is difficult to apply the usual definitions to them.

Zakaria and Seidel [90] relax Definition 2.1.2 to find points corresponding to the silhouette of a point-set surface. Rather than select points where $n_p \cdot v_s(p) = 0$, they normalize $v_s(p)$ and select points where the dot product falls below a fixed threshold. They then render the points with unique colour values to an image buffer, and use the set of visible coloured fragments to identify a subset of points on the surface corresponding to a silhouette arc. This method works well for their chosen application of stylized rendering, but has difficulty identifying the full silhouette in regions of high curvature. Silhouette extraction by thresholding the n_p-$v_s(p)$ dot product also has trouble with regions of low curvature, within which a large set of spurious points may be identified. When p_s is at the viewpoint, as in stylized rendering, this is rarely noticeable. Figure 5.10 in Section 5.4 gives several examples of the difficulties with normal thresholding.

Rather than adapt an existing geometric definition to point-set surfaces, Xu *et al.* [87] exploit the adaptability of image-space methods. Their method is similar in principle to the hybrid algorithm of Raskar and Cohen, in that they render oversized splats to produce silhouette pixels, then use depth culling while rendering the point-set surface normally to

overwrite interior fragments. The result is easily implemented and produces compelling results, but shares the limitations of other image-space approaches.

While not strictly a silhouette algorithm, Katz *et al.* [48] describe an elegant method for computing visibility on a point-set surface in based on the convex hull of a transformed point cloud. They define a *hidden point removal* operator which inverts points in the cloud about a sphere centred at p_s and bounded by the point-set surface, and show that the convex hull of this inverted point set, plus p_s, contains the "visible" points. Thus, points on the silhouette of this convex hull are on the contour of the input point set.

Implicit surfaces

While implicit surfaces are often used to model organic shapes, they are generally triangulated before rendering; hence, standard object-space silhouette algorithms can be applied. However, Tsai *et al.* [80] develop a visibility framework based on implicit ray tracing on a global signed distance function represented on a multiresolution grid. Their method produces visible silhouettes in voxel form as a side effect, and can be modified to produce full silhouette information.

Isosurfaces in volume data can be thought of as implicit surfaces. Burns *et al.* [13] develop a method for extracting silhouettes along isosurfaces in medical volume data using a zero-set method similar to that of Elber and Cohen, and exploiting spatial and temporal coherence in much the same way as the randomized algorithm of Markosian *et al.* by looking for new silhouette arcs near those of the previous frame. They argue that their method runs in $O(n)$ time on an n^3 volume. However, they admit that due to its randomized rather than exhaustive nature their algorithm cannot find all silhouette loops.

2.2 Silhouette applications

As silhouette loops incorporate a great deal of information about how the geometry of an object relates to the silhouette point p_s, they are vital for a number of applications in rendering and geometry processing. We describe several classes of these applications below, with particular emphasis on the use and modification of object-space silhouette methods.

2.2.1 Stylized rendering

Stylized rendering is the classic motivating application for silhouette algorithms. Many researchers, including Markosian *et al.* [59], Burns *et al.* [13], Hertzmann and Zorin [39], and Zakaria and Seidel [90], have developed novel and important silhouette-extraction methods as part of stylized rendering applications. These applications often seek to enhance visual comprehension of a model by emphasizing or exaggerating perceptually important features, as in Burns *et al.*. Others, such as Markosian *et al.*, endeavour to recreate artistic rendering styles on the computer.

Silhouettes – particularly visible silhouettes – are important for stylized rendering as they offer powerful perceptual cues to shape and geometry. Koenderink [52] notes that the human visual system often interprets complex spatial configurations correctly even when the viewer holds primitive or incorrect conscious interpretations of the scene. He explains that the visible silhouette provides the viewer with information regarding the curvature and structure of the shape being observed. Silhouette arcs therefore belong to a larger set of more generic *feature lines*, used in stylized rendering to convey shape and surface properties; however, while silhouette computation is relatively efficient, the construction of many types of more general feature lines depends on a deeper curvature analysis which is significantly more expensive, as shown for example by Kalogerakis *et al.* [47].

As mentioned in Section 2.1, the primary benefit of object-space silhouette algorithms to stylized rendering is the flexibility it grants for line rendering. While image-space algorithms produce a set of pixels already in place in the image buffer, object-space algorithms produce line strips which may be textured, extruded to different and varying thicknesses, jittered or deformed, or otherwise postprocessed before rendering. However, the dependence of these line strips upon the connectivity of the underlying mesh can make the mesh silhouettes diverge unacceptably from the ideal silhouette arcs of a smooth surface being approximated. Hertzmann and Zorin [39] demonstrate this phenomenon. They correct for it by computing subpolygon silhouettes on the mesh using the same algorithm used for subdivision surfaces and identifying cusps where the silhouette arc's tangent vector intersects p_s.

Rather than create new silhouette loops which are independent of an underlying mesh's connectivity, Brosz *et al.* [11] propose a method in which adds mesh edges near the silhouette to a "silhouette complex". These edges are weighted based on their stability, where edges on the geometric silhouette and far from the silhouette are considered stable, but edges near

the silhouette (which may join it under a small displacement of p_s) are not. The second group of edges are partially stylized based on their stability measure. This is a form of edge-based antialiasing; it is coherent across movement of p_s and smooth mesh animations.

Kirsanov *et al.* [51] identify the presence of extremely small silhouette loops and high-frequency details as impediments to stylized rendering of silhouette arcs. they develop two methods to produce "simple" silhouette loops on high-resolution meshes. In both cases they identify detailed silhouette loops on a high-resolution mesh with the simpler topology and geometry of corresponding loops on a coarse mesh of the same model. In the "loop picking" method, they select loops from the fine mesh which have small geometric distance from the coarse mesh's silhouette. In the alternative "loop mapping" method, they map the edges of the fine mesh onto the coarse mesh and pick a new loop on the former corresponding to the silhouette extracted on the latter. This both retains all major features of the silhouette and ensures spatial coherence.

2.2.2 Shadow rendering

Silhouettes are closely connected with volumetric shadow techniques. In fact, two major methods for rendering shadows correspond closely to image-space and object-space silhouette methods. Shadow maps, introduced by Williams [84], compute a depth image of the scene from the perspective of the light source. This image is then projectively mapped onto surfaces in the scene and each fragment's distance to the light source is compared with the projected value mapped onto them; fragments further from the light source than the mapped value are in shadow. This method of checking for mismatches based on a depth value is analogous to depth buffer-based image-space silhouette detection as described in Section 2.1.2 in that shadow borders correspond to discontinuities in the shadow map.

As with image-space silhouette algorithms, shadow mapping is conceptually simple and efficiently implemented in hardware. However, the limited resolution of shadow maps often results in one shadow-map fragment being projected onto a large screen space, incurring obtrusive aliasing artefacts. Much recent study of shadow mapping involves the application of powerful statistical techniques to adaptively refine or antialias these border fragments; see for example the work of Annen *et al.* [4] among many others.

Shadow volumes were first developed by Crow [18] and are analogous to object-space silhouette extraction. In these methods, the shadow-casting object's silhouette edges are found with respect to the light source. Silhouette edges are then extruded to infinity along

the plane containing the edge and p_s; we say that silhouette loops are extruded into *shadow frusta*. For static light sources and shadow casters this geometry may be integrated into the scene, for example using the BSP tree approach described by Chin and Feiner [15] to clip scene polygons against shadow geometry.

For real-time rendering of animated scenes, however, most practitioners use a hybrid approach involving the stencil buffer. After scene geometry is rendered to the depth buffer, polygons in the shadow frusta are rendered to the stencil buffer. Front-facing shadow polygons increment the stencil value, while back-facing shadow polygons decrement it. This leaves a nonzero value in stencil buffer fragments corresponding to shadowed geometry, which can then be correctly lit in a fragment shader. This method was developed by Heidmann [37] and refined by Hornus *et al.* [40]. Note that since the shadow borders are represented geometrically they are as precise as the geometry from which they were generated.

Shadow volume algorithms have two disadvantages compared to shadow maps for real-time rendering. First, until recent work such as that by Sander *et al.* [74], shadow geometry either had to be created on the CPU and transferred to the GPU on a frame-by-frame basis or, as done by Brabec and Seidel [10], incorporated into the model itself at a significant space penalty. Second, rendering shadow geometry to the stencil buffer imposes a substantial fill-rate penalty even when little geometry is actually in shadow. Thus, while many games have successfully implemented stencil shadow volumes, shadow mapping is the dominant paradigm on the current generation of graphics hardware.

Both of the above methods have focused on point light sources, which generate hard-edged shadows with no penumbra. Shadow volume methods are also adaptable to area light sources, which feature a penumbral region that is shaded from some but not all of the light source. Assarsson and Akenine-Möller [6] present a hybrid algorithm which uses silhouette edges (from a single point p_s placed at the centre of the light source) to calculate stable penumbral wedges from an area light source, then renders these wedges to a visibility mask used in a final shading step. This method is shown to be suitable for hardware implementation, and is further refined by the authors in later work [7].

A more precise method suitable for offline rendering is developed by Laine *et al.* [54] and refined by Lehtinen *et al.* [57]. These methods explicitly generate penumbral wedges from light source borders and store them in an acceleration structure, greatly increasing the speed of a soft shadow ray-tracing implementation. The authors note that their silhouette-based

geometric shadow implementation can be used as a "black-box shadow solver" in any offline renderer without incurring additional rendering passes.

2.2.3 Visibility determination

Visibility determination is closely related to shadow rendering; for example, the soft shadow method of Lehtinen *et al.* described at the end of Section 2.2.2 can be recast as a from-region visibility oracle without modification. Laine [53] notes that such a from-region visibility solver is a prerequisite to building a more structured viewcell visibility graph. Visibility in general is a broadly-defined problem with an extensive literature; see for example Bittner and Wonka's survey of the literature [9].

A subproblem of visibility determination known as hidden line removal is a significant component of many stylized-rendering methods, such as that presented by Markosian *et al.* [59]. In these circumstances it is usually the visibility of points on the silhouette that is being determined, rather than the silhouette being used to aid in visibility determination; however, Elber and Cohen [27] use silhouette arcs on restricted freeform surfaces to bound line visibility and extend this method to building visibility maps in a later work [28].

Silhouettes are particularly significant to the construction of the aspect graph developed by Gigus and Malik [31] and the 3D visibility complex of Durand *et al.* [23]. These methods essentially compute a subdivision of space around a scene such that when p_s moves only within a region represented by a leaf node, the topology of the silhouette arrangement does not change. These structures are generally prohibitively large, however, and are largely restricted to theoretical study for complex scenes.

Silhouette information can also be used to accelerate less comprehensive visibility algorithms. Given that discontinuities in the visible set occur at silhouette edges, sampling-based algorithms can increase density near a scene's silhouette arcs and save a great deal of effort. For example, Wonka *et al.* [85] describe a from-region visibility algorithm in which distributes sample rays pseudorandomly in general but focuses on depth discontinuities – namely, silhouettes. The from-region visibility problem is particularly well-suited to the transform-based algorithms described in Section 2.1.3, as an incremental sweep of p_s satisfies the spatial coherence assumptions of those algorithms and permits incremental update rather than full recomputation of the silhouette.

In the 2.5D case of heightfield rendering, silhouettes become horizons and can be used directly to perform occlusion culling. Archambault *et al.* [5] report complexity results and

describe an output-sensitive query structure for computing the set of horizon edges of a terrain from a fixed v_s. This is essentially the hidden line removal problem in a somewhat restricted domain. In the perspective case, Lloyd and Egbert [58] describe a straightforward and effective method for occlusion culling in hierarchical terrains based on the horizons of visible terrain cells. For their quadtree-based terrain renderer, the silhouette of each rendered node is simplified to the silhouette of its highest enclosed plane; the arcs extracted from these are then projected away from the viewer in much the same way as Crow's shadow volume frusta [18]. Potentially occluded nodes are then culled against these simple projected frusta.

2.2.4 Model capture and registration

Model capture and registration are related problems in which one or more 2D images of an object are brought into correspondence with a 3D surface model. In the model capture problem, a series of images is used to create a synthetic model of the object being studied. This is often done by silhouette carving, in which the contours of the objects are extracted from each image and a model conforming to those contours is produced. Model registration, on the other hand, begins with a 3D model of the object as well as 2D images, and transforms the model to fit the position and orientation described in the images. The literature on this topic is extensive; we will cover only a few representative and illuminating applications.

Pop *et al.* [69] describe a silhouette-based algorithm for image registration of medical models onto X-ray images. They begin by finding edges in the input image, then use a similarity measure based on the computed silhouette of the model to find a rigid-body transform that aligns it with the X-ray image. Their incremental silhouette update method is well-suited to incremental evaluation of this similarity measure as many transforms are attempted. They note that edges in an X-ray image can be caused by density gradients within structures as well as structural boundaries, so their silhouette-based alignment method is used as a first approximation to present a slower method with a suitable starting point.

Plaenkers and Fua [66] develop a method for registering a metaball-based model against images of humans. They match the silhouette of their 3D implicit model against the contour of a stereo image provided as input, using both stereo data and silhouette fitting to produce a model with high fidelity to the input by tuning the parameters of the metaballs.

More ambitiously, Vlasic *et al.* [83] develop a template-based method for animation capture. They use a template mesh with an articulated skeleton and fit the vertices of the

mesh to visible silhouette pixels from a set of multi-angle video streams. They sample the visible silhouette on each frame of the videos and deform the template's silhouette vertices to match it. The template mesh allows them to capture geometry correctly in the case of self-occlusion, particularly with loose clothing, where silhouette carving would fail.

Jones and Oakley [46] present a method for image-set registration in which is based on the consistency of image-space contours of an object. They place a rigid object on a turntable and capture a set of 2D contours as the turntable is rotated, then use the geometry of these contours to estimate changes in the orientation and position of the object between two images. In this setting no 3D model is created explicitly: the model's geometry is represented entirely in terms of the extracted contours. It can thus be seen as a step between model registration and model capture, and elegantly demonstrates the surprising amount of information that can be extracted even from an incomplete silhouette.

Among the silhouette carving literature, the development of projective visual hulls by Lazebnik *et al.* [55] is particularly relevant. They characterize the surface of an object's visual hull as a generalized polyhedron whose faces are patches of the visual cone – a construct which roughly corresponds to the volume between p_s and the shadow frustum of the object from p_s. By assigning an orientation to the contour loops generated from a set of input images they develop an algorithm for computing this visual hull with only weak requirements for camera calibration. As with transform-based silhouette algorithms, their method exploits spatial coherence, and their construction of visual hulls in terms of intersection events between contours bears similarities to the development of aspect graphs and the 3D visibility complex mentioned in Section 2.2.3.

Instead of capturing an existing model, Rivers *et al.* [72] synthesize new 3D geometry from a set of 2D silhouettes. They generate relatively simple geometry by extruding silhouettes in several orthogonal planes and performing CSG operations on the results, exploiting silhouette information to develop a robust and precise 3D CSG algorithm based on the supporting planes of shape features rather than shape volumes. A modified Laplacian smoothing algorithm generates smooth shapes when required while preserving the projections of their silhouettes on the orthogonal input planes.

2.2.5 Guided simplification

Since silhouettes carry so much perceptual information, it is natural to integrate them into mesh simplification schemes. Pop *et al.* [69] describe a variation of progressive meshing

based around their silhouette update method. They preserve detail near silhouette edges while aggressively simplifying other regions of the mesh.

Sander *et al.* [75] take a more drastic approach to silhouette-based mesh decimation. Rather than simplify a progressive mesh nonuniformly, they calculate the mesh's silhouette and render it as a mask to the stencil buffer. They then render a coarse hull of the mesh, clipping it against the high-resolution silhouette in the stencil buffer and using a texture lookup to produce correct normals at each pixel. The result is nearly indistinguishable from a high-resolution mesh.

In a different context, Sillion *et al.* [77] use depth discontinuities to create long-lived impostors. In an urban walkthrough application, they render distant geometry to the depth buffer, then retriangulate the result into a simpler, view-dependent mesh. Their retriangulation places extra detail on the original geometry's silhouette edges to properly recreate depth and contour details. The result is a three-dimensional impostor which remains valid for much longer than traditional billboard impostors.

2.3 The Hough transform for geometry processing

Introduced by Hough [41] for machine analysis of photographs, the Hough transform maps sample points in a primal space to a set of points in parameter space corresponding to the structures which they intersect. For example, a 2D edge-detection method based on the Hough transform would map each point in image space to the set of lines passing through it in parameter space as part of a voting scheme. Tuples in parameter space with the most votes are then selected as supporting lines of edges in the image. Figure 2.1 gives an example of this process.

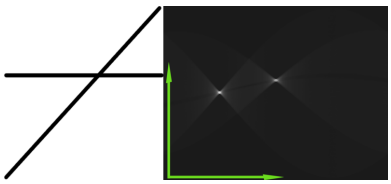


Figure 2.1: Left: two edges in a bitmap image. Right: vote counts in Hough space summed over all black image pixels. The two peaks correspond to the parameters of the two lines. Used under license from Wikipedia Commons [65].

For a more detailed treatment of the Hough transform, its generalizations, and the relationship between bin shapes, parameter domains, and captured geometry, see the excellent presentation by Princen *et al.* [70].

2.3.1 Geometric applications of the Hough transform

Where the Hough transform maps individual pixels into a parameter space where they vote for a structure which globally best fits them, some mesh-processing applications do the same with the vertices of a mesh. Décoret *et al.* [19] use such a voting scheme to create *billboard clouds*, sets of textured planes which represent a much more complex static mesh. Each vertex on the mesh casts votes for the planes which contain it; when every vertex has voted, a number of peaks are selected and the high-resolution mesh is rendered onto them to produce texture and transparency maps.

Wu and Kobbelt [86] use a similar scheme to build approximate signed distance fields of high-resolution meshes. In this case, supporting planes in primal space are projected to points in Hough space, which are used to define a set of undesirable splitting planes for a BSP tree. The signed distance field is then defined on the nodes of the BSP tree, whose splitting planes represent its zero set.

Finally, Zaharia and Prêteux [89] use the Hough transform of a mesh as a shape descriptor for mesh retrieval. They discretize parameter space in the usual way, but allow a set of flipping operators to aid alignment of corresponding meshes with different principal component axes. The meshes' Hough transforms, rather than their geometry, are compared and a similarity measure computed.

2.3.2 Overview of the geometric Hough transform

Given a plane $\pi : ax + by + cz - d = 0$ with normal (a, b, c) having unit length, if π does not pass through the origin, then its dual, denoted by $\mathcal{D}(\pi)$, is the 3D point $(a/d, b/d, c/d)$; otherwise, $\mathcal{D}(\pi)$ is the point at infinity. Conversely, given any 3D point (a', b', c'), its dual plane is given by $a'x + b'y + c'z = 1$. The 3D Hough transform $\mathcal{H}(\pi)$ of the plane π is the 3D point (ad, bd, cd) with no constraints placed on any of the coefficients.

Geometrically, we construct the 3D Hough transform of a triangle T by drawing a line from the origin perpendicular to the supporting plane of the triangle. The point of intersection H, between the line and the supporting plane, is defined to be

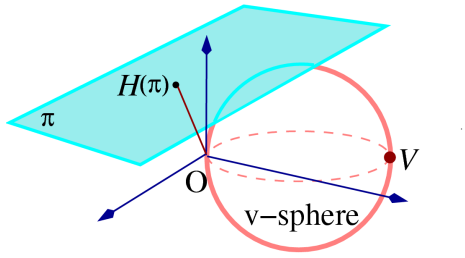


Figure 2.2: 3D Hough transform $\mathcal{H}$ of a triangle T and the v-sphere corresponding to a silhouette point p_s.

the 3D Hough transform of the triangle, as shown in Figure 2.2. Note that all triangles sharing the same support plane, though with opposite orientations, are mapped to the same point in Hough space.

An elegant way to relate the Hough transform to the dual transform is through the notion of inversion with respect to the unit sphere centered at the origin. In general, points P and P' are said to be inverses of each other with respect to a sphere of radius r and centered at O, called the centre of inversion, if P' lies on the ray $\overrightarrow{OP}$ and $|OP| \cdot |OP'| = r^2$. Note that the inverse of a plane not passing through the centre of inversion is a sphere passing through the centre of inversion, and vice versa. It follows that the dual of a point p_s, which is a plane not through the origin, is mapped via inversion to a sphere passing through the origin O in Hough space, as shown in Figure 2.2. When we consider the fact that $\mathcal{H}(\pi) \in \pi$ alongside this inversion process, we see that $\mathcal{D}(\pi)$ can be arbitrarily close to or far from the origin; thus, we can show that the Hough transform produces a point distribution that is better suited to hierarchical acceleration structures.

Chapter 3 compares the dual and Hough transforms in more detail.

2.4 Discussion

We have seen that geometric definitions of the silhouette tend to be expressed in planar terms, using tangent planes on smooth surfaces and double-wedges on meshes; and second that most of the efficient algorithms for finding silhouettes depend upon an efficient organization of *planes*, rather than of vertices or edges. Even hybrid methods such as those of Raskar and Cohen [71] and Zakaria and Seidel [90] rely upon normal information and thus tangent planes. In Chapter 3 we show that improving the organization of planes in a transform space leads to significantly improved silhouette extraction and update performance. Later, in Chapter 5, we use the intuitive notion of silhouettes defined by a collection of planes to drive a local reconstruction algorithm which leads to significantly improved point-cloud silhouettes.

In examining applications of geometric silhouettes, it is therefore not surprising that we find many applications whose essential nature is projective, planar, or both – shadow rendering and visibility determination being notable among them. These algorithms, too, often emphasize planes in their formulation and implementation, as for example in the 3D visibility complex [23]. In Chapter 4, we take a more global view of the component planes of a model, constructing scalar fields which allow us to address several nontrivial problems.

Finally, we examine the Hough transform as a feature-extraction technique. We note that its use in geometry processing shows it to be well-suited to describing simply-parameterized objects like planes, and propose an adaptation called the *3D Hough transform* for silhouette problems. Next we will see that this particular transform is well-suited to silhouette extraction and update.

Chapter 3

Hough space silhouette methods

As explained in Section 2.1.3, tracking disjoint silhouette loops as they appear on and vanish from the surface is a key challenge of object-space silhouette extraction. Previous methods using supporting-plane representations address this problem by computing silhouettes in a transform space where silhouette changes are spatially coherent, but suffer from a number of limitations mainly attributable to the plane transforms they employ.

We develop the *3D Hough transform*, introduced in Section 2.3.2, to address these limitations. This transformation is closely related to the classical dual transform used by Pop *et al.* in [69], but produces data sets with better properties for efficient storage and traversal. We detail the geometric attributes of this approach and its application to silhouette extraction later in this chapter. From this transform, we present algorithms for silhouette extraction and incremental update, showing significant improvement over previous methods.

3.1 Overview of data structure and algorithms

Our goal is to organize the elements of a triangle mesh into a data structure where silhouette changes are spatially coherent and easily identified. To this end, we transform each triangle into a point in 3D Hough space. The backbone of our search data structure is an octree built upon these Hough-space points, which we use to identify planes crossed by p_s by performing intersection tests against the *v-sphere* shown in Figure 2.2.

To facilitate fast silhouette extraction, each octree node, in addition to storing a set of Hough-space points spatially contained within the octree node, is augmented with two bounding volumes.

- **The point bounding volume (PBV):** The PBV is the tightest bounding volume of the set of Hough-space points belonging to the octree node. It can be utilized for more effective culling due to its tighter bounding of the set of Hough-space points in the octree node.

- **The edge bounding volume (EBV):** The EBV is the tightest bounding volume of the set of Hough-space points related to one or more Hough-space points in the octree node by an edge on the input mesh that is not on the silhouette from the origin; these *SFO edges* are treated separately as explained in Section 3.2.2. The EBVs facilitate efficient acceptance and rejection of silhouette edges.

Recall that the 3D Hough transform maps the view point to a sphere, referred to as the v-sphere, passing through the origin and the view point itself. The majority of silhouette edges, with respect to the view point, correspond exactly to the set of Hough-space point pairs, in which one point lies inside the v-sphere and the other lies outside. SFO edges do not fall into this category, as demonstrated below. For initial silhouette extraction, we apply a standard top-down, hierarchical octree-based culling scheme using the v-sphere. All octree nodes outside the v-sphere can be immediately culled away. Any octree node whose EBV is entirely contained in the v-sphere can also be culled. Recursion stops when the size of the octree node reached, measured by the number of Hough-space points it contains, is sufficiently small. At this point, mesh edges relevant to these points are tested to see whether they are silhouette edges. In addition, all edges that are on the silhouette with respect to the origin are also tested explicitly.

In many interactive applications, *e.g.* virtual walk-throughs and object tracking, the ability to quickly update the silhouette is highly desirable. Pop *et al.* [69] keep track of the difference between the silhouette sets at consecutive frames, in the dual space, and always start the search at the root of the octree. We show a substantial increase in performance with the same basic algorithm executed in Hough space. We improve performance further by taking advantage of frame-to-frame coherence and conduct an incremental neighbour search through the octree. A neighbour graph augments our octree and an appropriate selection of search nodes allows us to keep track of silhouette changes efficiently. Specifically, we only need to examine octree nodes that straddle or are contained in the *active region* defined by the v-spheres at two consecutive frames.

3.2 3D Hough transform for silhouette extraction

In this section, we provide some mathematical background on the 3D Hough transform and show how it can be applied effectively for silhouette computation. We also relate the Hough transform to the well-known dual-space transform and show the advantages offered by the former. We refer the reader to Section 2.3.2 for a formulation of the 3D Hough transform.

3.2.1 3D Hough transform and dual-space transform

Recall that the 3D Hough transform of a plane $\pi : ax+by+cz-d = 0$ is $\mathcal{H}(\pi) = (ad, bd, cd)$, while the dual transform of π is $\mathcal{D}(\pi) = (a/d, b/d, c/d)$. We can see that for any *fixed* π, $\mathcal{H}(\pi)$ and $\mathcal{D}(\pi)$ are related by inversion about a sphere. This implies that both the dual and the 3D Hough transform carry the same amount of information, and suggests that they are equally suitable for geometry processing. However, we will see that the 3D Hough transform has several advantages over the dual transform.

Advantages of the 3D Hough transform: It is easy to show that any bounding sphere S, centered at the origin, of a set of triangles also bounds the Hough transforms of the triangles. The tightest bounding sphere for the dual-space transforms however can be much larger than S. Inversion causes dual-space points to exhibit a highly nonuniform distribution, typically with severe clustering about the origin as well as points extremely far away from the origin; see Figure 3.1 for an example. Besides the precision issues, which influence the numerical stability of the algorithms, octrees constructed in dual space tend to have high and unevenly distributed leaf depths due to the nonuniformity of dual-space point distribution. These lead to poor performance in both initial and incremental silhouette extraction, compared to the use of Hough transforms. Note that Hough-space transforms of mesh models also often exhibit some level of clustering around the origin, as most meshes include faces whose supporting planes approach or even intersect the origin. Empirically, however, Hough-space silhouette extraction shows superior performance to its dual-space counterpart, with the same data structure construction and search algorithm, *e.g.* as given in Pop *et al.* [69].

3.2.2 Silhouette computation in Hough space

We can restate Definition 2.1.5 for meshes by describing the SGS of an edge e as the volume swept by the supporting plane of one adjacent face π_1 as it is rotated onto the other plane π_2 rather than as the symmetric difference of half-spaces. This is illustrated in Figure 3.2(a).

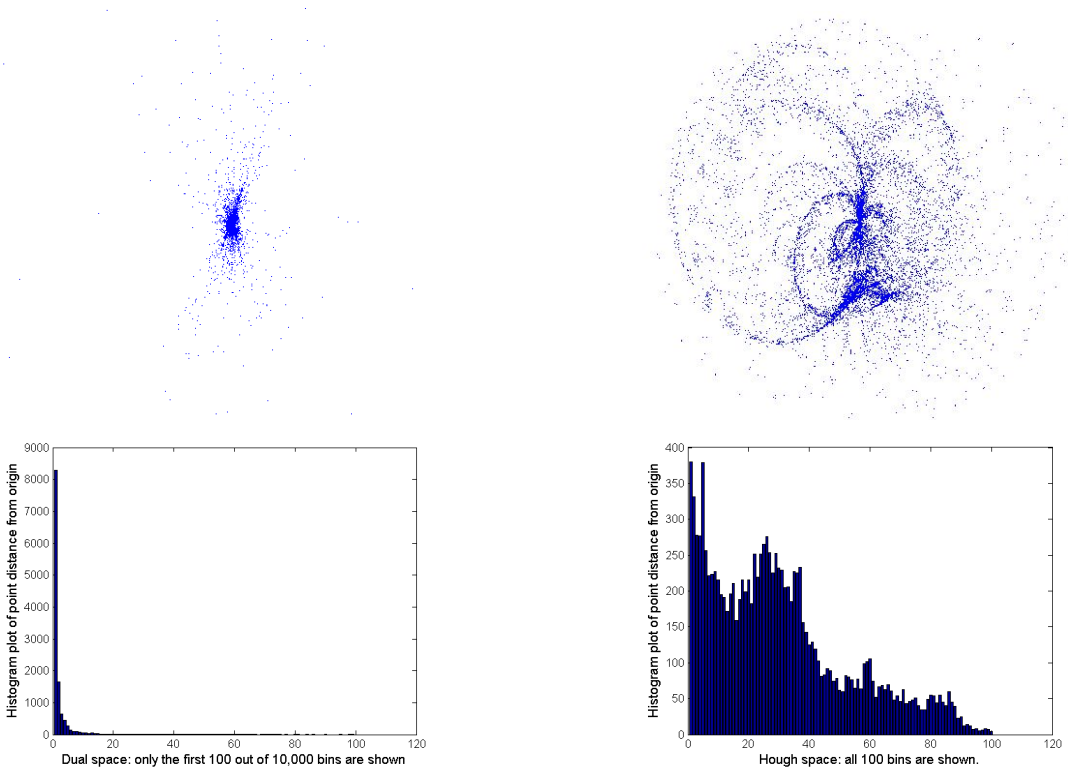


Figure 3.1: Scattered point plots (top) and histogram plots of distances from the origin (bottom) for the hand model shown in Figure 3.7. Left: dual space. Right: Hough space. The origin is chosen as the centroid of the model. Point plot in dual space shown is obtained after 10 levels of zooming in Matlab, while the Hough-space plot is shown as is. Some dual-space points are extremely far from the origin and not visible in the figure. Evidently, point distribution in Hough space is much more uniform (less clustering around the origin). This example is representative of the general trend.

In Hough space, the rotating plane traces out a *circular arc*, whose end points are the Hough transforms $H_1 = \mathcal{H}(\pi_1)$ and $H_2 = \mathcal{H}(\pi_2)$, as shown in Figure 3.2(b). The full circle, which we call the *Hough circle* for e, is defined by a diameter whose end points are O, the origin, and E, the intersection between the line extension of e and a plane passing through O and perpendicular to e. We define the Hough transform of the mesh edge e to be this circular arc traced out by the rotating plane. Clearly, the orientation of the triangles incident to e determine whether the arc contains the origin; this is precisely related to whether e is a silhouette edge when viewed from the origin O.

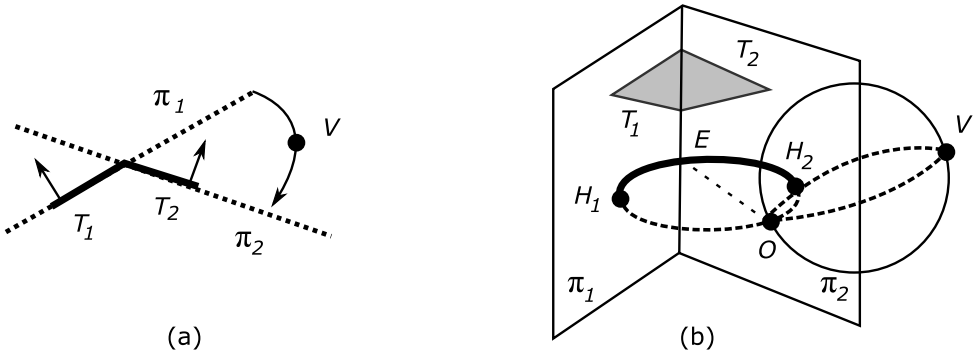


Figure 3.2: The Hough transform of an edge. (a) Plane rotation hits view point. (b) Plane rotation traces out a circular arc (thickened arc between H_1 and H_2), defined as the Hough transform of the edge e incident to triangles T_1 and T_2.

Theorem 3.2.1. *: The Hough transform of an edge e contains the origin O if and only if e is a silhouette edge, viewed from O.*

Theorem 3.2.2. *: Assume that no edges or their line extensions pass through the origin. Then an edge e is a silhouette edge with respect to a silhouette point p_s if and only if the Hough transform of e is tangent to the v-sphere defined by p_s or it intersects the v-sphere through a point other than the origin.*

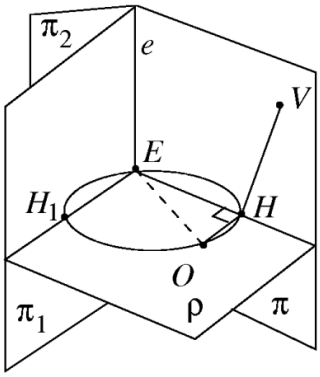


Figure 3.3: Figure for proof of theorems 3.2.1 and 3.2.2.

Proof: Refer to Figure 3.3. First assume that e is a silhouette edge with respect to $p_s = V$, with Hough transform $H(e) = E$. Then as the supporting plane π_2 rotates about e towards π_1 it will intersect V. Let π be the plane through e containing V and $H = \mathcal{H}(\pi)$, which lies along $\mathcal{H}(e)$. O is the origin. Then the Hough circle of e passes through O and H. It follows that $OH \perp EH$. As the supporting plane ρ of the Hough circle is perpendicular to π, we have $OH \perp \pi$. Thus $OH \perp HV$ and H must lie on the v-sphere determined by V. Conversely, if the v-sphere intersects $\mathcal{H}(e)$ at H, we have $OH \perp EH$ and $OH \perp HV$. It follows that $OH \perp \pi$ and thus $\rho \perp \pi$. Hence the plane π passes through e. Since H is on $\mathcal{H}(e)$, π is an intermediate rotating plane that hits V, implying that e is on the silhouette with respect to V. Finally, we can see that the tangency case in the theorem occurs when π passes through O (dashed line).

Theorems 3.2.1 and 3.2.2 are the Hough-space equivalents of Theorems 1 and 2 from Pop *et al.* [69]. Figure 3.4 depicts the different cases, in an orthographically projected view with projectors parallel to the edge in question, as an illustration for Theorem 3.2.1. The assumption in Theorem 3.2.2 can be ensured through perturbation in preprocessing.

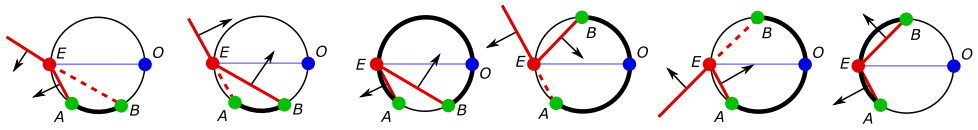


Figure 3.4: The Hough transform (thickened arc) of an edge, projected to E, contains the origin O if and only if it is a silhouette edge, viewed from O; arrows depict plane normals.

Corollary 1: If e is not a silhouette edge, viewed from the origin, then it is a silhouette edge with respect to a view point V if and only if H_1 and H_2, the Hough transform of faces incident to e, lie on opposite sides of the v-sphere associated with V. If e is a silhouette edge, viewed from the origin, then it is a silhouette edge with respect to V if and only if H_1 and H_2 lie on the same side of the v-sphere.

This corollary allows us to speed up initial silhouette extraction, since the majority of the edges are typically not on the silhouette [49] when viewed from the origin. To extract the silhouette from this set of edges, with respect to a v-sphere, one only needs to examine octree nodes inside or intersecting the v-sphere – details are given in Section 4.4.

The situation for incremental silhouette updates is simpler, as we are interested only in the *change* in the silhouette set. From frame t to frame $t+1$, the membership of an edge in the silhouette set changes if and only if the front- or back-facing status of one of its incident faces changes with respect to p_s. Thus it is sufficient to examine octree nodes inside or intersecting the active region between the v-sphere at the two frames.

3.3 Augmented octree in Hough space

We use a bounded augmented octree, built on top of the set of Hough-space points corresponding to a set of triangles in a scene, for silhouette extraction. The root of the octree represents the tightest, axis-aligned bounding box of the whole point set. Each node in the octree stores the following:

- Eight pointers to its children.
- Six pointers to its neighbour nodes, explained in detail in Section 3.3.1.
- Three axis-aligned bounding boxes. In addition to the octant bounding box, or *OBV*, which bounds the full octant associated with the node, we also store
  - *PBV:* the tightest bounding box of all the Hough-space points enclosed by the node; for now we us denote this set by W. Note that PBV is enclosed by OBV.
  - *EBV:* the tightest bounding box of the set of Hough-space points *related* to a point in W. Specifically, a Hough-space point H is related to $H' \in W$ if $H = H'$ or the triangles corresponding to H and H' share an edge and this edge is *not* on the silhouette when viewed from the origin. The EBV is utilized for static silhouette extraction, as explained in Sections 3.1 and 4.4.
- **Extra data** to indicate whether the node makes a good candidate for the *active set*. The active-set candidates determine where in the octree we perform neighbour traversal; this is described in details in Section 3.4.3.

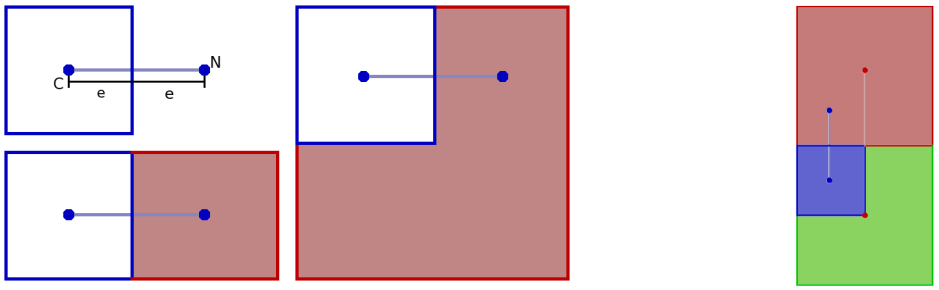
In our current implementation, octree nodes are recursively subdivided until each non-empty leaf node has precisely one Hough-space point. In general, one can stop the subdivision when the number of Hough-space points in a node falls below a user-defined threshold.

3.3.1 The neighbour graph

During incremental silhouette update, we walk from properly selected octree nodes to their neighbours rather than always searching from the tree's root, as in [69]. We build a *directed neighbour graph* whose vertices are the octree nodes and whose edges connect a node to its neighbours. A node may have up to six neighbours, one across each face of its OBV. Nodes on the boundary of the octree do not have neighbours along those boundary faces.

Since our octrees are not fully populated in general, it is not always immediately obvious which nodes are neighbours of a given octree node. To determine a node's neighbour in a given direction $\vec{u}$ (there are six such directions as given above), where $\vec{u}$ is of unit length, we first find a *neighbour point*. Consider an octree node whose OBV is centered at point C with extent or half-width e along the direction of $\vec{u}$. The node's neighbour point N with respect to $\vec{u}$ is $C + 2e\vec{u}$, as shown in the upper-left diagram of Figure 3.5(a).

We define the *neighbour node* of an octree node R, in a given direction, as the deepest node in the tree, no deeper than R, that contains R's neighbour point in that direction. Typically, R's neighbour would be at the same depth as R, as shown in the lower-left diagram of Figure 3.5(a). But this is not guaranteed if the octree is not full. For example, a node's neighbour may be its parent, as shown on the right side of Figure 3.5(a). It is also worth noting that the neighbour relation is not symmetric in general, as shown in Figure 3.5(b). Symmetry is ensured only between two neighbouring nodes that are at the same depth.



(a) (b)

Figure 3.5: 2D illustrations of the neighbour relations. (a) Upper-left: an octree node centered at C with its east neighbour point N. Lower-left: two neighbouring nodes at the same depth. Right: the east neighbour of a node is its parent. (b) Asymmetry of the neighbour relation: the north neighbour of the blue node is the red node, whose south neighbour is the green node (parent of the blue node).

3.3.2 Edge list

In addition to the augmented octree, we maintain an edge list to store relevant information for each edge: the position of each vertex on the edge, and facing (front or back with respect to the current view point) flags for each face adjacent to the edge. We can therefore easily determine whether an edge is on the silhouette or not by checking the facing flags of its incident faces, as done for edge buffers [12].

3.4 Silhouette extraction algorithms

In Section 4.4, we describe our initial silhouette extraction scheme. Subsequently, we offer two options for identifying changes to the silhouette on a frame-by-frame basis. One involves a full traversal of the Hough-space octree from the root at every frame; the other maintains

a list of active octree nodes near the silhouette and proceeds incrementally. Both algorithms work on the same principle. We are only concerned with faces that have passed from front-facing to back-facing, or vice versa. A face changes its facing status when it *crosses* the v-sphere (defined in Section 3.1). Thus any Hough-space point that has changed its facing status must lie within the symmetric difference between the v-spheres at the current and the previous frames; we refer to this symmetric set difference as the *active region*.

We maintain two v-spheres, one for the current frame and one for the previous frame, during view point changes. At each frame, we consider octree nodes whose associated PBVs intersect the active region. Once we have identified all the faces whose facing status relative to the view point has changed, we change the edge flags associated with these faces, retest those edges affected for silhouette status, and add or remove them from the silhouette set.

3.4.1 Initial silhouette extraction

In preprocessing, we determine the set $\mathcal{S}^+(O)$ of edges that are on the silhouette with respect to the origin O. Denote the remaining set of edges by $\mathcal{S}^-(O)$. Given a silhouette point p_s, all edges in $\mathcal{S}^+(O)$ will simply be tested explicitly against p_s to determine their silhouette status with respect to p_s. Note that $|\mathcal{S}^+(O)|$ is expected to be small and comparable to the size of the silhouette set with respect to the silhouette point p_s. Thus the cost of these explicit tests is expected to be roughly the same as the size of the extracted silhouette.

To extract silhouette edges with respect to p_s from the set $\mathcal{S}^-(O)$, we traverse the octree from its root. We can ignore subtrees whose EBVs are entirely contained within the v-sphere and, more aggressively, whose *PBVs* are entirely outside the v-sphere.

To initiate incremental silhouette search, to be described in Section 3.4.3, we add appropriate octree nodes that intersect the v-sphere at the *first time frame* to its *active set*. Details on active sets are given in Sections 3.4.3 and 3.4.4.

3.4.2 Full-traversal silhouette update

For full traversal, we start at the root of the Hough-space octree at each frame, analogous to Pop *et al.* [69], and descend into nodes whose PBVs intersect the active region. Recursion stops when the number of Hough-space points in a node is sufficiently small; these points are tested for facing to decide the silhouette status of their adjacent edges. If a node's PBV is contained by the active region, all the Hough-space points therein also undergo the facing

test. At leaf nodes, we check the facing of every point in the leaf node if the number of points is sufficiently small. Otherwise, we consider the node's PBV and check its points only if the PBV intersects the active region.

3.4.3 Incremental silhouette update

Between consecutive time frames t and $t+1$, our incremental silhouette search aims at quickly identifying all Hough-space points lying within the active region; mesh faces corresponding to these Hough-space points are tested to update the silhouette. Rather than searching from the root, as for full traversal, we walk along nodes deep in the octree to avoid computations on interior nodes close to the root.

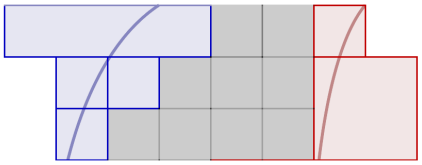


Figure 3.6: A partial example of incremental search. The red (respectively, blue) arc is part of the boundary of the v-sphere at frame t (resp., frame $t+1$), and the red (resp., blue) nodes are from the active set for that frame. Breadth-first search proceeds from the red nodes, through the gray nodes (and their parents), and ends at the blue nodes.

Our search starts with a set of octree nodes in the *active set* for frame t. Each node in this active set must intersect the v-sphere for frame t and be an *active-set candidate*. Note that not all octree nodes are deemed to be active-set candidates. A judicious choice of the set of active-set candidates plays an important role in speeding up incremental silhouette updates; this is described in details in the next section.

For silhouette update at frame $t+1$, we perform a breadth-first search through the neighbour graph (see Figure 3.6), starting with nodes in the active set for frame t. During the search, we recurse into octree nodes partially contained in the active region, performing intersection tests between bounding volumes and the active region. For a node completely contained in the active region, all the Hough-space points therein undergo the simpler facing test against the view point.

3.4.4 Selection of active-set candidates (ASCs)

If we allow incremental search to proceed along leaf nodes of the octree, we may end up processing a large number of nodes during traversal of the active region. Ideally, we wish to

cross the active region in few steps to avoid the overhead of many bounding box checks and queue updates. To this end, nodes at higher levels in the tree are preferred. On the other hand, if an octree node is completely contained in the active region, we need not check any of its children's bounding boxes — we can simply report all of the faces contained in its subtrees as changed. We are therefore interested in identifying nodes which are above leaves in the tree, but are still sufficiently low-level to avoid searching a large number of interior nodes. We call these nodes *active-set candidates* or *ASC*s, as we will add only these nodes to the active set at any time. We must be careful to ensure that every leaf node has an ASC above it in the octree.

A simple and reasonably effective way to choose ASCs would be to select the so-called *twig nodes* — nodes that contain a leaf child. However, twig nodes may vary in depth considerably across an unbalanced octree and may still be too small for a large mesh with a deep Hough-space octree. We thus resort to a different heuristic to produce a more suitable set of ASCs. Intuitively, we select nodes whose subtrees are well balanced, so as to take maximal advantage of hierarchical octree culling when the node intersects the active region, and whose neighbours are at the same or similar depths. We first define an ASC cost for each node in the octree,

```
ASC-cost(node) {
   subtree_cost = node->depth / node->num_kids

   nbr_cost = 0.0
   for (n in node->neighbours)
      nbr_cost += abs(node->depth - n->depth)
   nbr_cost /= node->num_nbrs

   return w1 * subtree_cost + w2 * nbr_cost
}
```

where $w1$ and $w2$ are set by experimentation to be 2.0 and 0.25, respectively.

We identify the set of ASCs in an octree in preprocessing by first computing the ASC cost of each node. To ensure that the candidate set stays near the leaves, we will only make a node an ASC if its children are either leaf nodes or ASCs themselves. If the ASC cost of a node is less than the average ASC cost of its ASC children, we remove the child nodes from the set of ASCs and insert the current node. We iterate this process until no changes to the set of ASCs are made.

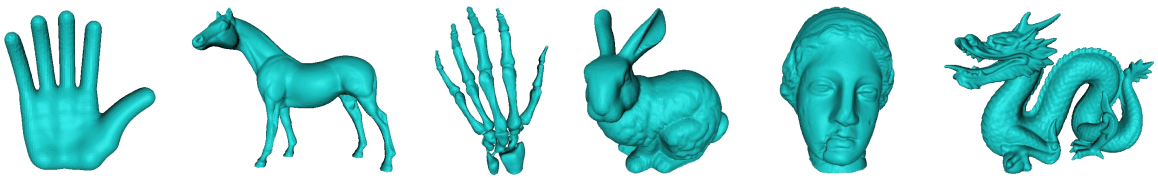


Figure 3.7: Test models (from left to right and top-down): Hand (face count: 12K); Horse (40K); Bone (65K); Bunny (70K); Igea (268K); Dragon (300K).

3.5 Experimental results

We have tested our methods, as well as a simple dual-space silhouette extraction algorithm similar to that of Pop *et al.*[69], on a PC running Linux 2.6.8 with an Intel Xeon 2.80GHz processor, 2GB of memory, and an NVidia GeForce 6800 GT. We used six test models, shown in Figure 3.7, with face counts ranging from 12K to 300K. The primary performance measure is the number of bounding box checks (against v-spheres for intersection tests) executed, as this is the most significant elementary operation used in the silhouette algorithms.

3.5.1 Static silhouette extraction

We have tested our static (initial) silhouette extraction algorithm on the test models with varying numbers of polygons, choosing 10 random silhouette points for each resolution and averaging the results. We see in Figure 3.8 that the average number of bounding box tests increases sublinearly to the model size, the same behaviour we expect from the silhouette. While several other papers, *e.g.* [39, 75], have presented static silhouette extraction methods which experimentally scale linearly with the size of the silhouette, none of them supports incremental silhouette extraction on the same data structure.

3.5.2 Incremental silhouette extraction

We have tested our incremental silhouette extraction algorithms by moving a silhouette point along a fixed circular path on the xz plane around the test models. Figure 3.9 shows the number of bounding box checks for three methods: full-traversal (always traversing from the root, as explained in Section 3.4.2) and incremental Hough-space methods and a simple full-traversal dual-space algorithm [69].

Silhouette extraction in Hough space significantly outperforms its counterpart in dual space. Incremental search through nodes near the leaves provides a noticeable and consistent

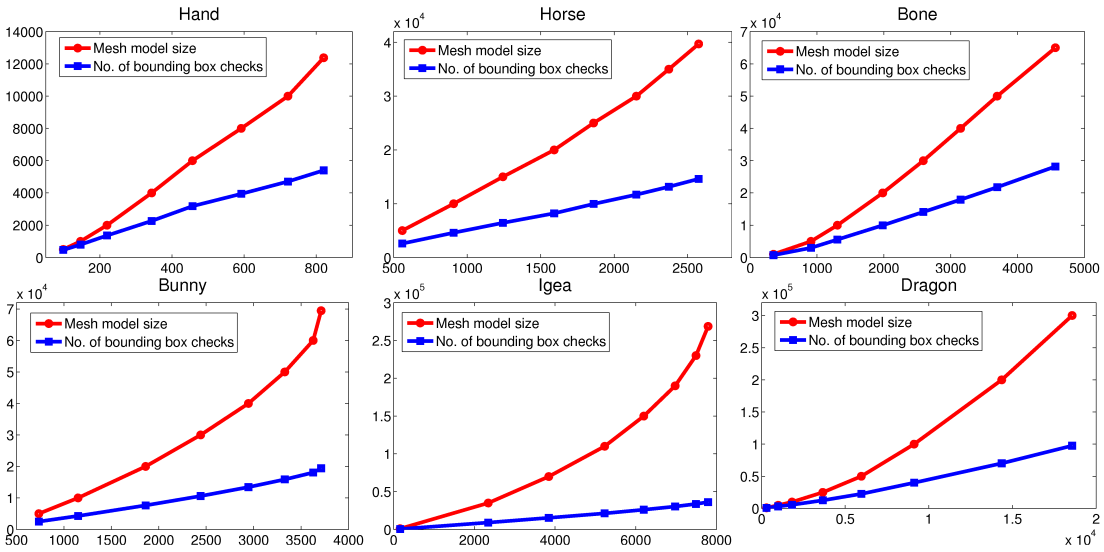


Figure 3.8: This plot experimentally shows output sensitivity of our static silhouette extraction. Horizontal axis gives average silhouette size (computed for 10 random silhouette points). Plotted in red is the model size and in blue the number of bounding-box checks (exhibiting a roughly linear behavior).

improvement over the full-traversal approach, though not as great a gain as that achieved by moving from dual space to Hough space. Dual-space search has to cull away far more interior nodes than either Hough-space algorithm, and in Hough space, full-traversal search must cull more nodes than incremental update. It is also worth noting that Hough-space algorithms exhibit much more stable behavior from frame to frame, compared to the dual-space algorithm, as the octrees built around Hough-space points are more evenly balanced than those in dual space.

The three algorithms are more evenly matched in terms of the number of face checks (sidedness test against v_s) performed, as shown in Figure 3.10 for two of the test models. Further, bounding-box checks are far more expensive than face checks, with roughly twice the number of floating-point operations and five times the number of conditional branches. Therefore, while the number of face checks performed by a silhouette extraction algorithm is important, the improved performance of our algorithms is best seen by examining bounding-box tests, as in Figure 3.9.

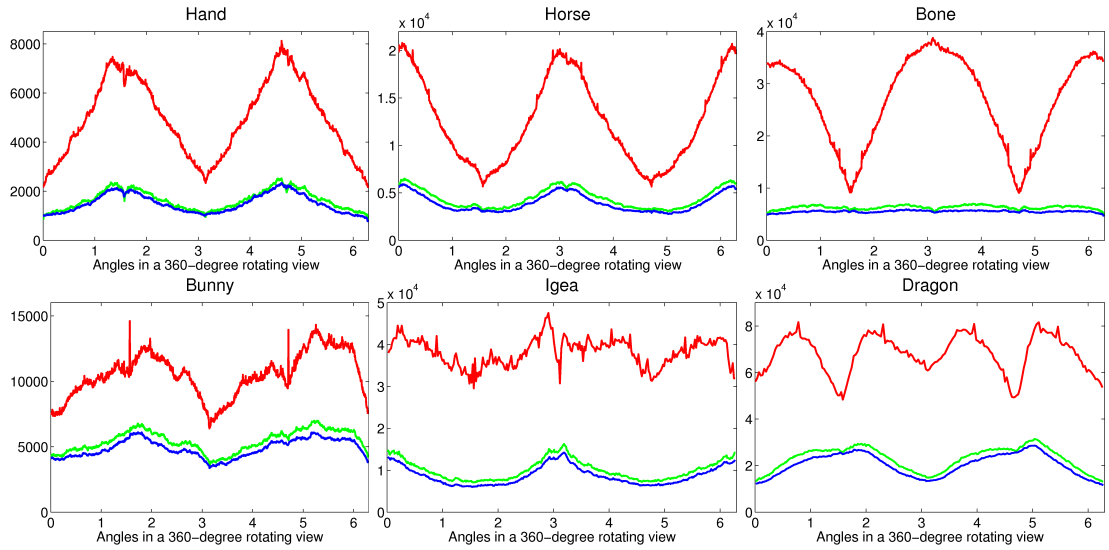


Figure 3.9: Number of bounding box checks under incremental silhouette extraction for our test models. Results for the full-traversal Hough-space algorithm are given in green; results for incremental tree search in Hough space are given in blue; results for full-traversal in dual space are given in red. The vertical axes denote the number of checks; the horizontal axes denote the position of the silhouette point, given by an angle in the xz plane measured from the $+x$ direction.

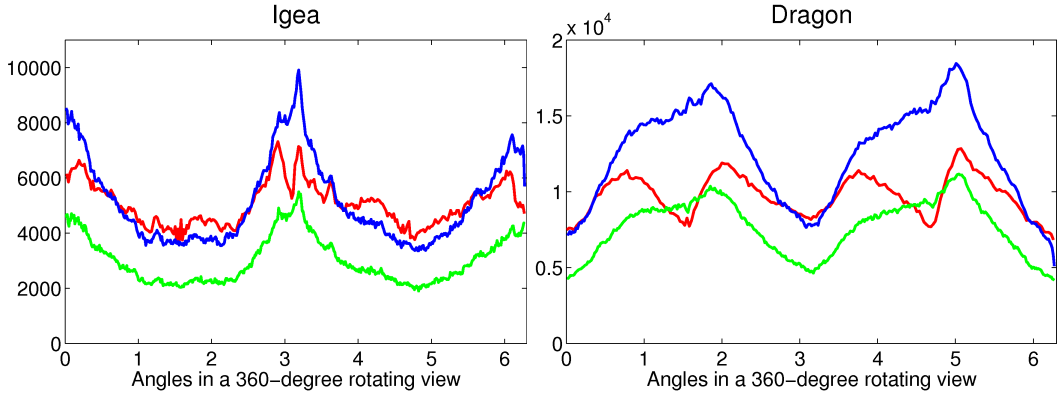


Figure 3.10: Number of face checks under incremental silhouette extraction for two test models. Colours and axes used are the same as for Figure 3.9.

3.5.3 Histogram of leaf node depths

The performance of Hough-space algorithms relative to their dual-space counterparts may be explained by certain characteristics of the octrees generated for the respective point sets.

Figure 3.11 shows histograms of leaf node depths for Hough-space and dual-space octrees. Specifically, we plot the number of leaf nodes at various tree depth levels. Hough-space octrees for all test models are consistently and considerably shallower, which leads to fewer interior nodes, fewer bounding box checks, and more efficient subtree culling.

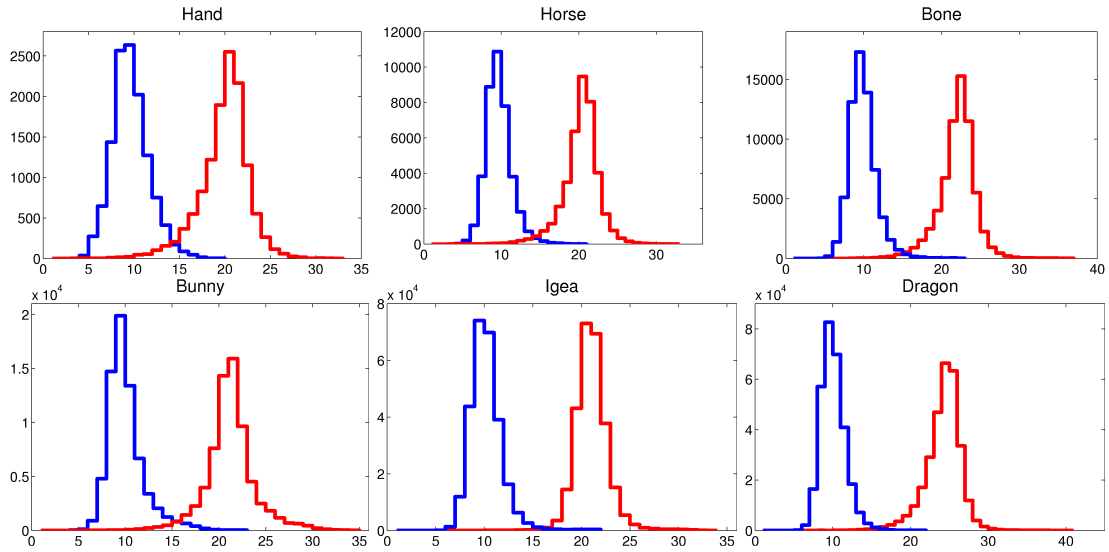


Figure 3.11: Histogram plots for the depths of leaves in an octree. Red: dual space. Blue: Hough-space.

3.6 Conclusions

We have developed a complete framework for both initial extraction and incremental update of geometric silhouettes, taking advantage of a thorough understanding of the supporting planes at each face and their 3D Hough transforms. By focusing on the behaviour of the 3D Hough transforms for a given edge and within a given neighbourhood in transform space, we are able to develop a unified method for the two problems and identify options for performance improvements.

However, two areas for improvement remain unaddressed in this chapter. First, while we have identified the importance of the origin for the behaviour of the 3D Hough transform, we have chosen only the centroid of the model as our origin. Second, while we have noted that the global characteristics of the 3D Hough transform's point distribution make it superior to the dual transform, we treat each transformed point independently. In the next chapter we take a global view of tangential planes, and in so doing are able to obtain a significant performance improvement in silhouette extraction.

Chapter 4

Tangential distance fields

In Chapter 2.3.2 we describe the 3D Hough transform in terms of a plane π and an origin, and in Chapter 3.2.1 we see that the 3D Hough transform outperforms the geometric dual transform specifically due to its behaviour around the origin. It naturally follows that we would like to choose an origin – or, put differently, translate our input – to maximize the advantages brought on by the 3D Hough transform's more uniform point distribution. Since the Hough transform of a plane varies with the distance from that plane to the origin, we must examine all candidate origins – all points in space – based on their distances to all supporting planes in the input. We must also develop a metric to evaluate candidate origins based on their aggregate distance from these planes.

The result is a structure called the *tangential distance field*, or TDF, which encapsulates as a scalar field some measure of the supporting planes on the input mesh. In this chapter, we describe the TDF and the functions used to generate it, and show that by careful construction of these functions we can apply the TDF framework to a number of geometric problems related to silhouette structure, such as viewpoint selection, camera path planning, and light placement.

4.1 Tangential distance fields

Consider a smooth surface S embedded in $\mathbb{R}^3$. At any point p on S, we compute a tangent plane $\pi(p)$ of S. The set of tangent planes $\mathcal{T}(S) = \{\pi(p) : p \in S\}$ is called the tangent-space representation of S; in fact, $\mathcal{T}(S)$ forms a surface in 4-D space. In the reminder of this chapter, we shall only consider tangent-space representations of triangle meshes and their

derived constructs. Naturally, the tangent planes are now replaced by supporting planes of the mesh faces (triangles). The mapping from mesh faces to supporting planes is not injective in general: many faces can share the same tangent plane. This is in fact one of the strengths of the tangent-space representation for geometry processing. For example, it allows us to identify points on the same silhouette, even when they are arbitrarily distant on the surface.

This reinforces the connection between mesh silhouettes and supporting planes. Recall that the silhouette status of a mesh edge changes precisely when the viewpoint crosses one of the supporting planes of its adjacent faces. It is therefore not surprising that constructs derived from the supporting planes, *e.g.* the 3D Hough transform, are often used for silhouette analysis. Here we step away from formulations of the silhouette based on a given p_s and examine supporting-plane representations in more general terms.

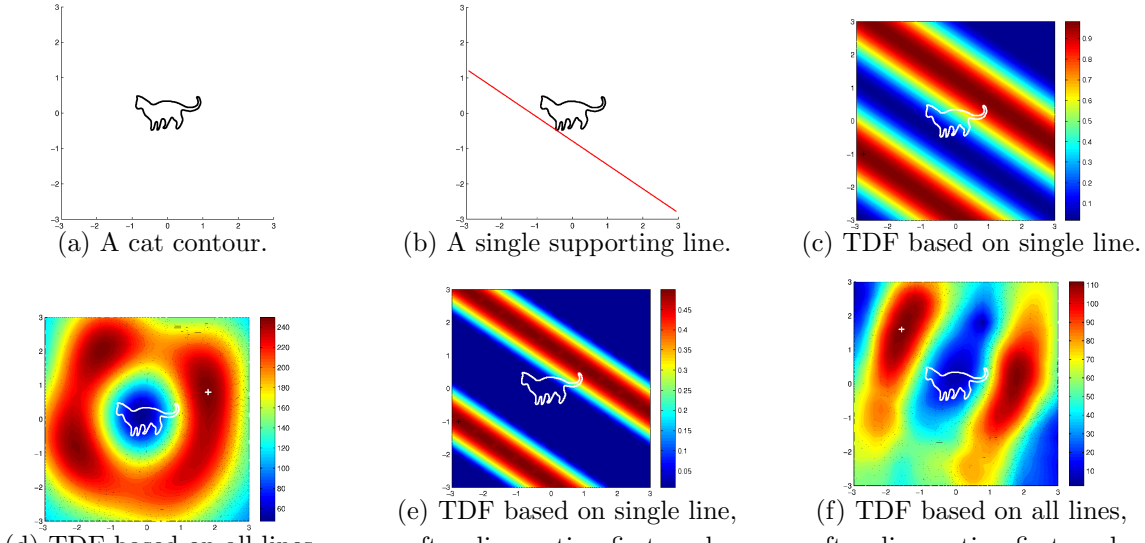
We define a scalar field in which every point in space is given a value as a weighted distance to the supporting planes. We call this a *tangential distance field* or TDF. Specifically, let M be a triangle mesh whose set of supporting planes is given by $\mathcal{T}(M)$, then for $p \in \mathbb{R}^3$, the TDF value at p is

$$\mathcal{D}(p, \mathcal{T}(M)) = \sum_{i=1}^{|\mathcal{T}(M)|} f_{\mathrm{TD}}\big(\mathrm{dist}(\pi_i, p)\big), \tag{4.1}$$

where $\pi_i \in \mathcal{T}(M)$, f_{TD} is a weight function which we refer to as the *tangential distance function*, and $\mathrm{dist}(\pi_i, p)$ is the *signed* point-to-plane distance between p and π_i. The function f_{TD} is critical to the meaning of the TDF. It specifies the voting scheme and should be chosen on an application-specific basis. Once we have defined the TDF, we can use it to select points of interest around a mesh.

4.2 Point selection scheme

In this section we describe the use of the TDF for selecting single and multiple points of interest. Depending upon the application, these points could be camera positions for generating thumbnails of an object, origins for an optimized partition of a mesh's Hough transform, or points of other types. This should be encoded in the support distance function and the search domain, which we cover in Section 4.3.



(a) A cat contour. (b) A single supporting line. (c) TDF based on single line.

(d) TDF based on all lines. (e) TDF based on single line, after discounting first peak. (f) TDF based on all lines, after discounting first peak.

Figure 4.1: A 2D example of TDF construction and point selection. (a): A cat contour composed of line segments, analogous to a mesh composed of triangles. (b): One supporting line (in red), analogous to a mesh supporting plane. (c): Plot of TDF based on the supporting line shown in (b); the f_{TD} for Hough-space silhouettes (see Figure 4.8) is used. (d): TDF plot based on all supporting lines. The highest peak is indicated by the white cross. (e): TDF based on the same supporting line in (b), discounted by the score it gave to the first peak. (f): TDF based on all supporting lines, after discounting contributions made to the first peak.

4.2.1 Selecting a single or first point

Selecting a single, best point amounts to finding the peak value in the TDF. This is illustrated using a simple 2D example shown in Figure 4.1; see (a)-(d). We have chosen a particularly simple peak selection scheme by coarsely sampling the field's domain, and then recursively subsampling and searching around the highest-valued sample until a quality criterion is met, *e.g.* until the difference between consecutively detected peaks falls below a user-specified threshold. More sophisticated schemes can certainly be applied as well.

To ensure good results via sampling, the sampled scalar field should be sufficiently smooth and slow-varying or pre-filtered to be so. Smoothness of f_{TD} is ensured by our choice in the applications; Section 4.3 provides the details. As the TDF is a sum of instances of f_{TD}, it is smooth itself. In practice, we have found that the functions we use are generally sufficiently well-behaved so that a relatively coarse initial sampling works sufficiently. We

use a 17^3 initial sampling grid throughout. Since the smoothness of the TDF depends upon the smoothness of the summed functions, rather than characteristics of the input mesh, sampling resolution can be kept constant for a given application. However, more abrupt f_{TD} changes than those encountered in this paper may generate steeper gradients in the TDF and will require a denser sampling pattern. An additional smoothing step may also be applied prior to sampling and peak detection.

In our experiments, we have not observed any meshes which produced multiple equal-valued maxima; however, it is possible that meshes with rotational or reflective symmetry may do so. In such a case, any of these maxima can be chosen as the first candidate. Any other maxima which are optimal for planes not contributing to the chosen point will be chosen later according to the weighting scheme described in Section 4.2.2. If these symmetric maxima are unsuitable for the given problem, an asymmetric f_{TD} can be used to slightly penalize points across symmetry axes or planes; Section 4.5 provides an example.

4.2.2 Selecting the next point

Having selected the peak value from the initial TDF, we may wish to find the next-highest peak, and the next, and so on. Each selected point will benefit some planes more than others. Depending on the application, this might mean placing some faces far from the silhouette, or maximizing the perceptual contribution of those faces. When selecting subsequent points, we would like to give priority to planes that have not yet contributed to a selected point, but we should not ignore planes that have already done so.

To provide a better intuition, let us use the voting analogy to describe our approach. Each plane assigns a vote to each point in space, weighted by f_{TD}. The sum of these weighted votes over all planes is the initial TDF. To select subsequent points, we bias the votes cast by each plane when selecting point k by the preference that plane has for the first $k-1$ points. Exactly how this is to be accomplished depends upon the application, but in general we wish to find the plane's highest weighted vote among the already-selected points and compare that to the plane's vote for the point currently under consideration:

$$w_{\mathrm{prev}}(\pi) = \max_{i \in 1..k-1} f_{\mathrm{TD}}(\mathrm{dist}(L[i], \pi)),$$

where π is the plane casting its vote and L is the list of previously-selected points of interest.

Now, when finding the weight for π's vote on a point P, we calculate $w = f_{\mathrm{TD}}(\mathrm{dist}(P, \pi))$ as usual, but cast a vote with weight

$$w_k(\pi) = \begin{cases} 0 & \text{if } w < w_{\mathrm{prev}}(\pi) \\ w - w_{\mathrm{prev}}(\pi) & \text{otherwise.} \end{cases}$$

Thus planes which have not yet elected a point which they favour cast heavily weighted votes, while planes which have already elected a suitable point of interest can still contribute to an even better point while not overwhelming the others. Figures 4.1 (e) and (f) give an example of this process.

4.2.3 Complexity considerations

Whenever we sample the TDF, we must compute the influence of each of the n triangles in the mesh, an $O(n)$ operation. Any performance improvements over the brute-force solution must therefore come from efficiency in the sampling. These improvements are a result of tradeoffs between the precision with which an application-dependent f_{TD} represents the features we need and the sampling density required to find peaks in the TDFs generated by the functions.

A more detailed analysis of the relationship between the properties of each f_{TD} and the number of samples required to find peaks in the TDF to a given accuracy may provide some insight into the structure of the method. However, we have obtained consistently good results by subsampling around peaks using a 17^3 lattice of increasing resolution, needing no more than three subsampling iterations.

Model	Size (tris)	Origins	Time
Hand	12379	2	1m30.88s
Horse	39699	3	8m52.12s
Bone	65001	2	7m26.82s
Bunny	69452	3	16m49.0s
Igea	268686	4	76m32.32s
Dragon	200000	3	43m12.57s

Table 4.1: Timing results for Hough space origin optimization (see Section 4.4). These times show that our method is presently unsuitable for real-time processing, but quite practical as a preprocessing step.

Due to the linear complexity of evaluating the TDF and the not insignificant number of samples required to find a peak, our method is not at the moment suitable for real-time application. It is, however, quite practical as a preprocessing tool, and all of the applications given in this paper fit this role. As an illustration, we provide timing results for the Hough space optimization problem (our first application, described in Section 4.4) in Table 4.1.

4.3 A voting scheme cookbook

In every case we assume that the input mesh has been normalized to fit within the unit sphere. For the given applications, we assume that the mesh is largely smooth; however, since the voting scheme acts as a filter over the whole mesh, a small number of local irregularities, boundaries, and even non-manifold edges are easily tolerated.

4.3.1 Domain restrictions

In many cases, the application itself will restrict the domain of the TDF. In the viewpoint selection problem, for example, it makes sense to place cameras within a spherical shell centered around the mesh, maximizing the size of the model in the rendered image while not allowing the view frustum to clip the mesh. However, we must understand how different restricted domains affect the interpretation of f_{TD}s.

Unbounded domain

We first consider an unbounded domain to illustrate the importance of bounding the domain of f_{TD}. Given a domain that extends to infinity, we cannot infer anything new from the distance of a point to a plane. In such a domain, therefore, the f_{TD} will only tell us whether a point is close to, or far from, the plane in question. However, when addressing problems involving the dual or 3D Hough transforms, this may be sufficient: these transforms are defined in terms of distances from the chosen origin to a set of planes.

Bounded domain

Most domains will be bounded, if for no other reason than that it is inconvenient to sample an infinite structure at a fixed resolution. When the domain is bounded on the outside, the maximum widths of the silhouette wedges of each edge on the mesh are likewise bounded. These wedges are the regions defined by the supporting planes of the wedge's adjacent triangles from which the edge is on the silhouette; see Figure 4.6 and previous chapters for more details.

If we know (or assume) the mesh to be smooth, we can infer silhouette information from point-plane distance: any p_s a certain distance away from a given plane is unlikely to be within the SGS of any edges on that plane. The converse is not necessarily true: near an

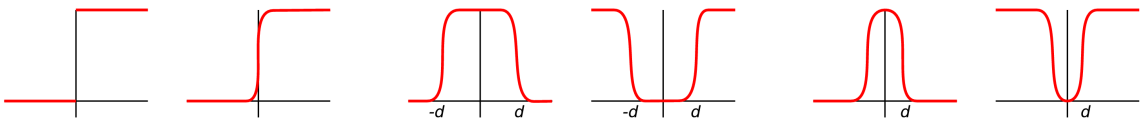


Figure 4.2: Tangential distance functions to identify front-facing planes (left), point-plane distance (centre), and silhouettes (right). The far-left function is discontinuous, and cannot be used as an f_{TD}. The function to its right is an acceptable substitute.

edge, a p_s may be arbitrarily close to a supporting plane on that edge but still outside its SGS. Summing over every plane in the mesh, we can identify p_ss with small silhouettes; if we have reason to prefer p_ss far from mesh edges, we can identify p_ss with large silhouettes as well.

Spherical shell domain

If the domain of the TDF can be restricted to a spherical shell around the mesh, we can infer further information from the f_{TD}. Since we can guarantee a minimum distance from each edge on the mesh, we can reliably identify p_ss which see large silhouettes, or which place certain edges on the silhouette.

We can also infer the viewing angle from a point to the triangle that generated a given supporting plane. Since we are restricted to a spherical shell some distance from the mesh itself, we can interpret the point-plane distance as roughly representative of the sine of the viewing angle. We should take care not to expect great precision from this interpretation – however, since our f_{TD}s must be smooth, we can identify viewpoints with "small" and "large" average viewing angles with some confidence. Note that this can be seen as a generalization of silhouette identification.

4.3.2 Tangential distance functions (f_{TD}s)

To maintain the desirable sampling properties of the TDF, we must avoid abrupt changes in the f_{TD}. Any function that steps smoothly from 0 to 1 with zero derivatives at each extreme will be suitable. In our work, we have chosen the cubic function $f(x) = 3x^2 - 2x^3, x \in [0, 1]$ for efficiency, which when scaled produces a result similar to Figure 4.2 (second from left). Note that we can invert the meaning of f by taking $g = 1 - f$.

Triangle facing

The input argument to the f_{TD} is the *signed* point-plane distance. This allows us to account for plane orientation (and therefore triangle facing) in our functions. We must maintain smoothness. We cannot, for example, select for all front-facing triangles with a step function. We must instead insert a smooth step at the origin, even though this gives a small weight to back-facing triangles (see Figure 4.2).

Point-plane distance

It is straightforward to build a f_{TD} that votes for near or far points; see Figure 4.2 (centre). As with the plane-orientation function shown in Figure 4.2, we cannot select exactly those points closer or further than a set distance d; we accept a certain imprecision in our f_{TD}s to maintain its desirable sampling properties.

Silhouette identification

As discussed above, with a restricted domain it is possible to identify viewpoints which see large (or small) silhouettes by their aggregate distance from the mesh's supporting planes. We can generally assume a smooth mesh and restrict the TDF domain to the near vicinity of the mesh, thus identifying viewpoints with large (or small) silhouettes with respect to a given plane usually amounts to voting for (or against) points that are very close to the plane; see Figure 4.2.

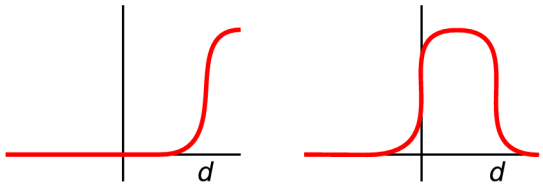


Figure 4.3: f_{TD}s combining several of the basic properties described earlier. The left-hand function votes for points far in front of each plane; the right-hand function votes for points near, but still in front of, each plane.

Viewing angles

By further restricting the domain, we can treat distance as roughly indicative of the sine of the viewing angle, as mentioned in Section 4.3.1. The relationships between point-plane distances and approximate viewing angles (with appropriate f_{TD}s) are shown in Figure 4.4.

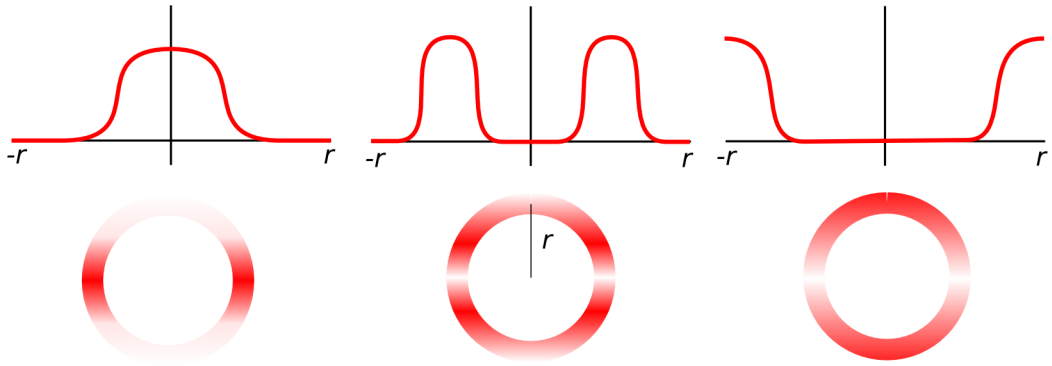


Figure 4.4: f_{TD}s for viewing angles. From left to right: f_{TD}s selecting small (near zero), moderate (near $\pi/6$), and large (near $\pi/2$) viewing angles. The bottom row shows the effect of the function above on a slice through a restricted-spherical domain.

Combining f_{TD}s

Most applications will require more detailed f_{TD}s than the single-feature functions given above. Hough space origin optimization, for example, combines the off-silhouette function in Figure 4.2 with an inverted far-from-plane function from Figure 4.2; see Section 4.4.

When combining f_{TD}s, we select the *features* we need from each of the source functions. For example, to build a function that votes for points far from, and in front of, each plane, we can take the far-from-plane function from Figure 4.2 and remove the negative (left-hand) lobe. However, to build a function that votes for points *near* and in front of each plane, we combine the smooth step up at the origin from the front-facing function in Figure 4.2 and the smooth step down at an appropriate distance from the near-plane function in Figure 4.2. See Figure 4.3 for an illustration.

Choosing function scales

The above examples of f_{TD}s are given without scales. In general, the output of the function ranges between 0 and 1, although some applications may wish to weight these values, *e.g.* by the area of the triangle generating the supporting plane. The width of the function is more variable, and should be determined on an application-specific basis. We consider this problem in more detail in each of the following sections.

4.4 Hough-space origin optimization

To date, none of the dual-space or Hough-space representations used for silhouette computations [39, 69, 64] has been optimized. We have observed that the location of the origin for Hough transform can drastically influence the resulting point distribution, which in turn can strongly affect algorithm performance. Furthermore, the use of *multiple origins* and an appropriate *grouping* of mesh data can lead to additional performance gains.

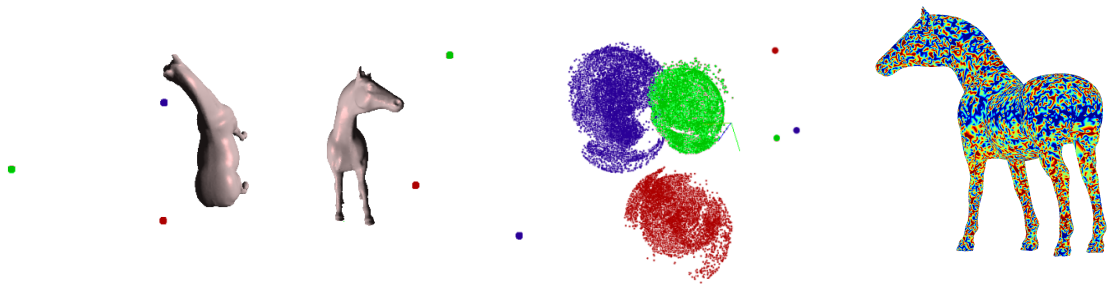
In this section, we describe a method to find these optimized origins using the voting approach described in the previous section. Substantial performance gains can be obtained for static silhouette extraction without degrading the performance of silhouette updates in any way. Note that while quick silhouette updates is crucial in an interactive setting, frame-to-frame coherence can be taken advantage of at the same time. In this sense, the initial extraction of silhouettes might offer more of a computational challenge. When the viewpoint is teleported or snapshots of a scene are taken, *e.g.* for object recognition, one needs to extract mesh silhouettes from scratch efficiently.

Please refer to Section for details of the Hough-space silhouette extraction algorithm. The crucial point to recall is that more uniform point distributions in Hough space lead to more balanced search tree data structures, which in turn result in performance gains. Thus we select multiple Hough-space origins with this goal in mind.

Recall also that we must check every face adjacent to an edge on the silhouette from the origin (hereafter *SFO edges*) explicitly when performing initial silhouette extraction. By selecting an appropriate set of origins we can substantially reduce the number of SFO edges in the mesh. If the mesh is smooth, that is, if most dihedral angles between adjacent faces are close to π, eliminating SFO edges has the additional and significant benefit of reducing the number of spurious EBV intersections, as we show below.

4.4.1 Optimizing for initial silhouette extraction

Silhouette extraction incurs most of its cost in bounding-box checks; we must check against both PBVs and EBVs. Direct optimization for origin count and positions should measure these operation counts in silhouette computation. However, modeling that problem is unrealistic as it is difficult, if not impossible, to mathematically relate operation counts with mesh geometry. We thus resort to a heuristic, which is based on an observation that there is a correlation between the number of SFO edges and the number of unnecessary EBV checks,



(a) Top view. (b) Front view. (c) Hough points. (d) Face grouping.

Figure 4.5: A horse model (39,698 triangles) and three Hough-space origins (coloured markers) selected by our algorithm are shown in two views in (a) and (b). The origins and their respective Hough-space points are shown in (c). Grouping of the mesh faces based on the origins is visualized in (d) via color coding. Extraction of the horse silhouettes takes about 2 milliseconds, compared to close to 5 milliseconds using an unoptimized single-origin Hough transform, as done in the original Hough-space silhouette extraction algorithm [64].

and between the average distance of Hough points to their origins and general performance. These issues are elaborated in the next section and the discussion motivates the use of the TDF and our voting scheme.

It follows from the definition of the Hough transform that we can change the Hough transform of a set of planes by changing the origin. We can substantially increase initial silhouette extraction performance if we select a small set of origins and assign each triangle on a mesh to an appropriate origin. These performance gains come from minimizing the number of extraneous bounding-box checks incurred.

Some visual results from our approach are shown in Figure 4.5. As we can see, good origins can be some distance away from the mesh and the corresponding grouping of the mesh faces generally do not resemble results from known mesh segmentation algorithms. Naively placing origins at the centroids of different parts of an object generally leads to poor performance for silhouette extraction.

4.4.2 Reducing SFO edges and use of TDF

Recall that the Hough transform of a face is a scalar multiple of its unit normal vector, with the scalar being the distance from the origin to the supporting plane of the face. It follows that if the dihedral angle between two adjacent faces f_1 and f_2 is θ, the angle formed by their Hough points must be either θ or $\pi - \theta$ (see Figure 4.6). This angle is strongly

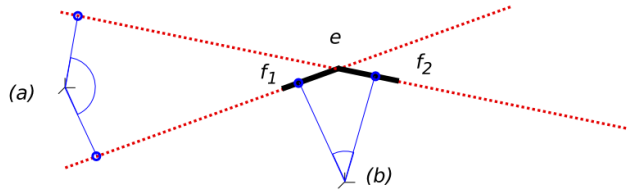


Figure 4.6: An edge e, its neighboring faces f_1 and f_2, and two candidate origins a and b. The edge is on the silhouette with respect to origin a, but not with respect to b. Note the angles formed by the Hough transforms of f_1 and f_2 from origins a and b — the smaller angle generated from b is preferred.

correlated with the probability that a v-sphere will intersect an EBV containing f_1 and f_2, whether the edge between the two faces is on the silhouette or not.

The probability that an EBV will intersect a v-sphere is not solely determined by its volume: since all v-spheres must pass through the origin, the geometry of a v-sphere depends strongly upon the orientation of the viewpoint it represents. By minimizing the angle between two Hough points, which we subsequently refer to as their *angular extent*, we reduce the angular extent of the EBVs they generate in their octree. This in turn reduces the likelihood that a randomly-chosen viewpoint's v-sphere will intersect those EBVs.

In the worst case, a poor choice of origin for f_1 and f_2 may result in an EBV that contains the origin; see Figure 4.7(a). This EBV will never be discarded by an octree traversal, as all v-spheres pass through the origin. Suppose, however, that we have a sharp edge — that the dihedral angle between two faces is close to zero; see Figure 4.7(b). In this situation, we may easily obtain a worst-case EBV if we try to eliminate the sharp edge from the SFO set. Using the methods of Section 4.3, we can choose origins which discourage both of these worst-case scenarios within the framework of our voting scheme by choosing a support distance function that penalizes candidate origins too close to supporting planes.

However, assigning a plane to its most distant origin does not by itself guarantee that the angular extent of its edges will be minimized. Furthermore, excessively distant origins tend to produce highly non-uniform point distributions in Hough space. Uniformity of point distribution is one of the greatest advantages of Hough transform for silhouette extraction. We therefore seek to keep our origins as close to the mesh as possible – but no closer.

We choose these origins by the peak-finding method introduced in Section 4.2. The f_{TD} shown in Figure 4.8 selects those points where, on average, each edge has a small likelihood

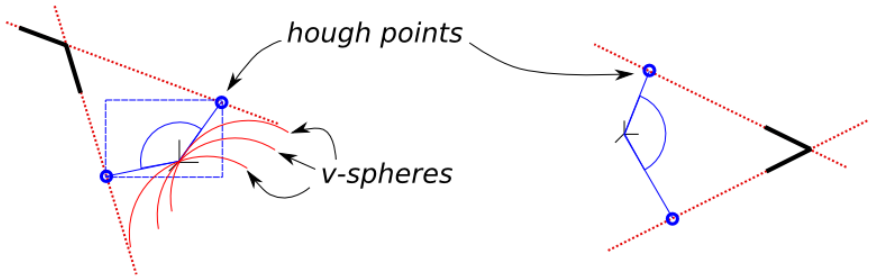


Figure 4.7: Two worst-case scenarios involving poorly-chosen origins. (a) The EBV generated by two adjacent faces includes the origin and will never be discarded. (b) A sharp edge induces a large angle between the Hough transforms of its incident faces, when it is not forced onto the SFO.

of lying on the silhouette from one of the selected origins, but is not located so far away as to unbalance any origin's resulting octree, as shown in Figure 4.5(c).

4.4.3 Domain restriction and voting scheme

We have found that restricting the domain of the voting field to a sphere of radius 3 around the normalized mesh produces excellent results. This restricted domain gives us the properties discussed in Section 4.3.1, but large enough to fully contain the peaks of the TDF.

We use the f_{TD} shown in Figure 4.8. Note that this function is parameterized by a scalar α and has range $[0, 1]$. An optimal value of α will capture not only the mathematically tractable property of minimizing silhouette size on a smooth mesh, but also the decidedly untidy property of generating a Hough transform whose points produce a well-balanced octree. However, experimentally, we have found that for the rather diverse set of mesh models in our test, an α value of about 1.72 generally produced the best performance. Figure 4.9 shows the effect of changing α on the number of bounding box checks for the hand mesh with two origins. This reveals a general trend that the algorithm performance does not vary much with α, as long as it is not too small.

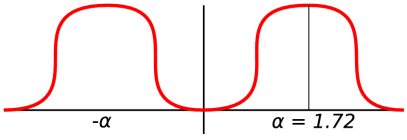


Figure 4.8: The f_{TD} for Hough space origin optimization.

4.4.4 Origin selection and face grouping

In our current implementation, we select a user-specified number k of origins based on the TDF defined above. The origins are selected one at a time in a greedy fashion. After one origin is selected, the scalar field is updated to remove its influence. An obvious question that arises is how large k should be. Indeed, multiple origins ($k > 1$) tend to improve performance but only to a certain extent. A large value of k tends to reduce the SFO set, and consequently reduces the number of unnecessary leaf-level EBV checks, but increasing the number of octrees increases the overhead of bounding-box checks near the roots of the trees. We find that values of k between 2 and 4 work well for most meshes.

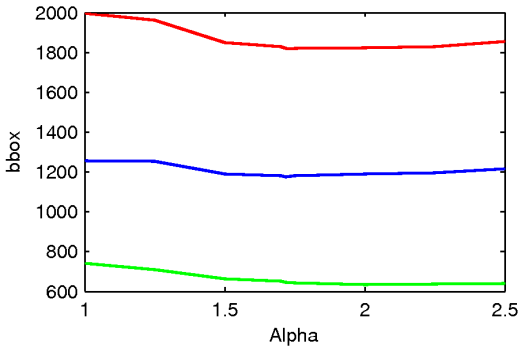


Figure 4.9: Number of bounding box (bbox) checks (green: PBV; blue: EBV; red: total) vs. α for the hand mesh with two origins.

Automatically choosing an optimal k is a nontrivial undertaking; this problem bears similar characteristics to choosing the right k in k-means clustering. Given sufficient processing time, one can always test a sequence of values for k and for each k perform the origin selection heuristic and rely on performance measurements to choose an appropriate k. In fact, this has been a well-practiced heuristic for choosing the number of clusters in clustering analysis [29].

The origin selection algorithm above is concerned only with distances from origin to face supporting planes, or equivalently, distances from origin to Hough points. As previously noted, such distances alone offer no guarantee that we can minimize the angular extent of Hough point pairs contributing to EBVs. Rather than assigning faces to the origins with which they score the highest, we first consider whether each origin would put any edges with low dihedral angle on that face on its silhouette, then use face distance as a tiebreaker.

We define the *SFO count* of a face f with respect to an origin o as the number of edges of f that are on the silhouette from o, not on the mesh boundary, and have dihedral angle less than $\pi/2$. Naturally, the dihedral angle at an edge is formed by its two adjacent faces. If f has a nonzero SFO count with respect to o, we expect that at least one of the edges on f will incur an EBV with a large angular extent from o. We therefore assign f to the origin

that minimizes its SFO count. In the event that f has the same (minimal) SFO count with respect to more than one origin, we assign f to the origin from which it is most distant.

4.4.5 Multi-origin silhouette extraction

Having found a set of k origins, we can now construct an augmented octree for each origin and its associated group of faces. However, for each group C, we must take care to account for faces that are not present in C but are *adjacent to faces that are* in C. If we were to ignore these faces during octree construction for C, we would not create the proper EBVs, and would risk not identifying these "group-crossing" edges during the initial silhouette extraction step. To account for these faces, we create "ghost points" corresponding to them while creating the EBVs for C. These ghost points are not present in the octree for C and do not contribute to its PBVs, but they do affect EBVs.

To find the static mesh silhouette with respect to a viewpoint v, we must first check each SFO edge, as before. We are only obliged to do so if it is on the silhouette from all the origins of the groups to which its adjacent faces have been assigned. This eliminates the vast majority of explicit SFO edge checks. We can now traverse the octree for each group independently, as in Chapter 3. Before doing so, we must transform the global viewpoint into the space of each cluster in order to calculate the correct v-sphere. Since a change of origin can be interpreted as a translation, this is a simple matter of subtracting the cluster origin from the viewpoint.

4.4.6 Experimental results

We tested our algorithms on a PC running Linux 2.6.18 with four Intel Xeon 3GHz processors and an NVidia GeForce 6800 Ultra graphics card. Six models, listed in Table 4.2, were used, with face counts ranging from 12K to 268K. Note that these same models were also used in the experiments of Chapter 3. In addition to reporting the number of bounding box checks, seen as the atomic operations of the algorithm, we also give timing for silhouette extraction. One should keep in mind however that our code had not been optimized and such timing results are only meant to provide a general impression of the speed of our algorithms.

We tested initial silhouette extraction performance for optimized Hough spaces by selecting 3,865 viewpoints evenly distributed over three spheres, all centered about the mesh centroid and having varying radii from 1 to 3. Recall that we have scaled the test meshes

			Unoptimized		**Optimized; multi-origin**			
Model	No. faces	Avg. Sil.	Bbox checks	Time (ms)	Origins	Bbox checks	Time (ms)	Time speed-up
Hand	12,378	639	5,691	1.45	2	1,818	0.44	3.29
Horse	39,698	2,511	16,049	4.85	3	6,884	1.96	2.47
Bone	65,001	4,724	30,825	10.70	2	11,816	3.78	2.83
Bunny	69,452	3,498	20,376	5.55	3	9,073	2.46	2.25
Dragon	200,000	13,666	72,472	17.20	3	27,767	7.04	2.44
Igea	268,686	7,191	36,757	16.60	4	20,268	7.36	2.25

Table 4.2: Performance statistics for our Hough-space origin optimization algorithm ($\alpha = 1.72$). Both the total number of bounding box (Bbox) checks and average silhouette extraction times (in milliseconds) are given. We also list the optimal number of origins, found by the heuristic described in Section 4.4.4.

to fit within the unit sphere. Results and comparison with extraction using single-origin Hough transform [64], where the origin is chosen as the mesh centroid, are summarized in Table 4.2. The relatively high cost for the bone and dragon meshes is due to the higher complexity of their silhouettes.

Our program requires anywhere between two minutes (for the hand) and seventy-five minutes (for the Igea) to find a set of optimal origins, depending on the size of the mesh and the number of origins required. Naturally we would like to improve upon this, but for a preprocessing step this is still acceptable and it measures up favorably against the cost of other optimization approaches, such as that of Sander *et al.* [75]. The best choice for the ideal distance α is likely model-dependent. We choose $\alpha = 1.72$ throughout this set of experiments, as discussed in Section 4.4.3. The optimal number of origins k varies across the test models, but remains relatively small.

4.5 Viewpoint selection

We approach the viewpoint selection problem from an essentially perceptual point of view: we wish to select a small set of k viewpoints — perhaps three or four — which together provide as complete a visual description as possible of the target mesh. This is important for users browsing large mesh databases, where a small number of low-resolution thumbnail images may be generated for each entry and should allow the user to identify the mesh.

4.5.1 Previous Work

The viewpoint selection problem has applications ranging from image-based modelling [82] to object recognition [88] and is also studied in robotics as the sensor placement problem [79]. Most approaches define a view quality metric and evaluate it over a set of candidate viewpoints, essentially via an exhaustive search, choosing the candidate with the highest score as the best view. An overview of these approaches is provided by Polonsky *et al.* [68]. Recently, reflective symmetry information has been used to select single high-quality viewpoints [67]. Our approach using TDFs provides comparable or better quality viewpoints and at reduced costs, as a TDF allows for more efficient viewpoint search.

Camera-path planning is a related problem. It has been studied in the context of motion planning and path finding [3], as well as in digital cinematography [16]. We consider camera-path planning in the same context as viewpoint selection – selecting a camera path which conveys as much information as possible about a given model based again on silhouettes. Our approach also leads to a straightforward method for planning such a camera path.

4.5.2 Selection criteria

Silhouettes are important perceptual cues in object recognition [52]. When humans sketch objects, they tend to define features by their silhouettes. Therefore, we seek to maximize the perceptual information provided by a set of viewpoints by choosing those viewpoints to maximize the length of the mesh's silhouette from each. This problem is difficult to formulate in terms of supporting planes. However, most large meshes are composed of triangles of similar size. We assume that the lengths of the input mesh's edges are consistent and close to uniform – if this is the case, we can reduce the problem of measuring silhouette length to the problem of counting silhouette edges. This problem can be formulated in the framework we have described.

In addition, we follow an observation of Vazquez *et al.* [82] that faces seen at orthogonal or nearly orthogonal viewing angles contribute greatly to view entropy, their measure of viewpoint quality. With this in mind, we also wish to select viewpoints that can see a large number of faces at large viewing angles. Again, this is simple to integrate with silhouette selection in our voting framework.

In general, we are not concerned with back-facing triangles in the viewpoint selection problem. The silhouette selection function from Section 4.3 will inevitably give heavily-weighted votes to back-facing planes near the silhouette, but as this reinforces points that

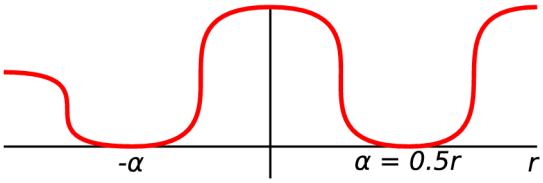


Figure 4.10: The f_{TD} used for viewpoint selection. The peak in the centre favours planes near the silhouette, while the plateau at $+x$ favours front-facing planes with orthogonal viewing angles. The shallower plateau at $-x$ slightly favours back-facing orthogonal planes, encouraging the use of symmetry without overwhelming the other factors.

generate a large silhouette it is not undesirable. However, a large number of back-facing planes orthogonal to the viewing direction, when combined with a similar number of front-facing orthogonal planes, tend to indicate a reflective symmetry plane. Podolak *et al.* [67] point out that symmetry information is perceptually useful for viewpoint selection. We therefore give a small weight to back-facing planes with orthogonal viewing angles, to take advantage of reflective symmetry in the input without overwhelming the above constraints.

4.5.3 Domain selection and voting scheme

We select viewpoints within a spherical shell around the mesh, with inner radius $\sqrt{3}$ and outer radius 2. This is sufficient to fit the whole mesh within our view frustum while not placing the camera too far away. As silhouette size can vary with distance, we search over a thick shell rather than a set of points at constant radius like most other methods. Our final f_{TD} is shown in Figure 4.10. We have set the α parameter to 1, half of the outer radius. This ensures angle-selecting behaviour as shown in Figure 4.4.

We have found experimentally that on most "object" meshes, points that maximize total silhouette size also have large visible silhouettes. However, we can easily account for any discrepancies introduced here by further weighting the f_{TD} by the ratio of visible silhouette pixels to total silhouette pixels. At each point, we render the silhouette to a buffer, first with the depth test enabled to find the visible silhouette, then with the depth test disabled to find the total silhouette. We calculate the total votes cast for the point in question as usual, and scale the number by the ratio of visible to total silhouette pixels. This does not drastically change the positions of the selected viewpoints, but does bias them slightly to make more features visible. See Figures 4.11 and 4.12.

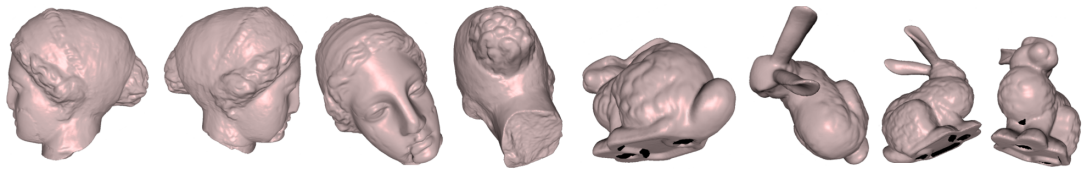


Figure 4.11: Four highest-scoring views for the bunny and igea meshes. As the results for the bunny demonstrate, our method produces good views even for meshes with holes or borders.

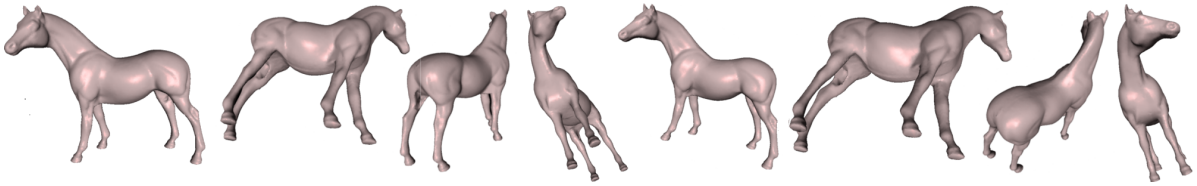


Figure 4.12: Four views selected for the horse mesh with (left) and without (right) weighting by the visible silhouette ratio. Note that the results are similar to those shown in Figure 4.11; however, in the third and fourth views, one leg is occluded.

4.5.4 Experimental results

The object views shown in Figure 4.11 show several improvements over the methods summarized in [68]. For example, the results presented here show each model from a variety of well-separated viewing angles, rather than selecting a number of spatially similar viewpoints. This is similar to the result achieved by [88], but does not require graph partitioning as in that work. Our method also avoids viewpoints positioned opposite each other across the model's planes of symmetry, correcting a flaw noticeable in the purely silhouette-based algorithms detailed in [68].

Figure 4.13 shows views generated for simplified versions of the horse shown in Figure 4.12. These views are nearly identical to those produced for the full-resolution horse; the only visually apparent difference is the fourth view chosen for the lowest-resolution mesh. This suggests that we can use the smoothing nature of the TDF to generate views for high-resolution models from a simplified input.

Figure 4.14 shows a wolf model in two different poses. Note that the views generated differ significantly between the poses and from the structurally similar horse mesh, highlighting the differences between the meshes.

Finally, Figure 4.15 shows the output of our algorithm on a CAD-type mesh, in this case

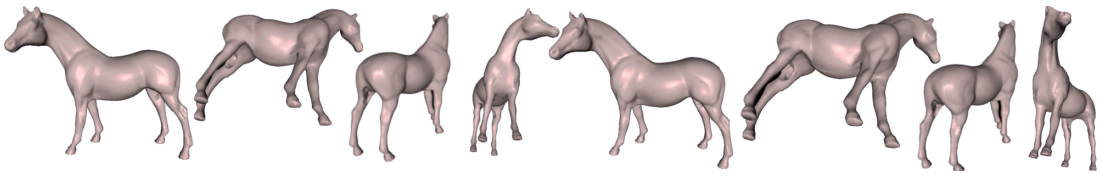


Figure 4.13: Four viewpoints selected for simplified horse meshes with 10k (left) and 20k (right) faces. The results are nearly indistinguishable from the viewpoints found for the full resolution mesh with over 39k faces.

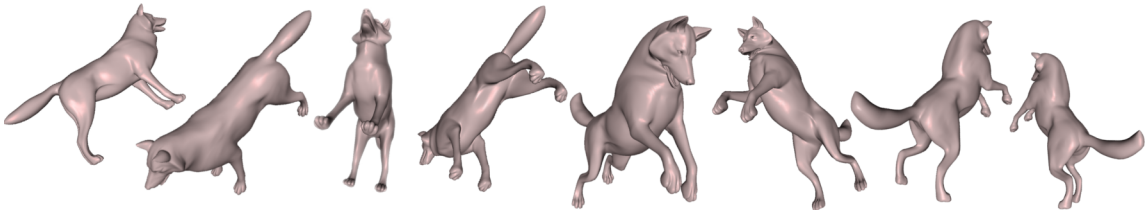


Figure 4.14: Four views selected for two poses of the wolf mesh. Views differ between the poses as the visually significant parts of the mesh change.

a marching-cubes reconstruction of the fan blade. The large flat bottom of the fan blade dominates the TDF for the first viewpoint, producing a view that by itself is insufficient to capture the whole shape. However, as the faces on the flat bottom are penalized in subsequent iterations, the silhouette length criterion takes on increasing importance. The chosen set of four views shows all sides of the mesh from well-separated viewing angles.

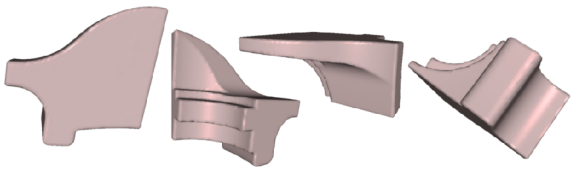


Figure 4.15: Four views for the fan blade mesh, from first choice to fourth from left to right. The mesh is readily identifiable even though the first view is dominated by a large flat region.

While our method chooses well-separated and informative viewpoints, it does not compute a natural orientation for the camera. This is an interesting and difficult problem in its own right which is beginning to be addressed, for example by Fu *et al.* [30].

4.5.5 Camera path planning

Certain applications will benefit from an automatically-generated camera path, providing a view of the whole model while focusing on the most visually significant parts. Generating

such a path from our viewpoint selection TDF is efficient and straightforward. We construct a path that interpolates all of the selected viewpoints in a way that seeks to maximize the significance of intermediate viewpoints.

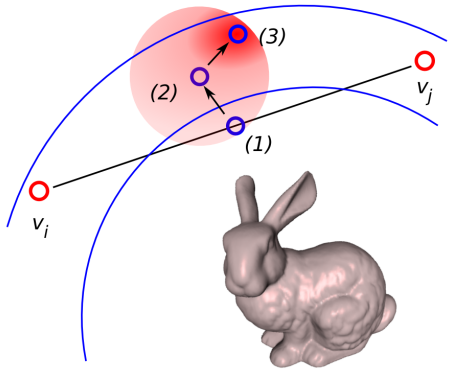


Figure 4.16: Generating an intermediate path node between two viewpoints v_i and v_j. We start with (1) the point between v_i and v_j, then (2) project it into the TDF's restricted domain, and (3) move it to a nearby peak.

We first discuss *intermediate path nodes*. These nodes serve two purposes: they keep the camera path between viewpoints from intersecting the bounding sphere of the viewed model, and they guide the path to local maxima or ridges in the TDF, representing points with interesting views. To find the intermediate path node p_{ij} between two viewpoints v_i and v_j, we simply take the midpoint of the line segment $\overline{v_i v_j}$, extend it to the middle circumference of the TDF domain, and move it to the peak TDF value within a small neighbourhood, as shown in Figure 4.16.

Note that we use the initial TDF to compute intermediate point positions, rather than the modified TDF used to compute the second and subsequent viewpoints. The initial, unbiased TDF provides an overall metric for the significance of every viewpoint around the mesh, while the later, modified TDFs reduce the contributions of planes that are satisfied by previous viewpoints.

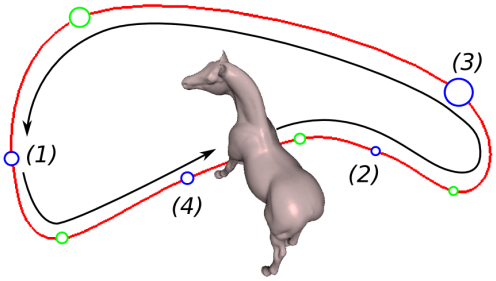


Figure 4.17: Camera path generated for the horse mesh based on the four viewpoints chosen by our viewpoint selection algorithm, as shown in Figure 4.11. Viewpoints are shown in blue, intermediate path nodes in green.

We can now compute a camera path as a sequence of alternating viewpoints and intermediate path nodes. Starting at the first viewpoint v_1, we select the destination viewpoint v_d that maximizes the value of the TDF at the intermediate node p_{1d}. We add p_{1d} and v_d to the camera path, remove v_1 from consideration, and repeat the process from v_d. When we have visited all viewpoints, we return to v_1, closing the path. When following

the path, we interpolate between the path nodes using cubic Bézier curves. By applying the methods from Chapter 4 of [78] – namely, by ensuring that control points across an endpoint are collinear and symmetric – we can ensure a smooth path between viewpoints.

4.6 Placement of single light source

Like silhouette arcs, lighting information is a powerful perceptual cue that can add significant visual information to a scene. The cues provided by a light source depend upon the location of the viewpoint and the properties of the mesh. Given a viewpoint (perhaps chosen by our algorithm from Section 4.5), we would like to place a directional light source to provide as much visual information as possible.

4.6.1 Previous work

Light source placement is generally studied alongside the viewpoint selection problem, and has applications ranging from edge-based object recognition [17] to perceptual scene enhancement [35]. As with the viewpoint selection problem, light source placement algorithms generally define a quality metric and evaluate it over a set of candidate positions. Quality metrics can be quite similar to those for viewpoint selection. For example, [35] and [81] use an entropy measure similar to [82]. Unlike viewpoint selection, however, the amount of information conveyed by a given light source placement can be affected by the light's illumination parameters (diffuse and specular values, and shininess exponent), and placing multiple light sources can give drastically different results from each light taken alone.

4.6.2 Placement criteria and $f_{\mathbf{TD}}$

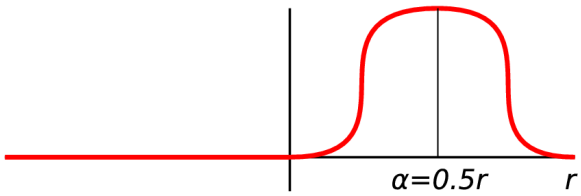


Figure 4.18: The f_{TD} for single light source placement.

We wish to light the surface facing the viewer with as much variation in intensity as possible to highlight even small changes in angle. Within the Phong lighting model, this involves minimizing the angle of incidence from the light source to each of the planes facing the viewpoint. We wish to avoid large lighting angles, which will wash out the small features we wish to highlight. However, we must also minimize the size of the

silhouette from the light source: silhouette loops enclose back-facing regions, which will not be lit (except by the light's ambient component) and thus give no information by lighting. This is almost exactly the opposite of our viewpoint selection criteria. We thus use the f_{TD} in Figure 4.18 on the same restricted domain used for viewpoint selection.

Note that the f_{TD} chosen for light source placement attempts to model the information provided by the light shining on a surface, and is thus tied to the parameters of the light. In this case we have chosen a matte material and a light with strong diffuse and weak specular components, in order to maximize the information conveyed by shading and minimize the saturating effects of specular highlights. See Figure 4.19 and compare to the other renderings in this thesis. It may be possible to generate a light-placement f_{TD} automatically from a set of material and light parameters; this is a direction for future research.

There are two major differences from the basic algorithm outlined in Section 4.2 in our approach to light source placement. First, rather than considering every supporting plane in the mesh, we operate only upon planes that face towards the viewpoint. Second, we are forced to discard the greedy next-best-point selection algorithm. In the other applications, once a plane has selected a point in which it has a high interest, it is unaffected by the selection of subsequent points. In this case, however, we may inadvertently select a new light position which washes out the contrasts which we have emphasized with our first. Hence, we are only able to place a *single* light source. Adapting our method for robust multiple light source placement, including control over intensity, is a topic for future research.

4.6.3 Experimental results

We show light placement results for the horse mesh in Figure 4.19, compared to an arbitrary light source placement from Section 4.5. We placed light sources for each of the four best viewpoints chosen by our algorithm.

Consider the regions highlighted by circles in Figure 4.19. First note the near-total lack of shadowed (ambiently lit) regions in the bottom row of images. By comparison, the top row of horse images show a great deal of shadowing. This obscures detail of the horse's left hind leg in the second view, and much of the horse's sides in the third and fourth views. In all views, the musculature of the horse is more apparent, particularly on the neck, forelegs, and hindquarters, in the bottom row of images. Finally, in the fourth view, the unoptimized light source washes out detail on the side of the horse's face and neck, whereas the optimized light position shows this detail clearly.

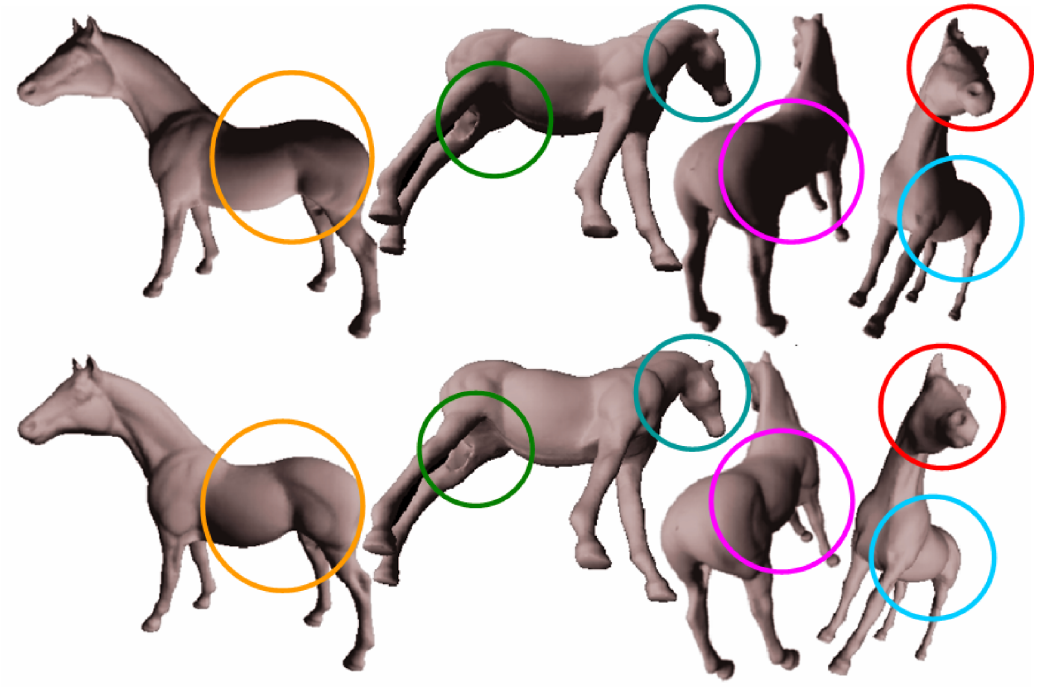


Figure 4.19: Light source placement results for the horse (bottom) compared to an arbitrarily-placed light source (top). Colored circles indicate corresponding highlighted regions where we can observe differences made by our algorithm.

4.7 Conclusions

In this chapter we have stepped back from the geometric silhouette of a model to consider the information that can be obtained by examining its supporting planes in a global, rather than local, sense. We have constructed a *tangential distance field* and shown that it serves as a unified framework for approaching point-selection problems based on planarity- and silhouette-related characteristics of a mesh. This emphasizes the utility of supporting-plane representations for global as well as local geometry analysis.

One factor that must be considered is that the methods developed so far operate only on *mesh* data. Surface models obtained from video capture techniques such as shape from motion, or from increasingly ubiquitous laser range scanners, must be reconstructed into meshes before these techniques can be applied. In the next chapter we construct local SGSes for samples in point clouds and use them to extract silhouettes without global reconstruction.

Chapter 5

Point-set silhouette extraction

As laser scanners proliferate, geometry processing algorithms can no longer depend upon receiving a complete mesh as input. While a wide variety of full remeshing schemes exist to convert noisy and incomplete scanned data into clean meshes suitable for traditional algorithms, not all problems merit so powerful and expensive a technique. Silhouette extraction and processing in particular can benefit from a fast and approximate *local* reconstruction method which nevertheless provides sufficient geometry and connectivity to obtain an accurate result. In this chapter we derive such an algorithm from an intuitive definition of the silhouette of a point-set surface and show that its results surpass those available from existing techniques; see Figures 5.1 and 5.2. We also demonstrate that the local reconstruction produced by our method is suitable for applying other traditionally mesh-based algorithms to point clouds.

5.1 Silhouettes and silhouette-generating sets

While silhouette sets are well-defined on polygon meshes and smooth surfaces as structures that separate front- and back-facing regions, as formulated in Definitions 2.1.2 and 2.1.3, it is difficult to extend these definitions to point clouds. Instead, we extend the *SGS* of Definition 2.1.5 to point clouds. Recall that on smooth surfaces, the SGS definition of the silhouette is identical to the others, and we will see that it can readily be extended to both meshes and point clouds.

We have seen that the SGS of a mesh *edge* is identical to the double-wedge volume defined by its adjacent triangles. For a mesh *vertex* the SGS is defined similarly, using the

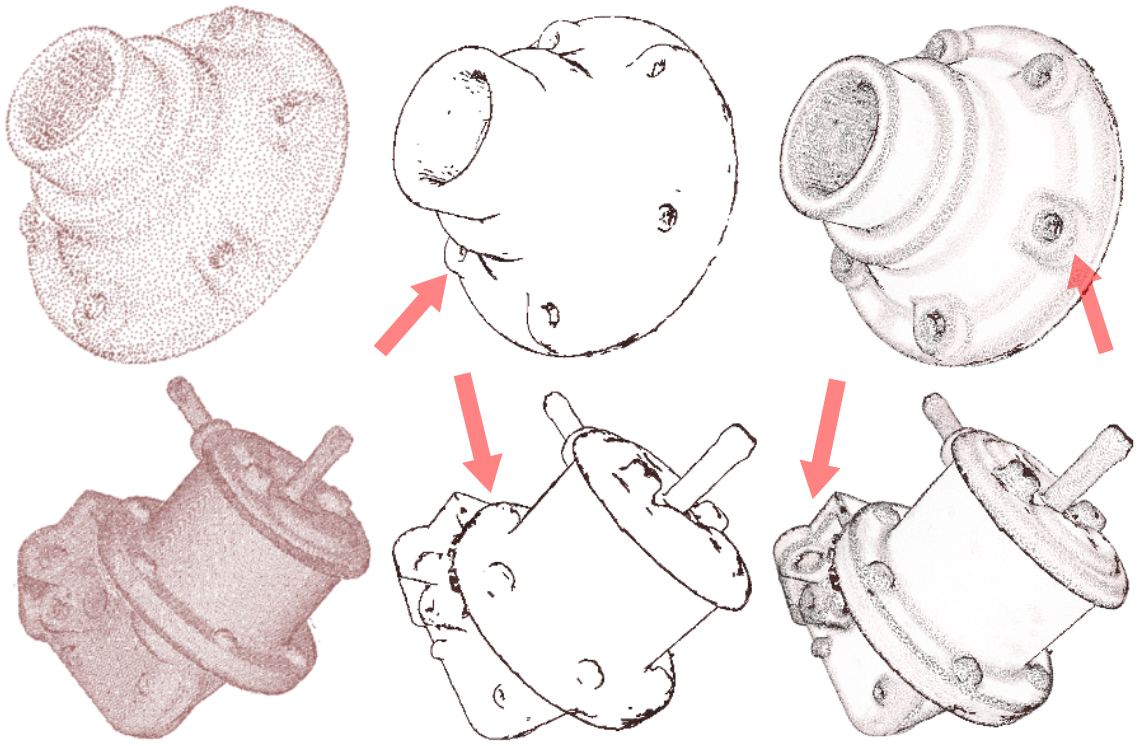


Figure 5.1: Surface features in unprocessed point clouds are difficult to visualize, even with visibility resolved (left). By rendering silhouettes (middle) and especially detected sharp features (right), some geometric details of the underlying shapes are better revealed.

planes of its umbrella triangles. A mesh vertex is on the silhouette only when at least one of its adjacent edges is on the silhouette; thus, the SGS of the vertex is the union of the SGSes of its adjacent edges.

Intuitively, to define the SGS of a point in a cloud we want to construct a local umbrella around it which approximates the underlying surface on which it was sampled. More formally, in order to define the SGS of a point p in a point cloud P, we consider its relationship to the underlying surface S. We assume that all points in P are on S; this in turn induces an intrinsic Voronoi diagram on S from the point samples. This gives us an intuition for point-cloud silhouettes: a point $p \in P$ should be on the silhouette when the silhouette curve on S passes through p's Voronoi cell; see Figure 5.3. Therefore, the exact SGS of p is the union of all planes tangent to points on S within p's Voronoi cell. Note that we do not need the Voronoi cell itself. As it is impractical to construct the exact SGS of p based

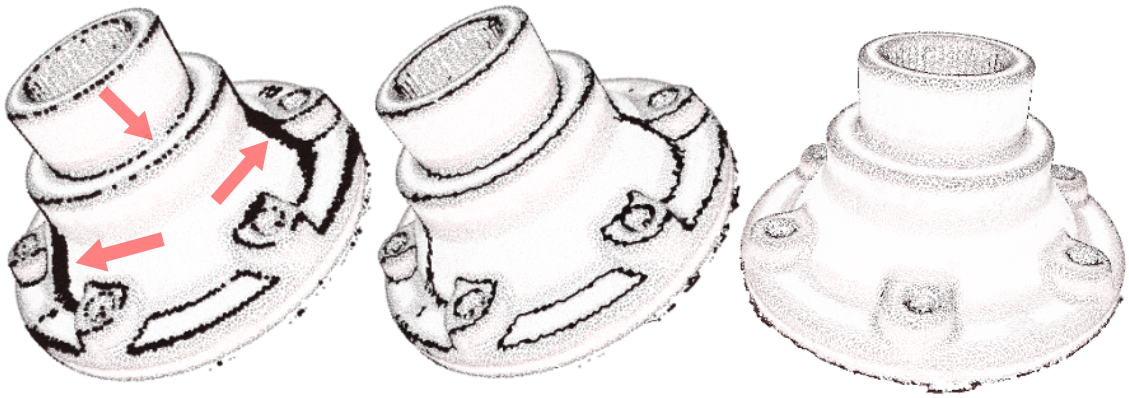


Figure 5.2: Normal thresholding (left) can over- and under-detect point set silhouettes. Results using our method (middle) based on SGS and local reconstruction show visible improvement on silhouette accuracy. The model is also shown from p_s (right).

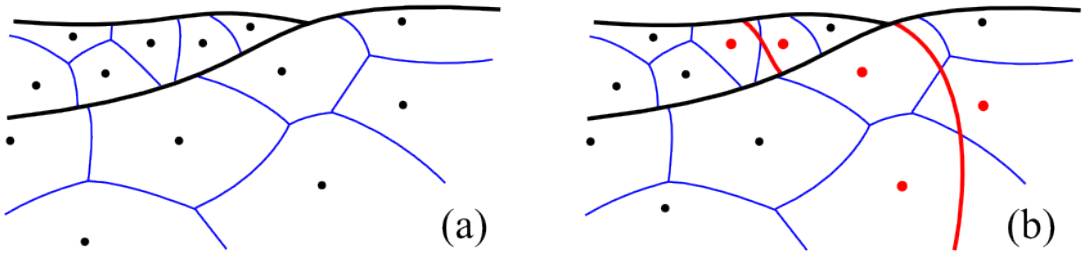


Figure 5.3: Finding point samples on a surface's silhouette. (a): Point samples on an underlying smooth surface S and their intrinsic Voronoi cells. (b): A silhouette curve on S. The points whose Voronoi cells are crossed by the curve (highlighted) are on the silhouette.

on this definition, we next present an approximate construction and show that it leads to high-quality silhouette extraction.

5.2 Local neighbourhood construction

To approximate the SGS of a point p in a point cloud P according to the definition presented in Section 5.1, we build an intrinsic Delaunay triangulation [25] from a subset of p's k-nearest neighbours $Q_k(p)$. We need not compute a full triangulation; any points in $Q_k(p)$ not in p's Delaunay one-ring will not affect the SGS of p in the local reconstruction and can be ignored. The supporting planes of these triangles approximate the tangent planes of the intrinsic Voronoi cell on the underlying piecewise-smooth surface S containing p, and thus describe p's SGS.

Our local triangulations are constructed in a series of four steps, each taking advantage of information obtained from the previous steps to produce a more accurate umbrella around p. We perform the following operations:

1. Normal estimation and neighbour filtering
2. Initial umbrella creation and boundary identification
3. Neighbour-based multi-umbrella creation
4. Boundary consistency enforcement

Next we describe each of these steps in detail.

5.2.1 Normal estimation and neighbour filtering

In the simplest case, p lies within a smooth region of S and the intrinsic Delaunay umbrella of $Q_k(p)$ will produce an appropriate local reconstruction. However, when some members of $Q_k(p)$ are *not* in the same region of S – if they lie across a sharp feature edge, or on a close-by surface sheet – we must exclude them from our triangulation. Our first tool to achieve this is local normal estimation.

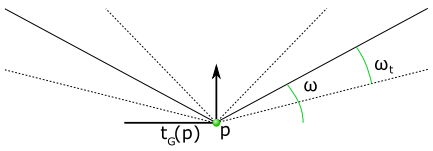


Figure 5.4: Angle bounds for neighbour filtering. Members of $Q_k(p)$ that fall within the wedge with half-angle ω_t are considered *marginal*, and must satisfy a distance constraint to be selected in the first pass.

We consider first this simple case, with p in a smooth region of S, relatively far from any feature curves. Let t be a triangle on p with normal $\boldsymbol{n}_t$, and let $\boldsymbol{n}_p$ be the normal to S at p. The acute angle between the lines generated by $\boldsymbol{n}_t$ and $\boldsymbol{n}_p$ is bounded by $\mathcal{O}(r_t/\rho_f(p))$, where r_t is the circumradius of t, and $\rho_f(p)$ is the *local feature size* at p: that is, the distance to the medial axis of S [22, Lemma 3.5].

We say that a smooth region $U \subset S$ is *well sampled* if any point $x \in U$ is closer than $\epsilon\rho_f(x)$ to the nearest sample point, where ϵ is an appropriately small constant. The function $\rho(x) = \epsilon\rho_f(x)$ is the *sampling radius*.

To estimate the local normal at p, we find the triangle with the smallest circumradius amongst those that have both the point p and its nearest neighbour as vertices. This triangle,

which we denote $t_G(p)$, is necessarily a *Gabriel triangle*; its smallest open circumball is empty of sample points [24, Lemma 4.12]. Since this canonical Gabriel triangle is the only one of interest to us, we take the liberty of referring to it as *the* Gabriel triangle. If q is the nearest neighbour to p, Mederos *et al.* [61] identify t_G as the triangle $[p, q, u]$ that has the largest (necessarily acute) angle at u. Appendix A of [63] shows that this triangle provides a good approximate normal in a well sampled smooth surface patch.

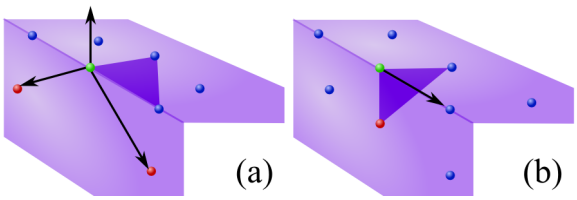


Figure 5.5: Filtering on the Gabriel normal at a point p (green) on a sharp edge may (a) include points on the opposing surface which pass the ω-test; also, (b) the Gabriel triangle itself may cross the edge.

If p lies in the interior of a well-sampled smooth surface patch, all samples in an umbrella on p should lie close to the tangent plane of p. The sampling radius ensures that $\|p - q\| = \mathcal{O}(\epsilon)\rho_f(p)$ for any neighbour q of p. The angle between $\overline{pq}$ and the tangent plane to p is $\mathcal{O}(\epsilon)$, and does not vary with the local feature size [22, Lemma 3.4]. Similarly, the angle between the tangent plane and the plane of t_G is $\mathcal{O}(\epsilon)$. This motivates our use of a constant angle threshold, ω, to filter the points in $Q_k(p)$ which are candidates for being neighbours on the same surface patch. This angle threshold reflects the implicit parameter ϵ governing the sampling radius.

Given the canonical Gabriel triangle, we discard all edges that form an angle greater than ω with the triangle's plane. In smooth areas of the surface this filtering discards most unrelated samples, such as those coming from close-by surface sheets.

When $Q_k(p)$ lies in a well-sampled region, filtering by ω produces a high-quality set of neighbours and is sufficient by itself. However, when $Q_k(p)$ does not conform to our assumed sampling density, we may inadvertently select points which are barely within the cone described by ω but geodesically distant. Further, if the surface curvature is high, we may also *reject* desirable neighbours which are barely outside of the ω cone. To address these cases, we introduce another parameter $\omega_t < \omega$ to describe *marginal* edges where our confidence in the ω criterion is weaker. If the angle between an edge pq and the plane of $t_G(p)$ falls within $[\omega - \omega_t, \omega + \omega_t]$, we accept q only when $|pq|$ is sufficiently small; we find that the condition $|pq| < \gamma r_{t_G(p)}$ works well in all of our examples, where γ is introduced in Section 5.2.3 to bound the circumradii of triangles in an umbrella.

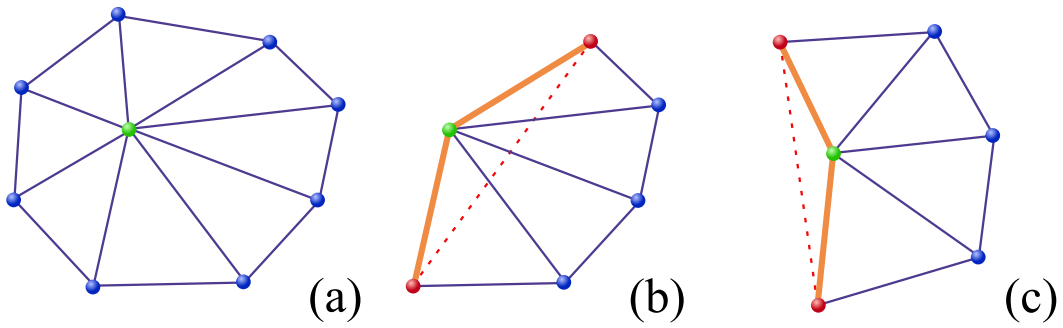


Figure 5.6: Boundary detection in the initial triangulation. Most initial triangulations have no boundaries (a). When the distribution of $Q_k(p)$ is severely biased, fold-overs may occur (b); these create *convex* boundaries (orange). Even if fold-overs do not occur, *concave* boundaries are created (c) when a triangle's angle on p exceeds φ. Dashed red lines are removed from the umbrella.

Filtering is even more challenging in the presence of sharp feature edges, as shown in Figure 5.5; these edges can be seen as an extreme case of undersampling. First, even filtering by both ω and ω_t may keep points on surfaces across sharp (approximately right-angle) edges, requiring an extra filtering step (Section 5.2.2). Second, the Gabriel triangle *itself* may include samples on both sides of the edge. In the latter case, only a small proportion of neighbours will pass the Gabriel normal filter; such points are dealt with in the second phase of our algorithm, described in Section 5.2.4. If we trust $t_G(p)$, we sort the remaining points by angle in its supporting plane and proceed as below.

5.2.2 Initial umbrella creation

At this stage we have an estimate of the relevant neighbours in $Q_k(p)$. The process of constructing an umbrella for p also drives our boundary detection. We work with the radial edges from p to its neigbours. We sort them in counterclockwise order according to their projection on the plane defined by t_G, thus defining an umbrella at p. See Figures 5.6 and 5.7(a) for examples.

We must account for the possibility that p itself lies on a boundary between surface patches. In this case our initial umbrella will be a *partial umbrella* on the patch containing the Gabriel normal. The construction of the remaining surface patch(es) is described in Section 5.2.4. Some boundaries are identified in the initial triangulation; we describe boundary detection in Section 5.2.3.

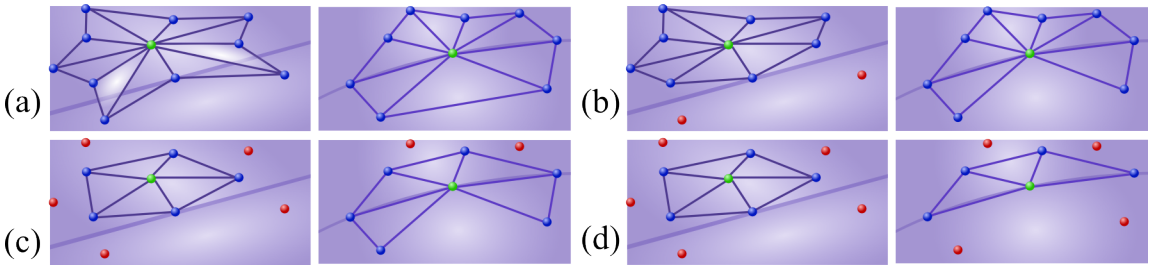


Figure 5.7: Umbrella creation near (left) and at (right) a feature line. We perform (a) triangle removal, (b) boundary detection, (c) Delaunay edge flipping, and (d) boundary expansion. See text for details.

This in turn lets us make a second, more aggressive pass on the remaining neighbours, as shown in Figure 5.7(a): we discard any edges whose *adjacent triangles' normals* form an angle greater than ω with the Gabriel normal. However, we do not remove an edge if the resulting triangle would also fail this criterion. (Removing an edge is akin to an edge flip, but we only preserve triangles incident to p.) We also avoid removing boundary edges, as defined in Section 5.2.3. This process eliminates most samples remaining on other surface patches, as well as distant neighbours on smooth surfaces with relatively high curvature.

Next we apply the extrinsic Delaunay edge-flipping algorithm from [25], removing edges adjacent to p which are not locally Delaunay, shown in Figure 5.7(c). At each step, we examine p's one-ring for newly-created concave boundaries; boundary edges are never flipped.

5.2.3 Boundary detection

Our boundary detection is based upon an additional sampling assumption. A *local uniformity constraint* is some criterion that limits the number of samples that can appear in a small region. Any algorithm that attempts to create an umbrella on p using only points from $Q_k(p)$ is at least implicitly assuming some local uniformity constraint. Otherwise, all k nearest neighbours could be confined to a tiny disk close to p such that no reasonable umbrella could be constructed.

We express our local uniformity assumption as a bound on the minimum distance between sample points. We assume $\|e_G\| > \delta\epsilon\rho_f(p)$, where e_G is a *Gabriel edge* as defined in [63], $0 < \delta < 1$, and ϵ governs the sampling radius, as above.

A good triangle t on p has its circumradius bounded by $\mathcal{O}(\epsilon\rho_f(p))$, and by a straightforward geometic argument (as shown by Kil and Amenta [50], for example), the largest

angle in t is bounded above by $\alpha = \pi - \mathcal{O}(\delta)$ Thus the largest angle in any triangle is governed by a constant parameter that is independent of the local feature size. We call this parameter φ.

Before performing Delaunay edge flipping, we first check for boundaries (Figure 5.6). *Concave* boundaries are identified with triangles whose angles on p exceed φ. *Convex boundaries* are defined by triangles whose angle in the counter clockwise ordering around p exceeds π, as in Figure 5.6(b). We mark these explicitly here to ensure correct behaviour from the edge-flipping algorithm in later steps.

We also use excessively large triangle circumradii as indicators of the presence of a boundary. We should have $r_t < \mathcal{O}(\frac{1}{\delta})\|e_G\|$, where $\|e_G\|$ is the distance to the nearest neighbour of p. However, we have found this particular method of detecting boundaries to be too sensitive to variations in the sampling uniformity. Instead, we have had better success requiring $r_t < \gamma r_{t_G}$, where γ is another constant parameter whose value reflects an expectation on the local uniformity of the sampling.

Finally, if any boundary edges have been detected, we attempt to *enlarge* them by examining the circumradii of the associated triangles, as in Figure 5.7(d). If a triangle t on a boundary edge has $r_t > \gamma r_{t_G}$, we disregard the sample on that edge, and make the other edge of t incident to p a boundary edge.

5.2.4 Alternate normals and supplemental umbrellas

We now have an estimate of the local surface around each point with a trustworthy Gabriel triangle. If that point is on a smooth surface patch, we expect it to have a triangulation without boundary. However, if p is on a boundary between smooth surface patches, the umbrella we have just constructed will only inform us about a single patch, and we must build umbrellas on its other adjacent patches using information from neighbouring points. Furthermore, if the Gabriel triangle, $t_G(p)$ is untrustworthy, we must obtain an estimate of p's normal from one of its neighbours. We decide that $t_G(p)$ is untrustworthy if fewer than half of the points in $Q_k(p)$ make an angle smaller than ω with the plane of t_G.

We address both problems in a second pass over the input, this time considering both points whose umbrellas have a boundary and points with untrustworthy Gabriel triangles. In either case we proceed as before, with the following modifications.

Rather than obtain a normal estimate from $t_G(p)$, we instead choose the computed normal from a *trustworthy* neighbour of p. This is a point in $Q_k(p)$ that has a single

umbrella without boundary. The closest trustworthy neighbour to p gives us a trusted normal even when $t_G(p)$ is not reliable.

Filtering $Q_k(p)$ on this normal is more restrictive when p already has a partial umbrella. We reject points that would be admissible in an already-constructed umbrella, *unless* they lie on that umbrella's boundary (indicating that they too lie on feature edges). However, if a point lies on the boundaries of two partial umbrellas, it cannot lie on a third if S is manifold, and therefore it must be rejected.

We build partial umbrellas using the algorithm of Section 5.2.2 with these additional criteria until all boundary points have been included in a partial umbrella or no more trusted neighbours remain in $Q_k(p)$. In some cases, one of the new umbrellas will not have a boundary; this occasionally happens in areas of high curvature and sparse sampling, where $t_G(p)$ might be misleading because the sampling assumptions do not hold. In these cases, we simply discard the complete umbrella when it contains the fewest vertices of all of p's umbrellas, and accept it (discarding the others) otherwise.

5.2.5 Enforcing boundary consistency

We have now constructed a local umbrella around each point p which is consistent with our characterization of the underlying surface (based on ω and γ) and incorporates our estimates of boundary and feature curves passing through p. However, aside from normal information in the cases of boundary points and untrustworthy Gabriel triangles, we have not incorporated any information contained in p's neighbours into its triangulation. For a well-sampled smooth surface this is generally sufficient; however, when the actual structure of our input point cloud does not satisfy our local uniformity assumptions this may lead to visual artifacts such as inconsistent or even spurious boundary detection where the input sampling breaks down.

Rather than attempting to enforce umbrella consistency across the whole point set as in the work of Kil and Amenta [50], we instead perform a third high-level pass over the input, identifying and correcting obvious inconsistencies near detected boundaries. This simple step significantly increases quality with minimal performance cost. Again, we are able to ignore most points in the input, instead focusing on points with boundaries that are incompatible with their neighbours' umbrellas. This may occur when a point's neighbour across a boundary edge does not itself have a boundary, or when two neighbours have boundary edges but do not agree on which edges those should be.

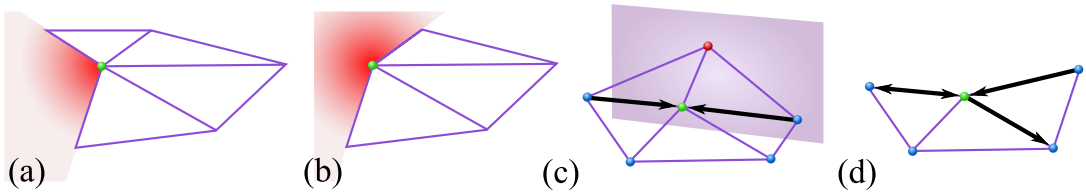


Figure 5.8: When a point p (green dot) has an inconsistent concave (a) or convex (b) boundary, we search $Q_k(p)$ for points that lie in the indicated area and contain reciprocal edges. Sparse sampling of boundaries may lead to point umbrellas with non-reciprocal edges on geodesically-distant sheets (c), or adjacent boundary vertices which do not agree with each other (d). Our method is able to address both problems by enforcing boundary edge reciprocity.

The first case may not indicate a problem at all; particularly in machine parts, when a fillet joins a smooth surface, a sharp edge will terminate rather than meet another feature curve. In this case, the sharp edge represents a crease in an otherwise smooth surface patch rather than a boundary between two distinct smooth patches. Thus, if p has multiple partial umbrellas which share all their boundary edges, we simply accept it.

Otherwise, we identify points in $Q_k(p)$ whose umbrellas contain p, and consider those edges when rebuilding p's umbrella. We call these *reciprocal edges*, and our general strategy in the third pass is to add missing reciprocal edges and remove spurious reciprocal edges when necessary to ensure consistency between adjacent boundary points.

We identify and address two cases: points with no boundary edges but at least two incoming boundary edges, and points with non-reciprocal boundary edges. The first case often occurs when a surface boundary approaches another sheet of the surface: points on the boundary may erroneously include points on the sheet in their umbrellas. In this case, we mark incoming boundary edges and search between them for onering neighbours without reciprocal edges. We remove these neighbours and construct a boundary consistent with the incoming boundary edges. See Figure 5.8(c).

The second case, where one or more of a point's open boundary edges connect to neighbours without reciprocal boundary edges, often occurs at a sparsely sampled boundary. Here we wish to correct the boundary, taking neighbour information into account. If the point has incoming boundary edges, we update its boundary to contain those edges, adding reciprocal edges as needed.

However, most of these points occur in regions of high curvature and low sample density,

where our φ assumption does not hold, and spurious and isolated boundaries are often detected. For these points, we find reciprocal edges in $Q_k(p)$ to fill the boundary, as shown in Figure 5.8(d).

5.3 Point set silhouette and feature extraction

We first describe our method for calculating the silhouette set of a point cloud and constructing local silhouette arcs. Then we present our preliminary attempt at point set feature extraction from the local umbrellas.

Silhouette extraction and silhouette arcs We slightly modify the Hough-space silhouette algorithm of Chapter 3 to handle the SGS-approximating umbrellas created in Section 5.2. Here we describe only the necessary modifications.

To take advantage of the spatially and temporally coherent nature of Hough-space silhouette extraction and update, we store the Hough transforms of each SGS face in an augmented octree as described in Chapter 3. Rather than consider every edge in p's umbrella, we store all faces associated with p together and test them as a group. This increases the complexity of testing the octree's edge bounding volumes against the v-sphere in initial silhouette extraction, but only by a constant factor.

Once we have identified a set of silhouette points, drawing silhouette *edges* between them is straightforward. Silhouette edges in each one-ring are easily identified but form a superset of the silhouette we wish to draw. Borrowing language from [50], we cull these edges into a more conservative set by drawing only *consensus silhouette edges*: We render a silhouette edge pq if and only if it exists and is a silhouette edge in the umbrellas of both p and q.

Note that we draw only local silhouette arcs, not full silhouette loops. The latter depend on global properties of point connectivity and must meet certain topological criteria [1] which we cannot guarantee from purely local constructions. It may be possible to augment our SGS construction with extra information and build consistent silhouette loops; we address this in our discussion of future work. For now we consider our silhouette arcs to be a first step towards a geometric solution.

Feature detection and emphasis To aid visualization of point clouds, we adapt the feature classification method of Hubeli and Gross [44] to our local reconstruction. While the original method is used to classify the feature strength of *edges*, we instead apply their ESOD operator to point samples. For each edge pq in p's umbrella, we use the umbrellas

of q's neighbours in p to evaluate $\cos\left(|\langle n_i, n_j\rangle|\right)^{-1}$, then divide by $\pi/2$. The absolute value term is required as our computed normals are not oriented, and thus are not guaranteed to be consistent between neighbouring umbrellas. To determine the weight of each point we simply take the maximum computed weight among its edges.

Rather than implement hysteresis and patch skeletonization, we find it sufficient for visualization simply to treat the normalized weight of each point as its alpha value during rendering. This conveys curvature information in a view-independent way without competing with the silhouette for emphasis; see Figures 5.1 and 5.10.

5.4 Results

Some results of our point set silhouette and feature extraction algorithm for quick point cloud visualization are shown in Figures 5.1, 5.2, and 5.10. In all the cases, we chose $Q_k(p)$ to be the 16 nearest neighbours, our sampling density parameter $\omega = \pi/6$, and our local uniformity parameters $\varphi = \pi - \omega$ and $\gamma = 2$. Small changes to these parameters tend to produce small changes in the results, and the values chosen here reflect the fact that all of our raw point cloud data were processed with WLOP [43]. Inputs with different sampling characteristics will require changes to the parameters that reflect those differences in sampling.

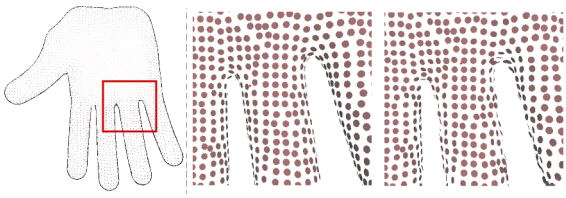


Figure 5.9: Point rendering using oriented splats on the hand mesh (left) to compare normal estimation: (centre) using our local reconstruction and (right) using PCA, where the same k for initial $Q_k()$ is used. The PCA-estimated normals between adjacent fingers are inconsistent with their neighbours, while ours are coherent.

Our SGS-building algorithm is a preprocessing step while silhouette and feature extractions are interactive. We performed our experiments on a workstation running Linux 2.6.18 with two Intel Xeon 3.2GHz processors, 4.0GB of RAM, and an NVidia GeForce 9800 GX2 card. The preprocessing step took between 5 seconds (hand model, 6,191 points) and 40 seconds (oil pump model, 54,220 points). Framerates for incremental silhouette updates varied with silhouette size, but never dropped below 190 frames per second.

Qualitatively, comparisons to the display of only visible points [48], as shown in Figure 5.1, reveal the ability of silhouettes and detected feature points to emphasize underlying

shapes especially near surface features and fine-scale details. It is also worth pointing out that as the point cloud becomes more sparse, pure point rendering (left of Figure 5.1) becomes less effective in revealing geometric details while the usefulness of the characteristic curves is increased.

While normal estimation is not a key contribution of our work, our method's ability to reconstruct correct umbrellas in the presence of close-by surface sheets is demonstrated by Figure 5.9. Here, we show the robustness of normal estimation using our local umbrella construction and patch filtering algorithm, in comparison to principal component analysis (PCA). While our umbrella construction starts with $Q_k()$, as in PCA, the optimization can identify the correct local neighborhood which does not straddle between the nearby sheets; this results in more accurate normal estimates.

In Figures 5.2 and 5.10, we compare to the use of normal thresholding for point set silhouette extraction, where features are included for a better depiction of the shapes. Focusing on just the silhouettes, it is quite evident that using our local reconstruction and the SGS-based extraction scheme effectively avoids both under- and over-detection of silhouettes, which occur simultaneously under normal thresholding even on models derived from triangle meshes. Specifically, in regions with low curvature (either due to dense sampling or the geometry of the underlying surface), we produce a set of silhouette points with far more consistent thickness than the thresholding method. This is particularly evident on the fandisk and oil-pump models. In regions with high curvature, we are able to identify silhouette points where thresholding fails, both on sharp edges such as on the fandisk and over smooth regions of high curvature such as the fingers of the hand. Even on the fertility model, which is best suited to normal thresholding, we are able to identify more silhouette points on the higher-curvature arms and avoid overselection on the flat base.

Finally, in Table 5.1 and Figure 5.11 we show the results of our boundary estimation on a number of datasets with boundaries. In order to evaluate our results we chose input data from triangle meshes with connectivity removed, and used the mesh boundaries as ground truth for the statistics in Table 5.1; while this may not be an ideal metric, it is at least an objective external standard. Each input surface contains one or more boundaries and includes sampling features that make boundary estimation nontrivial. Note that our method identifies more spurious boundary vertices than it fails to detect. Mesh connectivity is not restricted by vertex position or density, and in areas with extremely acute or obtuse triangles our method is likely to find small boundary loops.

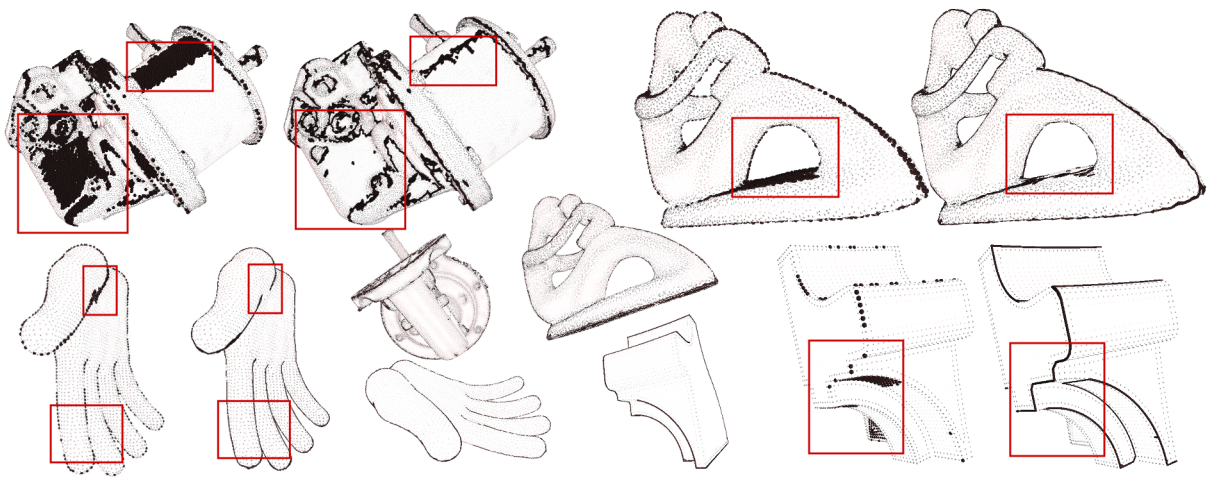


Figure 5.10: Comparison between normal thresholding (left figure of each pair) and our method (right). Insets show the models from the silhouette viewpoint. Red boxes highlight details discussed in the text.

We also compare our results to those of the excellent PEEL algorithm described in [21] by Dey *et al.*. This sophisticated method produces a provably-isotopic and globally consistent Delaunay mesh, even on non-orientable point clouds with boundaries. Note that only default parameters were used with PEEL. All three models generally satisfy PEEL's assumption of sufficiently uniform sampling, but its output mesh organization makes quantitative comparison difficult.

Model	Bdry	Vertices		Percentage	
		Missed	Extra	Missed	Extra
Saddle	256	0	0	0	0
Pig	544	16	38	2.9	6.9
Face-HY	338	13	23	3.8	6.8

Table 5.1: Quantitative comparison of our method to the base mesh used as ground truth. For each model in Figure 5.11, we show the number of boundary vertices on the mesh, and the number and percentage of missed and extra (spurious) boundary vertices.

The *saddle* model is a simple synthetic rendering of the neighbourhood of a saddle point. Of note is the fact that this model was computed as a single octant and stitched together with significant point overlap at octant boundaries; our method finds spurious features on some of these overlaps, though these are not marked as boundary vertices, while PEEL detects spurious potential boundaries. The inset shows our local umbrellas along an octant boundary near the centre of the model; note the abrupt changes in sample density.

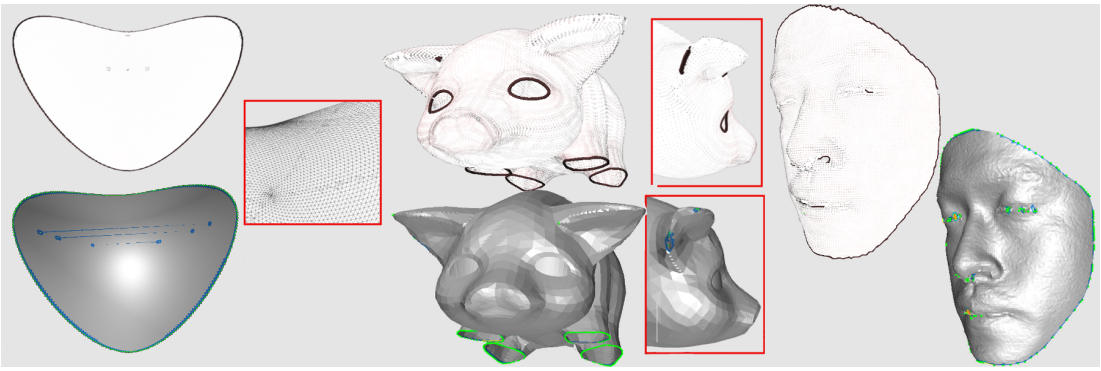


Figure 5.11: Comparison of our boundary detection results (white/brown) with those of PEEL (grey/green) on the *saddle*, *pig*, and *face-HY* models. Insets show features described in the text. Despite its local support, our method produces results comparable to PEEL.

Similarly, the *pig* model exhibits sharp and narrow linear features, particularly at the ears (shown in the inset). These features are by their nature undersampled, and neither our local reconstruction nor PEEL's more global method can perfectly reconstruct these regions. Also notable is that our method finds boundaries at the eyes, while PEEL triangulates them; this is due to our assumption that the local surface can be characterized by the Gabriel triangle radius.

Finally, the *face-HY* model is constructed from a single raw point scan, and exhibits typical scanner errors near the eyelashes, nostril, and lips. Both our method and PEEL misidentify internal points as boundaries where this occurs, but our ability to construct multiple partial umbrellas where sharp features occur allows us to produce fewer spurious boundaries at the eyes and mouth.

5.5 Conclusions

In this chapter, we generalize the SGSes described in Definition 2.1.5 to point clouds, use them to develop a purely local reconstruction algorithm, and demonstrate their utility for both silhouette extraction and simple feature detection. By exploiting successively more refined estimates of a point's SGS, we are able to cope with open boundaries, sharp feature edges, and close-by surfaces without the often-substantial processing required by global surface reconstruction techniques.

While the one-rings generated by our algorithm are not perfectly consistent with their neighbours, we are able to enforce a significant degree of consistency along boundaries, with

boundary-detection results comparable to those of more sophisticated techniques. In general we find a strong degree of consistency between nearby one-rings, and expect our SGS-based construction to be applicable to a wide variety of geometry processing algorithms beyond the feature-extraction algorithm described above.

Chapter 6

Conclusions

We have exploited the connection between object-space silhouettes and the tangent and supporting planes with which they interact to develop a number of novel tools for silhouette-based geometry processing.

6.1 Silhouette extraction using the 3D Hough transform

While transform-based silhouette extraction is not new (see Chapter 2.1.3), the efficiency of these methods depends heavily upon the distribution of points in transform space. In Chapter 3, we develop the *3D Hough transform* as an alternative to the plane-parameter transform of Hertzmann and Zorin [39] and the dual transform used by Pop *et al.* [69]. This transform exhibits a number of desirable properties when applied to the supporting planes of mesh faces; in particular, the distribution of points in transform space is far more uniform than that produced by the dual transform without the extra dimension of the plane-parameter transform. This allows us to produce data structures that are significantly more efficient, and easier to traverse at a low level, than those on point sets from other transforms.

Examination of the properties of 3D Hough space has uncovered efficient methods for from-scratch silhouette *extraction* from a given p_s and silhouette *update* with respect to spatially-coherent changes in p_s, features which had not been combined in previous works. Furthermore, we have discovered that the 3D Hough transform's advantages can be further exploited by careful selection of a small set of local origins, and in Section 4.4 have developed an algorithm for silhouette extraction based on these partitions. We note performance improvements of between two and three hundred percent using this method.

6.2 Silhouette aggregation for geometry analysis

In order to obtain the local origins used to achieve this performance improvement, we must consider a continuous domain of points in relation to every supporting plane on the mesh. We do this by constructing a *tangential distance field*, as shown in Chapter 4, which aggregates information from every plane at every point in our search space. Choosing an appropriate weight function, parameterized by the point's distance from the plane in question, lets us construct a voting scheme to select optimized origins, and weighting subsequent points by the weight of votes on earlier points produces a simple assignment of mesh faces to origins.

This method is suitable for a number of problems which consider global silhouette properties aggregated over the whole mesh, such as viewpoint selection and light source placement, when appropriate weight functions are selected. We show several examples based on the TDF which produce results comparable to the current state of the art, and present a number of primitives which can be combined to construct appropriate functions.

6.3 Local reconstruction of point clouds

By approaching the silhouette extraction problem in terms of supporting planes and containment of p_s rather than a separating curve, we develop an intuition for defining the silhouette of a point cloud based on Voronoi cells induced on the underlying surface. From this intuition we build a *local* reconstruction algorithm for point clouds with sufficient connectivity information to allow silhouette extraction and update. These local one-rings are rich enough to support other mesh-based algorithms on point clouds, without the complexity of a full reconstruction step.

While our local reconstruction is not intended to deal with noise in the sampling, it is able to detect and process sharp features in the data, which are by their nature chronically undersampled, by inferring the shape of smooth components from the plane of the Gabriel triangle. The same method allows us to detect boundaries in the input with accuracy roughly comparable to state-of-the-art global techniques such as Peel [21]. The local nature of our method's computations and small region of support lend themselves to a GPU-based implementation in the vein of [50].

6.4 Future work

A major theme of Chapters 4 and 5 is that, while geometric silhouettes are themselves useful for geometry processing as discussed in Chapter 2, extending the scope of silhouette-based geometry analysis with SGSes and TDFs results in a number of even more broadly applicable tools. We feel that the potential scope of this work far exceeds what has so far been achieved. Thus, while a number of limitations remain to be addressed in the work presented in this thesis, there is also significant opportunity to apply these methods to new geometric-computation problems.

In particular, the extension of Definition 2.1.5 in Chapter 5 underscores the flexibility of the silhouette-generating set. Armed with this concept, it is straightforward to define SGSes on polygons or regions in meshes, patches in displacement maps, voxels containing level-set surfaces, or any other primitive or assembly of primitives with a well-defined silhouette. We present some opportunities to exploit this flexibility below.

6.4.1 Hough-space silhouette extraction

The major flaw of precomputation-based silhouette extraction algorithms is their inability to cope with animated or deforming models. Silhouettes on these models must instead be found using brute-force or randomized algorithms, such as those in [12] and [59]. One possibility to bridge the gap between our methods and theirs is to follow the example of the real-time raytracing community and use a simpler data structure that can be quickly recomputed each frame; however, the recomputation cost of such a structure would have to be extremely low for the whole algorithm's performance to exceed the methods cited.

Another possible approach to the problem of dynamic meshes is to use the 3D Hough transform to guide a randomized algorithm. Since transformed points in Hough space make all silhouette changes spatially coherent, we may be able to address the tendency of randomized algorithms to miss new silhouette arcs by shifting random edge selection into a domain where new silhouettes are clustered with old ones. In order to make such an algorithm worthwhile, however, care would have to be taken to reduce the cost of each transformation and predicate well below the cost of a simple silhouette test. Clustering faces and operating on their SGSes may be the best way to achieve this cost reduction on average, if a sufficiently stable clustering scheme can be achieved. This latter problem is well-studied in the real-time raytracing community, as for example in Günther *et al.* [36].

Furthermore, when the silhouette of a model is used for geometry processing or rendering effects, multiresolution structures built around surface-based error metrics may prove to be suboptimal and silhouette extraction methods based on static connectivity are unlikely to function efficiently. A multiresolution silhouette extraction and update method combined with a specialized multiresolution mesh data structure would fill this gap, and the spatially-coherent nature of silhouettes in Hough space make it a natural tool for building such a combination. Recent work such as that of Hu *et al.* [42] shows that a significant amount of fine-grained geometry processing can be efficiently performed on the GPU; similar techniques may enable applications such as those mentioned above.

On a more theoretical level, none of the existing efficient silhouette algorithms has been proven to be output-sensitive – that is, to have time complexity that is linear in the size of the *output* set. These algorithms, including the one presented in Chapter 3, have been verified by experiment to grow more slowly than the input (see, for example, Figure 3.8), but a rigorous proof would contribute significantly to the understanding of silhouette complexity in general. The discrepancy between the theoretical bounds achieved in [32] and the empirical statistics obtained in [60] underlines the amount of work that remains to be done in analytic descriptions of silhouette complexity.

Finally, while this thesis emphasizes the geometric information available from the full silhouette, most rendering applications prefer only the *visible* silhouette shown in Figure 1.1(b), and many geometry processing algorithms can benefit from the occlusion information this structure contains. However, extracting the visible silhouette in object space is a difficult problem to solve efficiently, though the problem has been attacked in works such as [59]. Rather than address the problem purely geometrically, it may be possible to exploit the *combinatorial* structure of the complete silhouette to extract the visible rim more efficiently.

6.4.2 Tangential distance fields

As a conceptual tool, the tangential distance field is still in the early stages of its development. More sophisticated support distance functions – for example, those that weight the result by the size of the generating polygon, by the distance between that polygon's centroid and the point under consideration, or by the distance from the point to the bounding planes of the *polygon's* SGS – would no doubt improve the method's results in certain applications and allow the f_{TD}s to incorporate more precise information in their votes. This in turn may

permit a broader and more complete examination of f_{TD}s and their composition, which in turn may open more problems to TDF-based solutions.

More specifically, it is difficult to ignore the similarities between the TDF as a spatial function determined by a model's supporting planes and the aspect graph or visibility complex as a spatial *partition* determined by a model's supporting planes. While the visibility complex presents difficulties due to the inherent discontinuities in the visibility function, we take special care to ensure that the TDF is a sum of *smooth* functions. It may therefore be possible to construct a TDF that assists in the construction of the visibility complex.

Since we generate a small number of features – the points selected by the voting scheme – from aggregate mesh geometry using what can be seen as a low-pass filter, we expect our method to be robust to changes in connectivity and, to a certain extent, even geometry. We may therefore be able to use our framework to produce a compact mesh descriptor for rigid models. Understanding the insensitivity of our f_{TD}s to high-frequency detail also helps explain their stability as the source model is simplified, as shown in Figure 4.13, and may lead to performance enhancements later on as well as insights into the multiresolution silhouette structure described above.

6.4.3 Local point-cloud reconstruction

The local reconstruction technique we develop in Chapter 5 will naturally be compared to fully globally-consistent algorithms. Therefore, the obvious next step in this work is to compare the performance, in terms of both quality and efficiency, of our method to the state of the art in surface reconstruction. This should be done on a full range of filtered, noisy, and incomplete data sets, with an emphasis on determining the parameters and applications where one method is superior to another. Part of this comparison will include the implementation of our method on the GPU, for which it is particularly well-adapted. The work of Kil and Amenta in [50] demonstrates the potential of a locally-operating reconstruction algorithm on graphics hardware.

Despite their purely local construction, our umbrellas are surprisingly consistent with their neighbours. This leads us to two observations: first, that a more globally-consistent reconstruction algorithm may only be a short step away; and second, that the presence of inconsistency between neighbours may signal interesting properties of the underlying data. Both of these indicate future courses of research which merit further consideration.

Finally, the successful implementation of Hubeli and Gross's feature detection algorithm from [44] on our independent one-rings demands a more complete investigation of the application of mesh algorithms to our reconstruction. Algorithms with small regions of support, such as the calculation of graph Laplacians, seem likely to succeed, while most subdivision methods are unlikely to prove easy to adapt and the suitability of path-based algorithms like geodesic-distance approximations is unclear. Nonetheless, further research into mesh algorithms will broaden the impact of the local reconstruction and suggest avenues for its improvement.

Bibliography

[1] Tomas Akenine-Möller and Ulf Assarsson. On the degree of vertices in a shadow volume silhouette. *Journal of Graphics Tools*, 8(4):21–24, 2003.

[2] H. Alt, M. Glisse, and X. Goaoc. On the worst-case complexity of the silhouette of a polytope. In *15th Canadian Conference on Computational Geometry-CCCG*, volume 3, pages 51–55, 2003.

[3] C. Andujar, P. Vazquez, and M. Fairen. Way-finder: Guided tours through complex walkthrough models. *Computer Graphics Forum*, 23(3):499–508, 2004.

[4] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. Exponential shadow maps. In *Proceedings of Graphics Interface 2008*, pages 155–161, 2008.

[5] Daniel Archambault, William Evans, and David Kirkpatrick. Computing the set of all the distant horizons of a terrain. *International Journal of Computational Geometry and Applications*, 15(6):547–563, 2005.

[6] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics (TOG)*, 22(3):511–520, 2003.

[7] Ulf Assarsson and Tomas Akenine-Möller. Occlusion culling and z-fail for soft shadow volume algorithms. *The Visual Computer*, 20(8-9):601–612, 2004.

[8] F. Benichou and G. Elber. Output sensitive extraction of silhouettes from polygonal geometry. In *Proc. of Pacific Graphics*, pages 60–69, 1999.

[9] J. Bittner and P. Wonka. Visibility in computer graphics. *Journal of Environment and Planning B: Planning and Design*, 5(30):729–756, 2003.

[10] S. Brabec and H. Seidel. Shadow volumes on programmable graphics hardware. *Computer Graphics Forum*, 22(3):433–440, 2003.

[11] John Brosz, Faramarz Samavati, and Mario Costa Sousa. Silhouette rendering based on stability measurement. In *SCCG '04: Proceedings of the 20th Spring Conference on Computer Graphics*, pages 157–167, 2004.

[12] J. Buchanan and M. Sousa. The edge buffer: A data structure for easy silhouette rendering. In *Proc. of the 1st Int. Symp. on Non-Photorealistic Animation and Rendering (NPAR)*, pages 39–42, 2000.

[13] M. Burns, J. Klawe, S. Rusinkiewicz, A. Finkelstein, and D. DeCarlo. Line drawings from volume data. *ACM Trans. Graph.*, 24(3):512–518, 2005.

[14] D. Card and J. Mitchell. Non-photorealistic rendering with pixel and vertex shaders. In *ShaderX: Vertex and Pixel Shader Tips and Tricks*, pages 319–333, 2002.

[15] Norman Chin and Steven Feiner. Near real-time shadow generation using BSP trees. *ACM Trans. Graph.*, 23(3):99–106, 1989.

[16] M. Christie and P. Olivier. Camera control in computer graphics. In *Eurographics 2006 STARs*, pages 89–113, 2006.

[17] C. Cowan and B. Modayur. Edge-based placement of camera and light source for object recognition and location. In *IEEE International Conference on Robotics and Automation*, pages 586–591, 1993.

[18] F. Crow. Shadow algorithms for computer graphics. In *ACM SIGGRAPH*, pages 242–248, 1977.

[19] X. Décoret, F. Durand, F. Sillion, and J. Dorsey. Billboard clouds for extreme model simplification. *ACM Trans. Graph.*, 22(3):689–696, 2003.

[20] O. Deussen and T. Strothotte. Computer-generated pen-and-ink illustration of trees. In *Proc. SIGGRAPH 2000*, pages 13–18, 2000.

[21] T. K. Dey, K. Li., E. A. Ramos, , and R. Wenger. Isotopic reconstruction of surfaces with boundaries. In *ACM Symposium on Geometry Processing*, pages 1371–1382, 2009.

[22] Tamal Dey. *Curve and Surface Reconstruction; Algorithms with Mathematical Analysis.* Cambridge University Press, 2007.

[23] F. Durand, G. Drettakis, and C. Puech. The 3D visibility complex: a new approach to the problem of accurate visibility. *ACM Trans. Graph.*, 21(2):176–206, 2002.

[24] R. Dyer. *Self-Delaunay meshes for surfaces.* PhD thesis, Simon Fraser University, Burnaby, Canada, 2010.

[25] R. Dyer, H. Zhang, and T. Möller. Delaunay mesh construction. In *ACM Symposium on Geometry Processing*, pages 271–282, 2007.

[26] Alon Efrat, Leonidas Guibas, Olaf Hall-Holt, and Li Zhang. On incremental rendering of silhouette maps of a polyhedral scene. In *Proc. Sym. on Discrete Algorithms*, pages 910–917, 2000.

[27] G. Elber and E. Cohen. Hidden curve removal for free form surfaces. *ACM Trans. Graph.*, 24(4):95–104, 1990.

[28] G. Elber and E. Cohen. Arbitrarily precise computation of Gauss maps and visibility sets for freeform surfaces. In *SMA '95: Proceedings of the third ACM Symposium on Solid Modeling and Applications*, pages 271–279, 1995.

[29] B. S. Everitt, S. Landau, and M. Leese. *Cluster Analysis*. Hodder Arnold Publishing and Oxford University Press, 2001.

[30] H. Fu, D. Cohen-Or, G. Dror, and A. Sheffer. Upright orientation of man-made objects. *ACM SIGGRAPH*, 27(3):42:1–42:7, 2008.

[31] Z. Gigus and J. Malik. Computing the aspect graph for line drawings of polyhedral objects. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12(2):113–122, 1990.

[32] M. Glisse and S. Lazard. An upper bound on the average size of silhouettes. *Discrete Comput. Geom.*, 40(2):241–257, 2008.

[33] B. Gooch, P. J. Sloan, A. Gooch, P. Shirley, and R. Risenfeld. Interactive technical illustration. In *Proc. ACM Symp. on 3D Interactive Graphics*, pages 31–38, 1999.

[34] A. Greß, M. Guthe, and R. Klein. GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum*, 25(3):497–506, 2006.

[35] S. Gumhold. Maximum entropy light source placement. In *IEEE Visualization*, pages 275–282, 2002.

[36] J. Günther, H. Friedrich, I. Wald, H. Seidel, and P. Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3):517–525, September 2006. (Proceedings of Eurographics).

[37] T. Heidmann. Real shadows, real time. *IRIS Universe*, 18:23–31, 1991.

[38] A. Hertzmann. Introduction to 3D non-photorealistic rendering: Silhouettes and outlines, 1999. SIGGRAPH 1999 course notes.

[39] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *In ACM SIGGRAPH*, pages 517–526, 2000.

[40] S. Hornus, J. Hoberock, S. Lefebvre, and J. Hart. ZP+: correct Z-pass stencil shadows. In *ACM Symposium on Interactive 3D Graphics and Games*, pages 195–202, 2005.

[41] P. V. C. Hough. Machine analysis of bubble chamber pictures. In *Int. Conf. on High Energy Accelerators and Instrumentation*, 1959.

[42] L. Hu, P.V. Sander, and H. Hoppe. Parallel view-dependent level-of-detail control. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):718–728, Sept.-Oct. 2010.

[43] H. Huang, D. Li, H. Zhang, U. Ascher, and D. Cohen-Or. Consolidation of unorganized point clouds for surface reconstruction. *ACM Transactions on Graphics, (Proceedings SIGGRAPH Asia 2009)*, 28(5):176:1–176:7, 2009.

[44] Andreas Hubeli and Markus Gross. Multiresolution feature extraction for unstructured meshes. In *Proc. Visualization 2001*, pages 287–294, 2001.

[45] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte. A developer's guide to silhouette algorithms for polygonal models. *IEEE Comput. Graph. Appl.*, 23(4):28–37, 2003.

[46] M. Jones and J. P. Oakley. Registration of image sets using silhouette consistency. In *IEEE Proc. Vis. Image Signal Process.*, volume 147, pages 1–8, 2000.

[47] E. Kalogerakis, D. Nowrouzezaharai, P. Simari, J. McCrae, A. Hertzmann, and K. Singh. Data-driven curvature for real-time line drawing of dynamic scenes. *ACM Trans. Graph.*, 28(1):11:1–11:13, 2009.

[48] S. Katz, A. Tal, and R. Basri. Direct visibility of point sets. In *International Conference on Computer Graphics and Interactive Techniques*, volume 26, 2007.

[49] L. Kettner and E. Welzl. Contour edge analysis for polygonal projections. In W. Strasser, R. Klein, and R. Rau, editors, *Geometric Modeling: Theory and Practice*, pages 379–394, 1997.

[50] Y. J. Kil and N. Amenta. GPU-assisted surface reconstruction on locally-uniform samples. In *Proc. Int. Meshing Roundtable*, pages 369–385, 2008.

[51] D. Kirsanov, P. V. Sander, and S. J. Gortler. Simple silhouettes for complex surfaces. In *Computer Graphics Forum (Special Issue of Symp. Geometry Processing)*, pages 102–106, 2003.

[52] Jan Koenderink. What does the occluding contour tell us about solid shape? *Perception*, 13:321–330, 1984.

[53] S. Laine. A general algorithm for output-sensitive visibility preprocessing. In *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*, pages 31–39, 2005.

[54] S. Laine, T. Aila, U. Assarsson, J. Lehtinen, and T. Akenine-Möller. Soft shadow volumes for ray tracing. *ACM Trans. Graph.*, 24(3), 2005.

[55] Svetlana Lazebnik, Yasutaka Furukawa, and Jean Ponce. Projective visual hulls. *International Journal of Computer Vision*, 74(2):137–165, 2007.

[56] Svetlana Lazebnik and Jean Ponce. The local projective shape of smooth surfaces and their outlines. *Int. J. Comput. Vision*, 63:65–83, June 2005.

[57] J. Lehtinen, S. Laine, and T. Aila. An improved physically-based soft shadow volume algorithm. *Computer Graphics Forum*, 25(3):303–312, 2006.

[58] Brandon Lloyd and Parris Egbert. Horizon occlusion culling for real-time rendering of hierarchical terrains. In *IEEE Visualization*, pages 403–409, 2002.

[59] L. Markosian, M. A. Kowalski, D. Goldstein, S. J. Trychin, J. F. Hughes, and L. D. Bourdev. Real-time nonphotorealistic rendering. In *ACM SIGGRAPH*, pages 415–420, 1997.

[60] Morgan McGuire. Observations on silhouette sizes. *Journal of Graphics Tools*, 9(1):1–12, 2004.

[61] B. Mederos, L. Velho, and LH De Figueiredo. Moving least squares multiresolution surface approximation. In *Computer Graphics and Image Processing, 2003. SIBGRAPI 2003. XVI Brazilian Symposium on*, pages 19–26, 2003.

[62] Microsoft inc. DirectX 11 SDK. Accessed 2011 May 4.

[63] M. Olson, R. Dyer, A. Sheffer, and H. Zhang. Point set silhouettes via local reconstruction. In *Proc. Shape Modeling International*, 2011. Conditionally accepted.

[64] M. Olson and H. Zhang. Silhouette extraction in Hough space. *Computer Graphics Forum (Special Issue on Eurographics 2006)*, 25(3):273–282, 2006.

[65] "Daf-de" on Wikimedia Commons. Hough transform example image, 2006. Published under the GNU Free Documentation License, accessed 2011 May 4.

[66] R. Plaenkers and P. Fua. Model-based silhouette extraction for accurate people tracking. In *European Conf. on Computer Vision*, pages 325–339, 2002.

[67] J. Podolak, P. Shilane, A. Golovinskiy, S. Rusinkiewicz, and T. Funkhouser. A planar-reflective symmetry transform for 3D shapes. In *ACM SIGGRAPH*, pages 549–559, 2006.

[68] O. Polonsky, G. Patane, S. Biasotti, C. Gotsman, and M. Spagnuolo. What's in an image? Towards the computation of the "best" view of an object. *The Visual Computer*, 21(8):840–847, 2005.

[69] M. Pop, C. Duncan, G. Barequet, M. Goodrich, W. Huang, and S. Kumar. Efficient perspective-accurate silhouette computation and applications. In *Proc. of Annual Symp. on Computational Geometry*, pages 60–68, 2001.

[70] J. Princen, J. Illingworth, and J. Kittler. A formal definition of the hough transform: Properties and relationships. *Journal of Mathematical Imaging and Vision*, 1(2):153–168, 1992.

[71] R. Raskar and M. Cohen. Image precision silhouette edges. In *Proc. of the Symp. on Interactive 3D Graphics*, pages 135–140, 1999.

[72] A. Rivers, F. Durand, and T. Igarashi. 3D modeling with silhouettes. *ACM Trans. Graph.*, 29(4):1–8, 2010.

[73] T. Saito and T. Takahashi. Comprehensible rendering of 3D shapes. *ACM Trans. Graph.*, 24(3):197–206, 1990.

[74] P. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. Graph.*, 27(5):1–9, 2008.

[75] P. V. Sander, X. Gu, S. J. Gortler, H. Hoppe, and J. Snyder. Silhouette clipping. In *ACM SIGGRAPH*, pages 327–334, 2000.

[76] A. Sethi, D. Renaudie, D. Kriegman, and J. Ponce. Curve and surface duals and the recognition of curved 3D objects from their silhouettes. *Int. J. Comput. Vision*, 58:73–86, June 2004.

[77] F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum*, 16(3):207–218, 1997.

[78] B. Su and D. Liu. *Computational Geometry: Curve and Surface Modeling*. Academic Press, 1989.

[79] B. Triggs and C. Laugier. Automatic camera placement for robot vision tasks. In *IEEE International Conference on Robotics and Automation*, pages 1732–1738, 1995.

[80] R. Tsai, L. Cheng, P. Burchard, S. Osher, and G. Sapiro. Visibility and its dynamics in a PDE based implicit framework. *J. Comput. Phys.*, 199(1):260–290, 2004.

[81] P. Vazquez. Automatic light source placement for maximum visual information recovery. *Computer Graphics Forum*, 26(2):143–156, 2006.

[82] P. Vazquez, M. Feixas, M. Sbert, and W. Heidrich. Viewpoint selection using viewpoint entropy. In *Proc. of the Vision, Modeling, and Visualization (VMV) Conference*, pages 273–280, 2001.

[83] D. Vlasic, I. Baran, W. Matusik, and J. Popović. Articulated mesh animation from multi-view silhouettes. *ACM Trans. Graph.*, 27(3):1–9, 2008.

[84] Lance Williams. Casting curved shadows on curved surfaces. *ACM Trans. Graph.*, 12(3):270–274, 1978.

[85] Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov. Guided visibility sampling. *ACM Trans. Graph.*, 25(3):494–502, 2006.

[86] Jianhua Wu and Leif Kobbelt. Piecewise linear approximation of signed distance fields. In *Proc. Vision, Modeling, and Visualization*, pages 513–520, 2003.

[87] Hui Xu, Minh Nguyen, Xiaoru Yuan, and Baoquan Chen. Interactive silhouette rendering for point-based models. In *Eurographics Symposium on Point-Based Graphics*, pages 13–18, 2004.

[88] H. Yamauchi, W. Saleem, S. Yoshizawa, Z. Karni, A. Belayev, and H-P. Seidel. Towards stable and salient multi-view representation of 3D shapes. In *Proc. of Shape Modeling and Applications (SMI)*, pages 40–46, 2006.

[89] T. Zaharia and F. Preteux. Hough transform-based 3d mesh retrieval. In *Proc. of the SPIE Conf. 4476 on Vision Geometry X*, pages 175–185, 2001.

[90] N. Zakaria and H. Seidel. Interactive stylized silhouette for point-sampled geometry. In *Computer Graphics and Interactive Techniques in Australasia and South East Asia*, pages 242–249. Association for Computing Machinery, 2004.