Visibility Layer Decomposition

A Thesis presented

by

David Jong Ryu

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 2, 2001

Abstract

Many algorithms exist that compute visibility for sets of polygons; however, object-based visibility resolution on general manifold surfaces has not been widely studied.

Given a manifold in R^3 with certain regularity properties, we present a visibility determination algorithm based on the concept of the *visibility layer decomposition*. The visibility layer decomposition (VLD) is a partitioning of the visible portions of the manifold into regions that do not self-occlude nor form occlusion cycles. With such a partitioning, visibility determination can be done with a simple topological sort based on pairwise occlusion. This algorithm is general in the sense that it can be used on any surface representation for which the basic operations of the algorithm have been implemented. Thus, object-based occlusion resolution can be leveraged onto a surface of any type.

Our algorithm is based on the idea of the *occlusion set*, a novel characterization of coherent areas of occlusion between surfaces. We compute the VLD by "cutting" the manifold to remove occlusion until the manifold is partitioned into visibility-independent regions. Cuts are determined entirely by analysis of silhouette curves in projection.

We also present our adaptation of this algorithm for triangle meshes. Unlike previous methods, our algorithm applied to triangle meshes treats a mesh as a manifold instead of a set of unrelated faces, and thus is able to take advantage of the inherent structure present in well-behaved manifold surfaces. Because our algorithm only considers the properties of the silhouette in projection, we have to touch only a small number of the faces in the mesh.

Contents

Chapter 1

Introduction

Visibility computation is an essential part of nearly all computer graphics rendering. To create an image, any rendering algorithm must somehow determine which surfaces are visible from the viewpoint and which are obscured; methods of performing this task are called visibility determination algorithms.

1.1 Z-Buffer

The method most commonly used today is called *Z-buffer*, so named because the post-transform distance away from the image plane to any point in space is usually denoted by Z. Invented by Catmull [1], the Z-buffer algorithm is designed for use in raster displays; that is, any display where the image is composed of an array of pixels. In the Z-buffer algorithm, two buffers are maintained: the image buffer and the Z-buffer. Both of these are the size of the raster display. The image buffer stores the color of the closest object thusfar drawn at every point and the Z-buffer stores the Z value of that object at that point. Before the renderer writes a pixel to the image buffer, it first checks the Z-buffer to see if a closer pixel has already been drawn, by comparing the current object's Z with the one in the Z-buffer. If the current object is closer, it updates both the image buffer with the new color, and the Z-buffer with the new Z. In this way, geometric objects can be specified in any order, and the closest one at every pixel in the raster display will always be drawn in the final image.

Z-buffer fits very neatly into today's rendering architectures; fast memory for the Z-buffer is cheap, and the algorithm is easily hardware accelerated. For these reasons, almost all graphics hardware in use today uses Z-buffer to resolve visibility.

1.1.1 Drawbacks of Z-buffer

Despite their near-ubiquitous deployment, Z-buffer and Z-buffer-based algorithms suffer from certain drawbacks. While Z-buffer resolves visibility at a per-pixel level, sometimes what is required is an actual geometric description of what is visible and what is not in the scene. Vector language based rendering (such as Macromedia Flash) is an example of such an application. In these rendering systems, the basic rendering unit is not the 3D triangle nor even the pixel; the scene is composed of a set of geometry elements in the plane. As such, Z-buffer cannot be applied because the raster display is either non-existent or inaccessible. Algorithms that resolve visibility apart from the resolution of the raster display are called *object precision* algorithms.

In addition, Z-buffer fails to give a full description of visibility even on a per-pixel level. Instead, the only information we have is which object is the closest, not neccessarily what objects occlude what others. For certain applications, such as alpha compositing, this does not suffice. In alpha compositing, an object's color is determined not only by the color of the object but also by the color of the objects behind it (alpha blending is commonly used for non-refractive transparency). In order for alpha blending to be computed correctly, the objects need to be specified in back-to-front order; one can no longer depend on Z-buffer to determine depth per-pixel.

1.2 Sorting-based Algorithms

Z-buffer reduces visibility of a scene to visibility per-pixel. Another method of visibility computation is to first compute what objects are in front of what others, and then draw them in back-to-front order, simply overwriting what is in the image buffer with the current object. This is called the *painter's algorithm*, based on observation that painters traditionally deal with visibility by painting further objects first, and closer objects later.

Essential in this algorithm is how to determine the back-to-front ordering. One must note that a simple depth sort of the faces will not generally provide the correct back-to-front order. Consider the following situation:

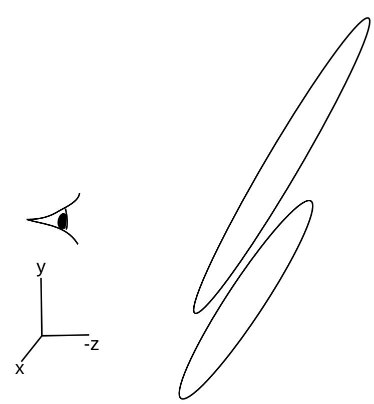


Figure 1.1: No sort based on a single Z will be correct.

In this situation, no method of sorting based on a single Z (by closest Z, furthest Z, or median Z, for instance) will sort the objects so that the visibility is correct when rendered back-to-front. In addition, sometimes no ordering is possible that assures correct back-to-front visibility:

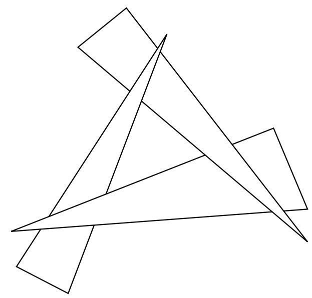


Figure 1.2: These objects admit no ordering.

Sorting-based techniques for visibility resolution require ways of dealing with these situations to yield an ordering that, when rendered in that order, yields correct visibility. Such an ordering we call a *visibility ordering*.

Snyder and Lengyel[8] introduced the *occlusion graph*, the directed graph where the relation $A \to B$ in the graph means that object A occludes object B from the viewpoint. A topological sort of such a graph, by definition, is an ordering where if A preceeds B, A must occlude B. Thus, a last-to-first ordering of such a sort will be a back-to-front ordering of the objects in the scene.

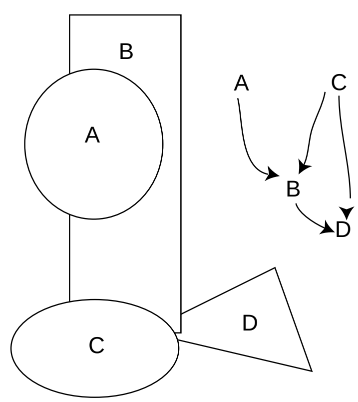


Figure 1.3: Some objects and their associated occlusion graph.

Simple graph theory tells us that if there is a cycle in the occlusion graph, then no topological sort exists. Such a situation is called an *occlusion cycle*. Some algorithms deal with this by splitting objects so that a link in the cycle is broken, thereby removing the cycle.

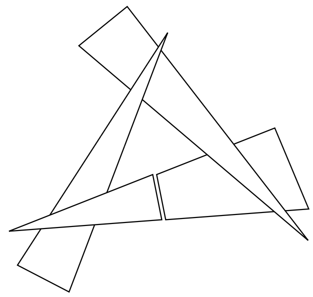


Figure 1.4: Splitting the bottom object removes the cycle.

Newell, Newell and Sancha[5] describe an algorithm for computing a visibility ordering on a set of polygons. This algorithm (NNS) begins by first sorting the triangles based on furthest Z. We have already demonstrated that such a sort is not a proper visibility ordering. To extract a visibility ordering from the depth sort, NNS then goes on to resolve occlusion ambiguities between triangles whose Z extents overlap. This ambiguity resolution consists of a series of tests of increasing granularity; first, they test overlap in X and Y, then try to find splitting planes, then check explicit overlap in projection. If they find that these polygons cannot be resolved (as with an occlusion cycle), then they split the polygon to break the occlusion.

1.3 Layer-based Rendering

Lengyel and Snyder[4] describe a rendering system based on layer compositing. They first factor a geometric scene description into a set of *parts*, based on geometric, image-based, and temporal coherence, so that each individual part more or less behaves as a unit through the course of an animation. Each part is rendered as a 2D image into a layer. Output images are created by compositing the layers in back-to-front order. Layer factorization means that changes to individual parts of the scene do not require a full rerender of all the geometry, and many layer updates can be expressed as a 2D warp of the image. In this way, their rendering pipeline takes advantage of temporal coherence present in most animations.

To compute the back-to-front ordering of the layers, they use an algorithm based on NNS that takes further advantage of coherence in visibility relations. This algorithm, described in Snyder and Lengyel[8], is similar to NNS in that they take an initial "guess" at the visibility ordering and refine it through a series of occlusion tests of the constituent objects. Unlike NNS, their algorithm operates on layers, and not on polygons; in addition, their method is incremental in the sense that is uses the results from the previous invocation of the algorithm to drive the current visibility determination. Furthermore, their algorithm combines layers instead of splitting them; to resolve occlusion cycles and other cases where NNS would dictate splitting a polygon, they combine the offending layers into a single layer through a compositing expression that correctly captures the intra-layer occlusion relationships.

Layer-based asynchronous rendering has many advantages over the traditional rendering pipeline. In addition to naturally taking advantage of coherence in scenes, these approaches permit many techniques not available in the traditional, Z-buffer based pipeline. These include rendering with transparency, special effects such as depth-of-field and motion blur, and fast hidden line rendering.

Current layer-based rendering systems, however, cannot correctly deal with objects that exhibit complex self-occlusion behaviors, and objects that form occlusion cycles. In these cases, they must combine layers so that the full benefits of layer decomposition are not available for such objects. Our research will show how to decompose arbitrary objects or sets of objects into layers that can be used in such a layer-based system, bringing the benefits of layer-based rendering to arbitrary objects without resorting to layer combination.

1.4 Cutting

Krishnan and Manocha[3] presented an algorithm that resolves visibility for smooth polynomial (NURBS) surfaces. First, a NURBS surface is decomposed into a set of trimmed Bezier patches. Each patch is then partitioned into a set of surfaces by the patch's *visibility curves*, the curves consisting of the projection of the silhouette curves onto the patch from the viewpoint. The surface is "cut" along the visibility curves into a set of smaller patches. This partitioning is guaranteed to yield a set of surfaces which are completely visible or invisible. Thus, they can be visibility-ordered.

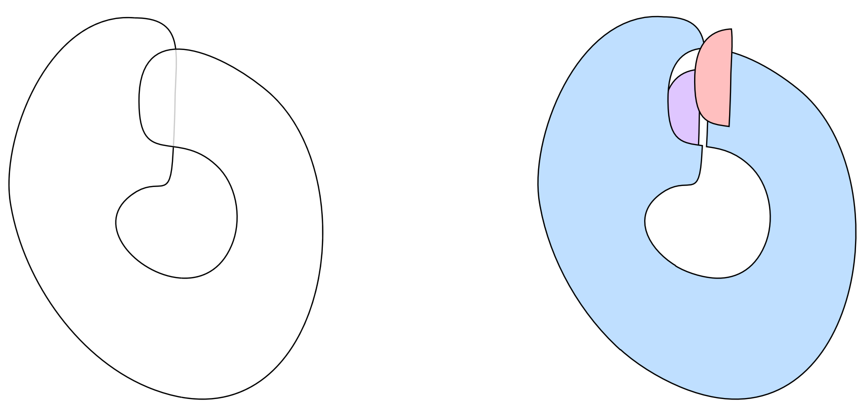


Figure 1.5: An object and the partition induced by the visibility curves.

1.5 Our Approach

Most of the previous approaches to visibility rely on an *a priori* partitioning of the object(s) to simplify occlusion testing. Z-buffer tests visibility per-pixel; NNS assumes that the object must be polygonized, and the occlusion resolution occurs per-polygon. While Krishnan and Manocha analyzed visibility for Bezier patches, to actually resolve global visibility they first must partition a NURBS surface into Bezier surfaces. None of these partitionings are fundamental to the task of resolving occlusion. While they might be helpful for completing a particular visibility task with a certain type of surface, they exist only to simplify the task and are not necessary for the task itself.

Moreover, the dependency on these pre-partitionings does not admit a general, object-based approach to visibility that treats arbitrary surfaces, because the visibility schemes deduced by these algorithms is dependent on one type of surface or another. Thus, no general object-based approach to visibility currently exists. Our analysis seeks to analyze visibility while assuming only basic properties of continuity and non-self-intersection. Because of this, our algorithmic framework is applicable to any such surface, regardless of representation. Thus, with our framework, visibility algorithms for any surface can be created, if the basic tools are implemented for those types of surfaces.

We also wish to analyze exactly how a partition on the surface changes the visibility properties of the surfaces generated. We do this for two reasons: firstly, to better understand how visibility is resolved through partitioning, and secondly, in the hope that more efficient algorithms can be designed from this analysis. From our conclusions, we will then present an adaptation of the algorithm implemented for triangle meshes that partitions an mesh (or set of meshes) into surfaces that can be visibility-ordered.

Chapter 2

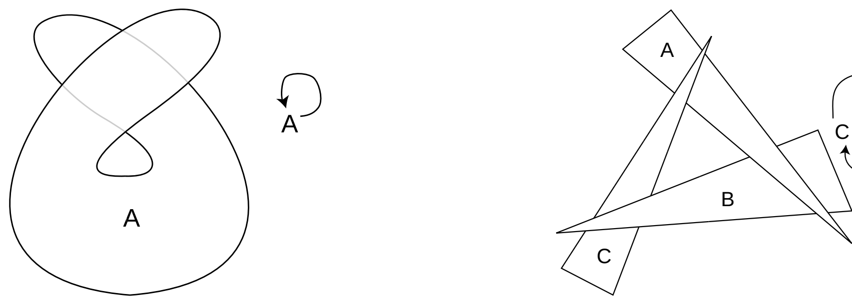
Occlusion on Arbitrary Surfaces

Our approach to occlusion is based on a novel characterization of coherent regions on occluding surfaces. In this chapter, we will explore the behavior of occlusion on non-self-intersecting manifold surfaces. We will provide a method for characterizing coherent regions of occlusion in a structure we call the *occlusion set*. We will then show how this characterization can be used to resolve occlusion through partitioning of the surface into subsurfaces. These insights will be put together to give a general algorithm for resolving occlusion on manifold surfaces.

2.1 Introduction

Our goal is to decompose an arbitrary set of front-facing surfaces S into a set of subsurfaces S' so that S' can be visibility-ordered. A set of surfaces cannot be visibility-ordered if the occlusion graph contains cycles (a visibility ordering is a topological sort of the occlusion graph). Thus, our task is equivalent to removing cycles from the occlusion graph.

We distinguish between two different types of cycles: self-occlusion cycles, and multi-surface cycles. A self-occlusion cycle is a cycle of the form $s \rightarrow s$ in the occlusion graph. A multi-surface cycle contains more than one surface. Our analysis will focus on how to identify and remove these "problem" features of the graph.



(a) Self-occlusion (b) Multi-surface occlusion

Figure 2.1: Problem features of occlusion graphs.

2.2 Surfaces

We define a surface as a continuous, orientable, non-self-intersecting manifold in R^3, with or without boundary. We will discuss certain properties of such surfaces to aid our discussion of occlusion resolution.

2.2.1 Surface Orientation

A point p on a surface is called *front-facing* with respect to a viewpoint v if:

$$(p - v) \cdot normal_p < 0$$

In the case that:

$$(p - v) \cdot normal_p > 0$$

the point is *back-facing* (note that the normal vector to a closed manifold is chosen here to point out of the region of space enclosed by the manifold).

For closed surfaces, back-facing regions are always invisible from the viewpoint. Because of this, back-facing regions of surfaces are often discarded in the rendering process; from now on, we will only consider the front-facing surfaces of an object for the purposes of occlusion resolution. Note that all of the following analysis (and subsequent algorithm) can also operate on the back-facing regions if all orientations are flipped. Thus, a full visibility decomposition of an object can be obtained by two applications of our algorithm.

2.2.2 Silhouette

The silhouette of a smooth surface s is the set of points p with the following property:

$$(p - v) \cdot normal_p = 0$$

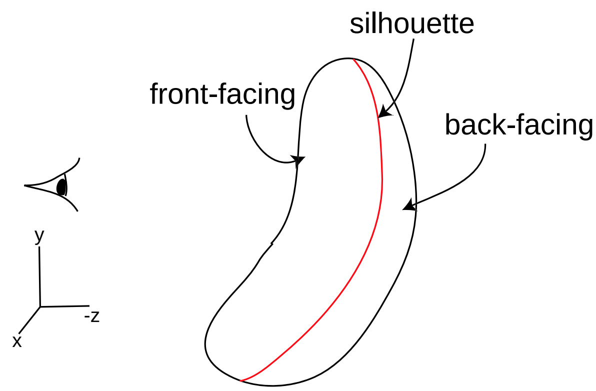


Figure 2.2: The silhouette and front and back-facing regions of an object.

This set of points form closed curves on the surface that separate front and back facing regions. Because we are only considering front-facing regions, the silhouettes of an object form the initial partition of the object into surfaces for our algorithm.

2.2.3 Partitioning

Our method of occlusion resolution uses surface partitioning to remove features in the occlusion graph (such as occlusion cycles and self-occlusion) that prohibit visibility ordering. A partition of a surface s is a set of surfaces S such that $\bigcup S = s$, and $\bigcap S$ is empty. We will perform these partitions by introducing new boundary curves onto surfaces, like so:

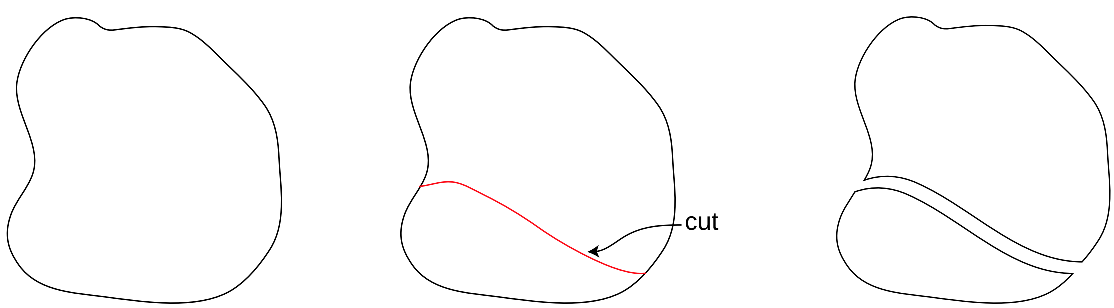


Figure 2.3: An object, a cut (in red) on the object, and the partition induced by the cut.

The process of adding a boundary to a surface along a curve is called *cutting* the surface. The curve along which the boundary is added is called the *cut*. Note that our boundaries are all oriented so that the surface they bound lies locally to the left. This gives an implicit orientation to our boundaries. Some of the cuts that we introduce do not actually divide a surface into sub-surfaces; they merely add boundaries to the surface:

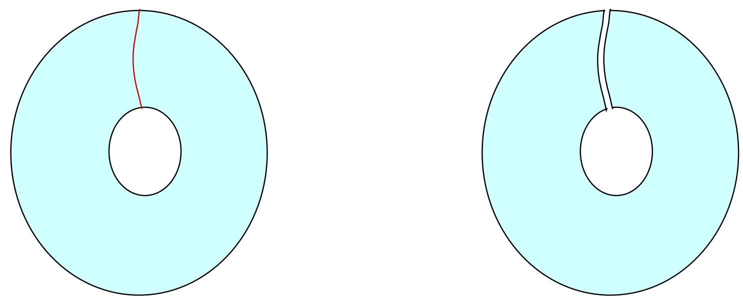


Figure 2.4: Some cuts do not partition; they merely add boundaries.

Note that we will consider points across partitioned boundaries as not connected. That is, we imagine an infinitesimal gap at boundaries of a partition:

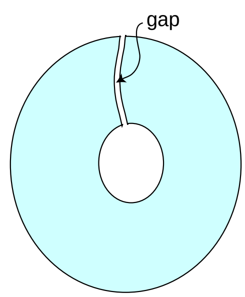


Figure 2.5: Although geometrically the figure spans across the cut, we imagine there is a gap between the two boundaries created by the cut.

In addition, each point on a surface can be a member of at most one boundary. Thus, boundaries that meet at points on the surface are considered as being separated by some infinitesimal distance, and the point shared by multiple boundaries is duplicated:

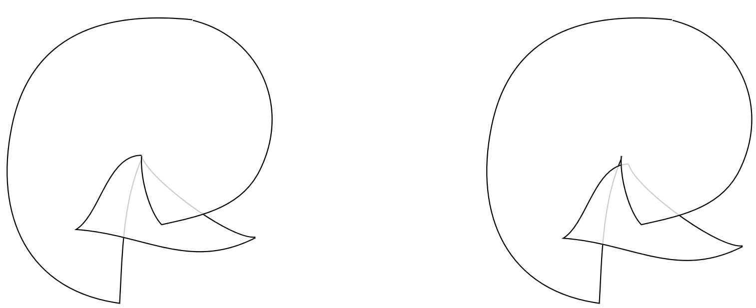


Figure 2.6: At multivalence vertices, we imagine that the individual surfaces do not meet exactly.

We make this distinction primarily for ease of analysis. We will see that this adds complexity for the implementation.

2.3 Occlusion

All visibility problems arise from occlusion of different sorts. To motivate our analysis, we first examine the properties of occlusion on surfaces.

2.3.1 Geometric vs. Projective Definitions

Occlusion can be thought of in two equivalent but representationally different frameworks. In 3D space, an object o_1 occludes o_2 if there exists some ray r emanating from the viewpoint v that first intersects o_1 and then o_2:

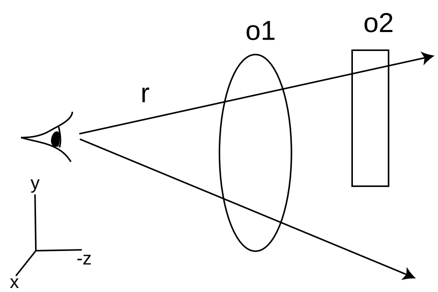


Figure 2.7: o_1 occludes o_2.

The viewer, if looking in the direction of r, would see o_1 but not o_2. This geometric characterization of occlusion is used in raytracing algorithms and similar applications.

Another way of looking at occlusion is after projection. The traditional computer graphics pinhole camera applies a projective transformation to the 3D data that maps the view frustum to the canonical cube. Rays emanating from the viewer into the view frustum are mapped to parallel lines of constant x and y. We denote the projection of a point $c \in R^3$ to be $\Phi(c) \in R^3$. It is convenient to also define the (x, y) coordinates of $\Phi(c)$ as $\phi(c) \in R^2$. We define the projection of some object o as $\Phi(o) = \{x : y \in o \text{ and } \Phi(y) = x\}$. We define $\phi(o)$ similarly.

In projection, an object o_1 occludes an object o_2 if there exists an (x, y) such that $(x, y, z_1) \in \Phi(o_1)$ and $(x, y, z_2) \in \Phi(o_2)$ and $z_1 > z_2$. Essentially, o_1 occludes o_2 if $\Phi(o_1)$ is somewhere "on top of" $\Phi(o_2)$.

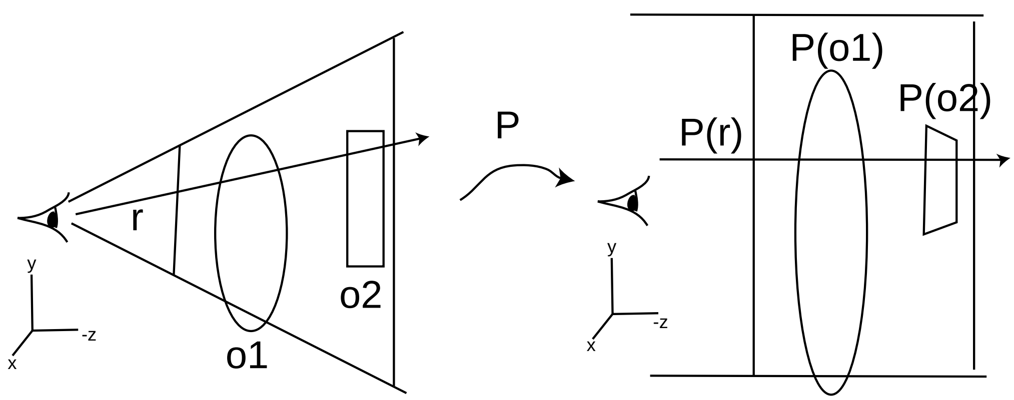


Figure 2.8: The projective transformation maps rays emanating from the viewpoint to parallel lines.

The *preimage* of some set of points $S \subset R^2$ onto some object o is all points:

$$\{p | p \in o, \phi(p) \in S\}$$

The preimage of a pointset on an object, then, is the set of points on the object that project to the pointset. An object o has a *full* preimage of a pointset S if, for every point in the $p \in S$, there exists a point on o that projects to that p.

After application of the transformation, occlusion has been essentially reduced to a $2D$ correspondence problem. This greatly simplifies many occlusion calculations. Because of this, the bulk of our analysis and associated algorithm considers the properties of object occlusion in projection.

2.4 Self-Occlusion

2.4.1 Partitioning by Visibility Curves

Krishnan and Manocha [3] give a method for partitioning a tensor product Bezier patch into a set of non-self-occluding surfaces. They define the *visibility curves* of a region s to be:

$$V(s) = \{p_i | p_i \in int(s), \exists p_b \in \delta s, \phi(p_i) = \phi(p_b)\}$$

That is, the visibility curves are all points p on the interior s such that p projects to the same place as a point p_b on the boundary of s. They proved that (1) the visibility curves induce a partition on the surface, and (2) each region in that partition cannot self-occlude. To yield a set of non-self-occluding surfaces from a patch, they computed visibility curves and partitioned the surface along them.

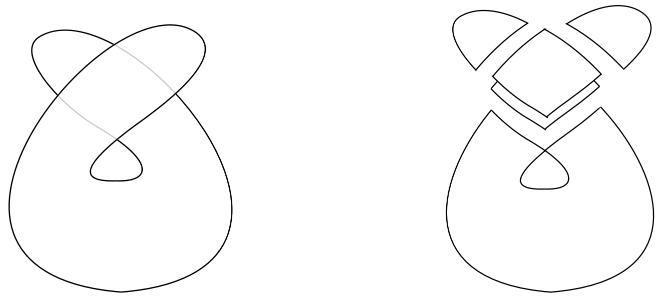


Figure 2.9: A self-occluding object and the partition induced by cutting along visibility curves.

We note, however, that partitioning by visibility curves introduces many unneccesary cuts. Consider this cut for Figure 2.9 instead:

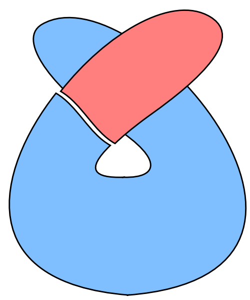


Figure 2.10: Not all the cuts induced by the visibility curves are necessary.

We can decompose this surface into a non-self-occluding set of surfaces with one cut that creates two surfaces. Decomposing along all visibility curves creates 5 subsurfaces and requires 4 separate cuts. In the proceeding analysis we will examine why the simpler partitioning is possible, and give a criterion for determining it.

2.4.2 Properties of Self-Occlusion

To motivate our algorithm for removing self-occlusion, we first must state some properties of self-occluding surfaces.

First, we must define a *cusp* and a *boundary intersection*. A cusp is a point c on a boundary of a surface s such that in any half-disc neighborhood of c, s self-occludes. A boundary intersection is a point on the image plane where two distinct points p_1 and p_2 on the boundary of s project to the same place. We will examine both of these cases in greater detail later. First we will state how these two types of events relate to self-occlusion.

Lemma 2.4.1 *If a surface s occludes itself, it must be the case that the boundary of s intersects itself in projection, or there must be a cusp on the boundary of s.*

Proof We will show that if a surface s self-occludes, then the set of visibility curves of s is not empty. Then we will show that the existence of a visibility curve implies a boundary intersection, or a cusp.

Say s self-occludes. The partitions induced by the visibility curves of s must not self-occlude (shown in [3]); thus, if the set of visibility curves is empty, then there can be no self-occlusion (because in that case the partition induced by the visibility curves is simply the surface itself). So, it must be the case that there exist some visibility curves.

All visibility curves must meet δs (shown in [3]). This implies that the projection of some boundary on the interior of s must touch δs. If the visibility curve and the boundary touch in projection but not on the surface, this implies a boundary intersection; otherwise, the point at which they meet must be a cusp. ∎

This lemma gives us a way to determine when no surfaces occlude themselves; we simply must check to see if any boundary intersections or cusps remain. Note that local to both cusps and boundary intersections, there must be self-occlusion.

2.4.3 Cusps

A cusp of a surface s is a point c on δs where, locally to c, s occludes itself. Formally, c is a cusp if in any half-disc neighborhood of c on the surface, there exists two distinct poitns p_1 and p_2 on s such that $\phi(p_1) = \phi(p_2)$.

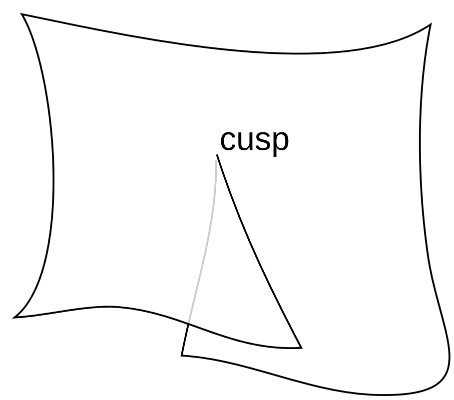


Figure 2.11: A cusp on a surface.

Consider a partitioning of s that produces a set of surfaces that do not self-occlude. Because, local to c, s self-occludes, any such partitioning must split every local neighborhood of c. That means that the partition must split s along a path that is incident to c. Because any proper partitioning must include a path incident to c, we will treat cusps separately from other self-occlusion.

Consider the neighborhood of c. The boundary containing c splits the neighborhood into 3 regions in projection: two $+1$ regions and a $+2$ region(where a region is $+n$ if the increase in depth complexity due to surfaces local to c on the region is n):

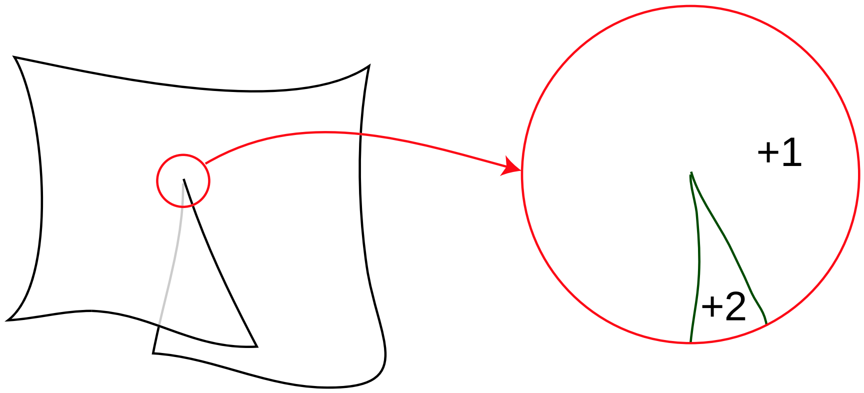


Figure 2.12: Depth complexity local to a cusp.

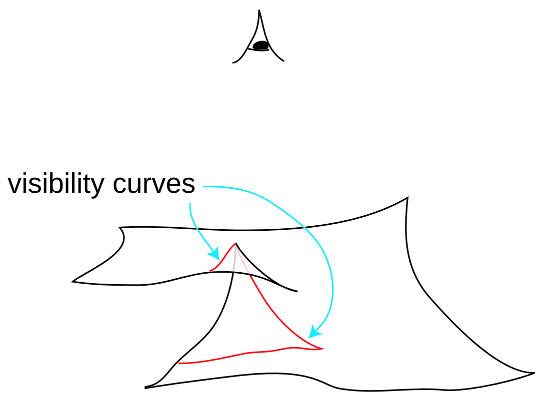
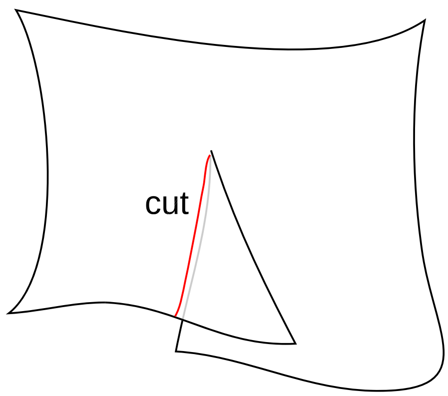
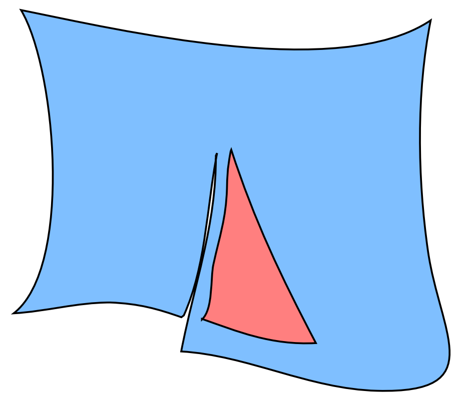


Figure 2.13: A tilted view of the cusp. Visibility curves are in red.

Note that the +2 region in projection corresponds to 2 separate regions on the surface; these are the regions of the surface that self-occlude. A cut that separates these regions in the neighborhood of c will remove local self-occlusion from c. Also note that both regions of the self-occlusion are bounded on one side by δs and on the other side by a visibility curve induced on s by δs local to c. A cut that separates these two regions, then, must lie on the surface between the two visibility curves. We can construct such a cut by cutting along one of these visibility curves, stopping when it hits a boundary (which it must ([3]).



(a) A cut along a visibility curve



(b) The resulting partition

Figure 2.14: Cutting along a visibility curve removes occlusion local to the cusp.

Note that cutting along a visibility curve as described not only removes a cusp, but also avoids the creation of a new self-occlusion “equivalent” to the cusp:

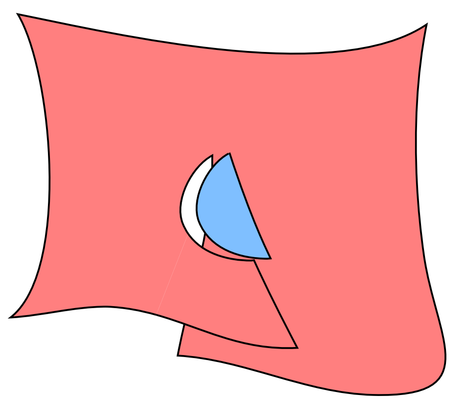


Figure 2.15: This cut removes the cusp, but simply repaces it with a “loop”.

In the case where the cusp c has local self-occlusion of maximal depth complexity 2, cutting in this manner will always make c not a cusp as shown above. However, if c has local depth complexity $k > 2$, this cut will separate the half-disc neighborhood of c into two separate neighborhoods, where one of those neighborhoods must have depth complexity $k > 1$. However, because every such cut yields two neighborhoods of maximal depth complexity at most $k - 1$, repeated cuts in this manner will eventually remove all local self-occlusion.

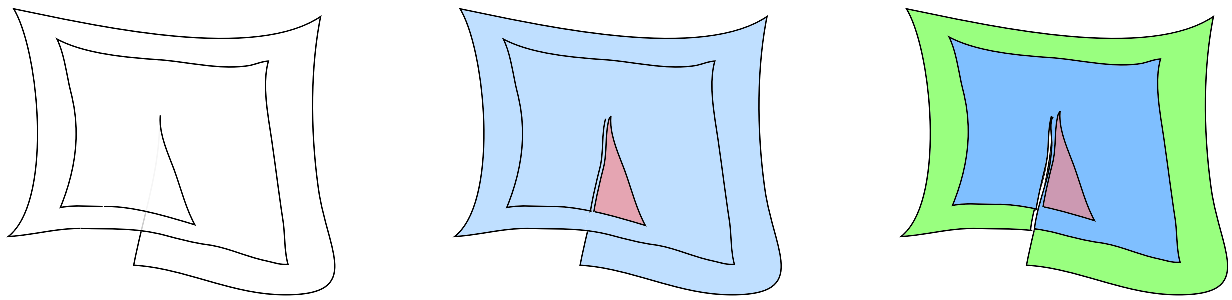


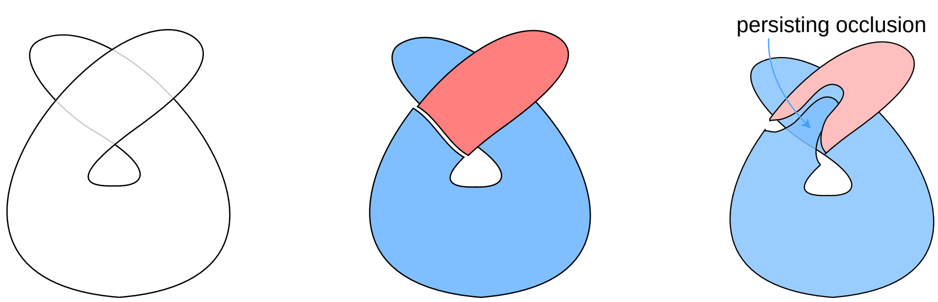
Figure 2.16: A cusp of depth value 3, a cut that reduces the depth value to 2, and another cut that reduces the depth value to 1.

2.4.4 Non-cusp Self-occlusion

Given an arbitrary surface, we showed how to introduce cuts on the surface so that the set of surfaces induced by the cuts have no cusps. Now we will consider non-cusp self-occlusion, so that we have a method for removing all self-occlusion from an arbitrary set of surfaces.

A surface self-occludes if there exists points p_1 and p_2 on the interior of the surface such that $\phi(p_1) = \phi(p_2)$. Each surface induced by our partition must have no such two points.

Consider the simple self-occlusion presented earlier, and the cut that removed the self occlusion:



(a) An object

(b) A good partition

(c) A bad partition

Figure 2.17: The good partition did not leave any persisting occlusion.

The successful cut removed the self-occlusion because it separated the regions where the surface overlapped. That is, all the p_1 and p_2 such that $\phi(p_1) = \phi(p_2)$ were placed on different surfaces in the partition. We observe that any successful cut of this surface must separate these two regions. Because of this, we desire a way of characterizing such regions.

Definition Given k distinct points $P = (p_1, p_2, \cdots, p_k)$ such that $\phi(p_1) = \phi(p_2) = \cdots = \phi(p_k)$, and each p_i is on the interior of some surface s, we define the *k-occlusion region* r_i generated by p_i to be the set of points C such that for every $c \in C$:

- c is on the interior of s
- there exists a path d_i on the interior of s from p_i to c such that:
  - for each p_j such that $j \neq i$, there exists a path d_j on the interior of s containing p_j such that $\phi(d_i) = \phi(d_j)$
  - when $i \neq j$, $d_i \neq d_j$
  - for every such pair of paths d_i, d_j where $i \neq j$ there exists a smooth, one-to-one mapping M_{d_i,d_j} from d_i to d_j that satisfies the following properties:
    * $M_{d_i,d_j}(p_i) = p_j$
    * for every point $q \in d_i$, $M_{d_i,d_j}(q)$ is a point on s distinct from q
    * for every point $q \in d_i$, $\phi(M_{d_i,d_j}(q)) = \phi(q)$

The set of regions $R = (r_1, r_2, \cdots, r_k)$ generated by $P = (p_1, p_2, \cdots, p_k)$ is called the *k-occlusion set* generated by P.

The case where $k = 2$ is of special importance. We call a 2-occlusion set an *occlusion pair*. We see that the two regions of occlusion for the surface in Figure 2.9 form an occlusion pair:

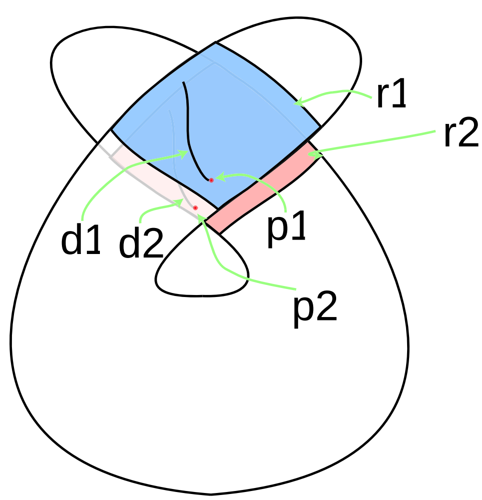
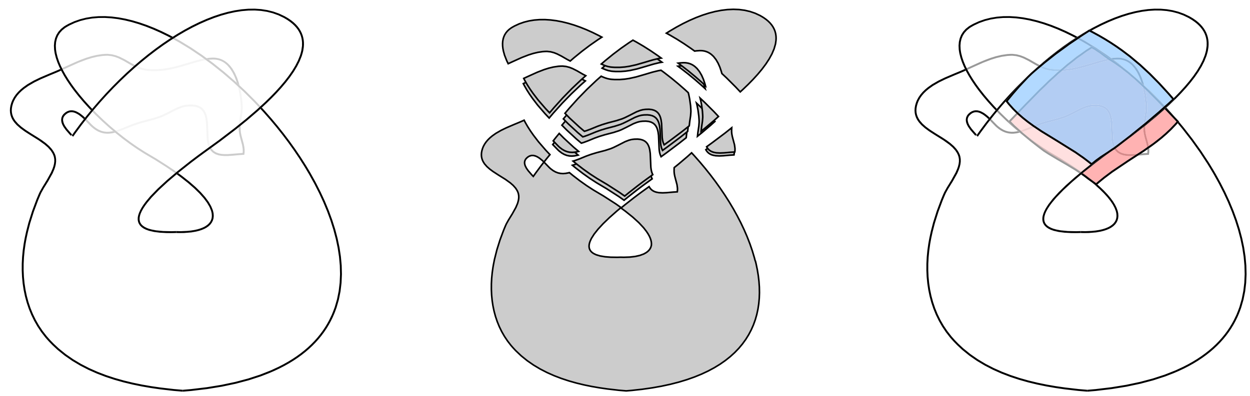


Figure 2.18: The occlusion regions of the snake, and two corresponding paths in those regions.

Removing this block of occlusion entails partitioning the surface so that the two regions of the occlusion pair are in different subsurfaces induced by the partition. This is because the occlusion regions capture all the points where $\phi(p_1) = \phi(p_2)$. For figure 2.18, the occlusion pair generated by points in the occlusion corresponds exactly to surfaces in the partition induced by the visibility curves. However, this generally is not the case. Consider:



(a) An complex object

(b) Visibility Curve partition

(c) An occlusion pair

Figure 2.19: The visibility curves partition the object into many subsurfaces. The occlusion region is unchanged.

Here, the visibility curves partition the regions of occlusion into many subsurfaces. However, notice that points inside the original occlusion still generate the same occlusion pair. This demonstrates that the occlusion pair generated by a set of points is unaffected by boundaries not "salient" to that occlusion. This enables us to reason about locally coherent areas of occlusion without having to bother with boundaries that do not matter to that coherent area. Notice that the good partition on the surface in Figure 2.17 resolves the same block of occlusion for Figure 2.19, even though the visibility curves no longer partition the surface into the occlusion regions being separated. The concept of the occlusion pair thus enables us to consider that region independent of other boundaries of the surface. This leads us to parititon Figure 2.19 like so:

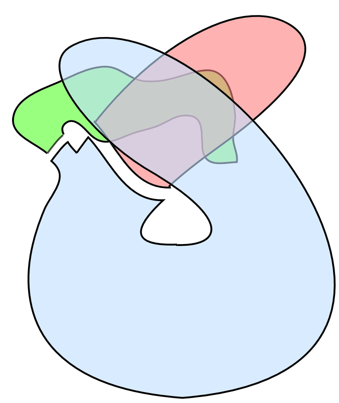


Figure 2.20: Only 2 cuts are required to remove all problem occlusion.

The characterization of occlusion, the occlusion pair, forms the backbone of our analysis and associated algorithm. It enables us to reason about areas of occlusion that are "coherent," independently of other boundaries and surfaces. Thus, from our analysis we can make cuts that remove entire areas of coherent occlusion.

With the definition of the occlusion pair, we can now state our overall approach for handling self-occlusion. Self-occlusion must imply the existence of a boundary intersection or a cusp (by lemma 2.4.1). We will first consider the case of the boundary intersection. Locally to a boundary intersection, there must be two points that occlude each other; we will introduce cuts to resolve occlusion for the occlusion pair generated by those two mutually occluding points. We will repeat this until no boundary intersections remain.

To measure the progress of our algorithm, we will define the *depth map* of a surface s to be a function from R^2 to $\{0, 1, \cdots\}$ defined as such:

$$D_s(\{x, y)\}) = \# \text{ of unique points } p' \text{ on } s \text{ such that } \phi(p') = \{x, y\}$$

The depth map measures the "amount" of self-occlusion of a surface at every point in the image plane. Notice that the depth map changes only at the boundary curves in projection. Thus, the depth map is composed of a set of "cells" of constant D.

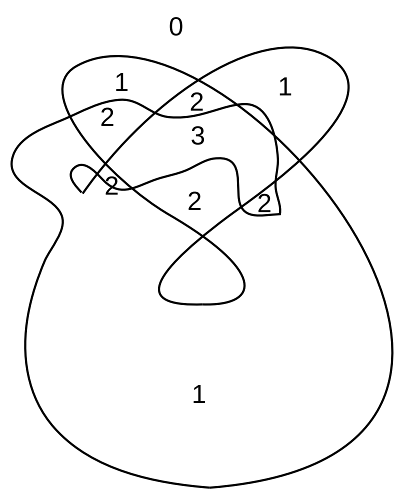
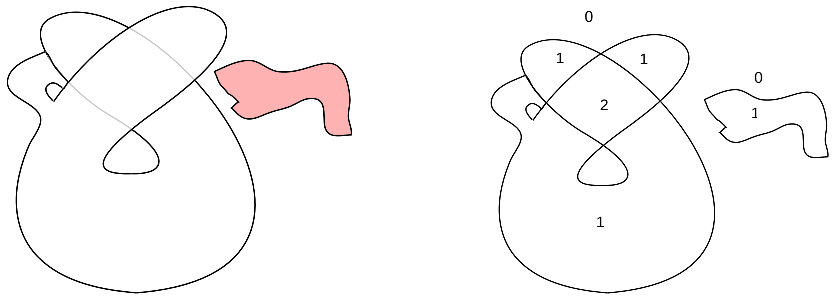


Figure 2.21: Depth map for an object.



(a) A partition along a visibility curve

(b) The resulting depth map

Figure 2.22: A cut along a visibility curve that reduces depth complexity in a set of cells by 1. Note that this cut creates no new cells.

A surface that everywhere in R^2 the depth map is 0 or 1 does not self-occlude. Given some self-occlusion on a surface s, we will show how to partition s into a set of surfaces S such that:

- s is partitioned along cell boundaries
- there exists some cell in the depth map of s such that for every $s' \in S$, the corresponding cell in $D_{s'}$ has reduced depth complexity

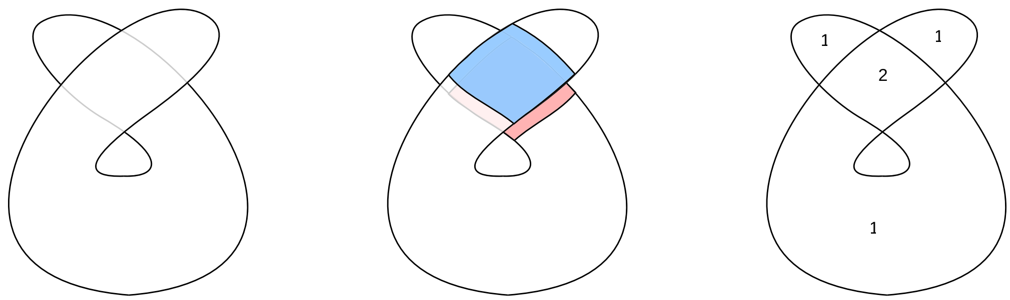
For well-behaved (finite and non-fractal) surfaces, the number of cells in the depth map must be finite; thus, repeated application of such partitions must reduce depth complexity to 1 or 0 for every resultant surface.

2.4.5 Disjoint Occlusion Pairs

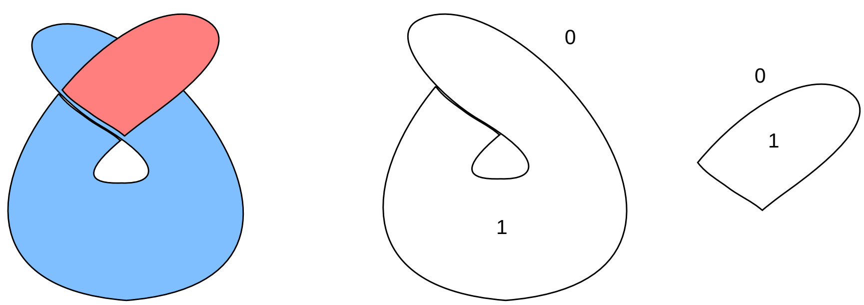
We will first consider the case of occlusion pairs (r_1, r_2) on some surface s such that r_1 and r_2 are disjoint.

Because r_1 and r_2 are disjoint, it must be possible to partition s such that r_1 and r_2 are on separate surfaces (one can trivially construct such a partition by placing r_1 in its own subsurface). For the set of cells corresponding to the occlusion pair, the depth value in the subsurface containing r_1 has decreased by at least 1 (because r_2 is in a different surface). The same fact is true for r_2. Thus, the partition makes discrete progress.

If such a partition occurs on a visibility curve of the surface, then there are no new boundaries in projection, so no new cells are created in the depth map.



(a) The snake (b) The occlusion regions (c) The depth map

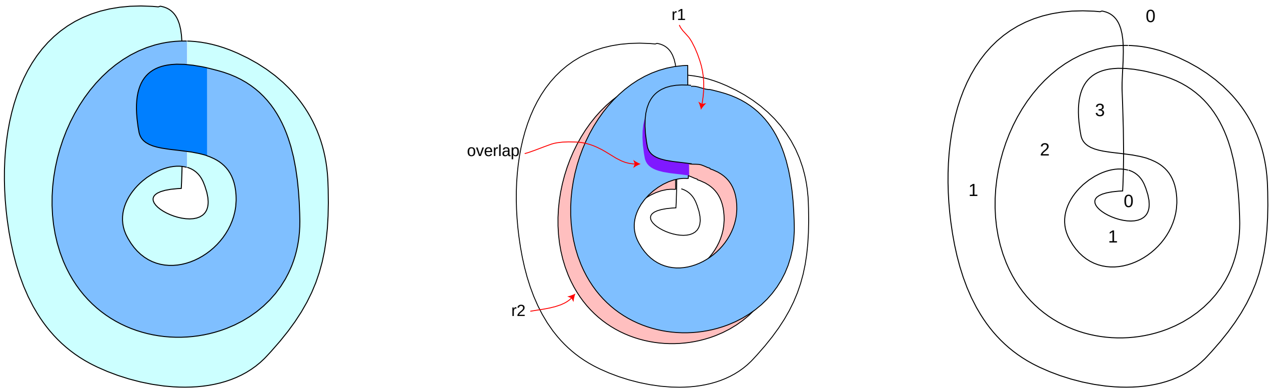


(d) A cut along a visibility curve (e) The resulting depth maps

Figure 2.23: This cut along a visibility curve reduced depth in cells of the depth map and created no new cells.

2.4.6 Overlapping Occlusion Pairs

Now we must consider the case where some regions (r_1, r_2) are not disjoint; that is, there is some set of points both in r_1 and r_2. In this case, one cannot construct a partition that places all of r_1 in a separate subsurface than all of r_2, because they share area of the surface. We must show, then, that in this situation we can make a cut that makes progress even though the regions in the occlusion pair might not be completely partitioned apart.

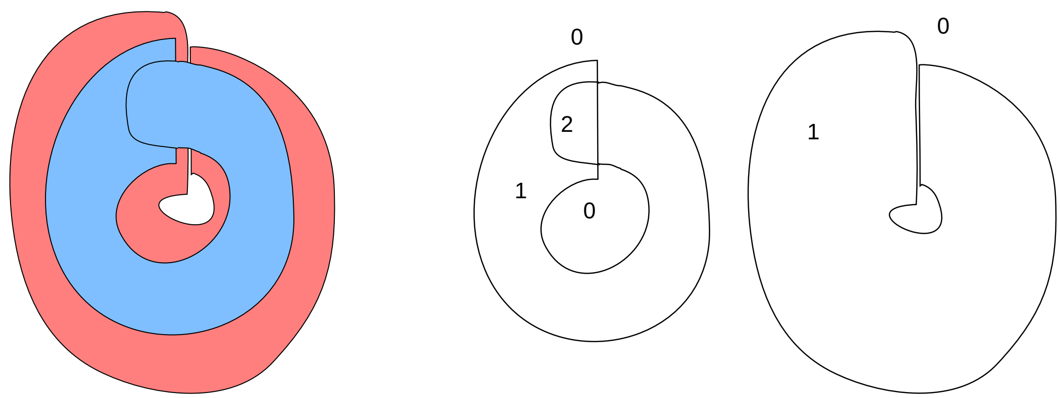


(a) A complex object (b) An occlusion pair (c) The depth map

Figure 2.24: This region has an occlusion pair where the two regions overlap. Notice that the area of overlap has depth complexity 3.

Lemma 2.4.2 *Consider the region in the image plane of $\phi(r_1)$, where r_1 is some region in an occlusion pair $S = (r_1, r_2)$. For every point $c \in \phi(r_1)$ there must exist a point p_1 in r_1 but not in r_2, and a point p_2 in r_2 but not in r_1, and $\phi(p_1) = \phi(p_2) = c$.*

Say we partition a surface s so that some region r_1 in a occlusion pair (r_1, r_2) is separated from all points in r_2, except for the areas where r_1 and r_2 overlap. Then, in the region on the image plane $\phi(r_1)$, the surface that contains all of r_1 has been separated from at least one point in r_2 that projects to the same place in the image plane (such a point must exist by Lemma 2.4.2). Thus, in those cells on the depth maps of the subsurfaces of the partition, the depth value must be less than the depth value of the original surface. Note that we have made progress for all cells of the depth map touched by $\phi(r_1)$ even though we have not placed all of r_1 and all of r_2 on separate subsurfaces (as we were able to do for non-overlapping occlusion pairs).



(a) A cut on the object

(b) The resulting depth maps

Figure 2.25: A cut that separates all of r_1 from $r_2 - (r_1 \bigcap r_2)$. This cut reduces depth complexity in all cells of the depth map that r_1 projects into.

We will construct the cut to lie on visibility curves of the surface. Because the region on the depth map that an occlusion region maps to must be a set of cells, such a partition is guaranteed to (1) reduce depth in some cells of the depth map, and (2) not create any new cells in the depth map.

Note that to make discrete progress as we have defined it, it is not necessary to separate all regions that do not overlap. It is only necessary to separate some parts of the pair that correspond to the same cell in the depth map. We will use this fact later when constructing cuts.

2.4.7 Discussion

Now we have dealt with all types of self-occlusion. We have shown how to remove cusps from a surface by introducing cuts, and we also have given a criterion for a cut that measurably reduces the amount of self-occlusion. For an occlusion pair that does not self-occlude, we showed how to resolve all the occlusion represented by the pair in one partition; for an occlusion pair with self-occluding regions we showed that partitioning apart one of the regions will lower depth complexity by 1 in a coherent area of the surface in projection.

2.5 Multi-Surface Cycles

Now, given an arbitrary set of surfaces, we can partition them so that no surface self-occludes. We will now deal with multi-surface cycles, so that the resulting surfaces can be visibility-ordered.

A multi-surface cycle is a cycle of the form:

$$s_1 \to \cdots \to s_n \to s_1$$

for $n > 1$.

Definition An occlusion cycle C consisting of surfaces $s_1, s_2, \cdots, s_n$ is *locally minimal* if there exists no cycle that contains only surfaces in C that has size smaller than $|C|$.

We call the regions of s_i under $si - 1$ the *under regions* of s_i; we call the regions of s_i over s_{i+1} the *over regions* of s_i. We show in the appendix that if any surface s_i in a locally minimal cycle C is partitioned into S_i so that no $s' \in S_i$ contains both a point in an under region of s_i and a point in an over region of s_i, then no cycle exists containing only surfaces in C and S_i. Thus, the partitioning makes measurable progress that will lead to the removal of all multi-surface cycles.

Such a partitioning is possible because the over and under regions in a locally minimal cycle C cannot overlap (as shown in the appendix).

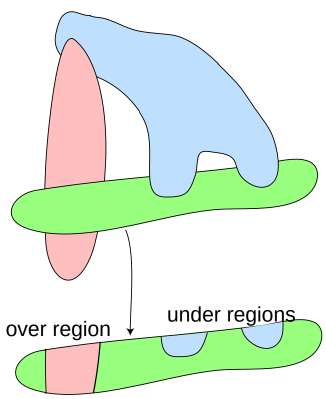
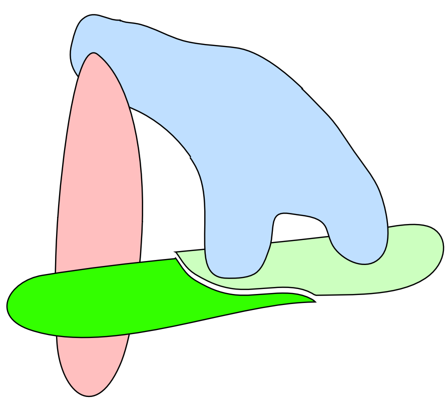
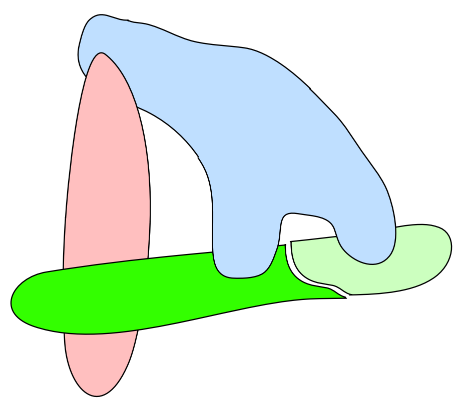


Figure 2.26: A multi-surface cycle and the under and over regions on one of the surfaces.



(a) A good cut



(b) A bad cut

Figure 2.27: The good cut separated the over regions from the under regions; the bad cut did not.

2.6 Constructing Partitions

In the previous sections we gave criteria for partitions that lead to non-self-occluding surfaces, and sets of surfaces without occlusion cycles. Only in the case of cusp removal did we actually construct a cut, however. Now we will go about doing so for the other cases.

2.6.1 Identifying Occlusion Pairs

For non-cusp self-occlusion, the requirement of our partitioning was that it must separate one region in an occlusion pair from the other. For cycle removal, the requirement was that the partitioning must separate "over" regions from "under" regions on the surface. To accomplish both of these tasks, we need a method of determining an occlusion pair corresponding to some boundary intersection.

Notice that the boundaries of an occlusion region must be either (1) boundaries of the surface, or (2) projections of a boundary onto the surface (a visibility curve of the surface). Notice also that the occlusion boundaries have a coherent quality. Consider an occlusion pair (r_1, r_2). Say a point p on δs is on the boundary of r_1, and p', a preimage of $\phi(p)$, is on the boundary of r_2. If p' is not locally near a boundary, then all points local to p on δs are on the boundary of r_1, and the projections of those points local to p' are on the boundary of r_2. If p' is local to a boundary, then that boundary becomes a boundary of r_2 and its projection local to p becomes the boundary of r_1. That place where the boundaries "switched" is a boundary intersection point.

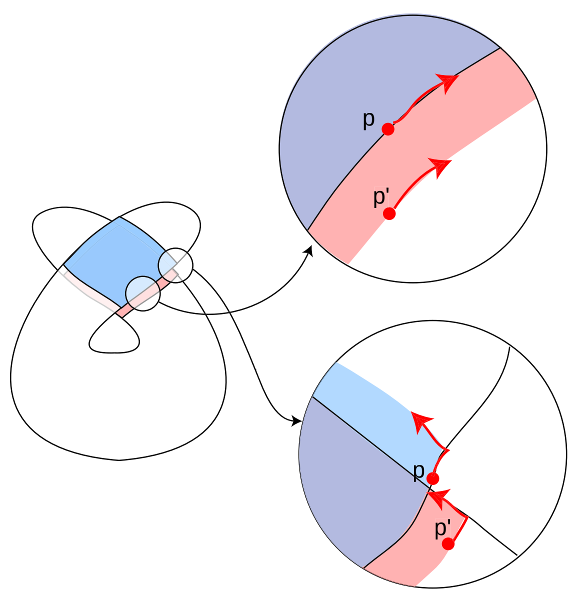


Figure 2.28: Following a boundary of an occlusion region and its projection will outline the occlusion region. Where the projection hits a boundary, "switching" to that boundary will outline the occlusion region.

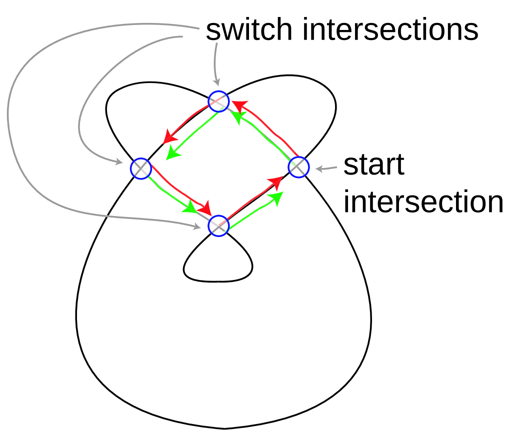
Based on this observation, we have developed an algorithm that identifies the boundaries of an occlusion pair corresponding to some boundary intersection. From a boundary intersection, we follow a boundary outlining the +2 region, and its projection on the surface local to the other boundary. When the projection onto the other surface hits a boundary on the other surface, this must indicate a boundary intersection; from there, we simply repeat the process from the boundary intersection we hit.

Note that because our boundaries have orientation, at each intersection there is only one boundary to follow that will correctly outline the +2 region.

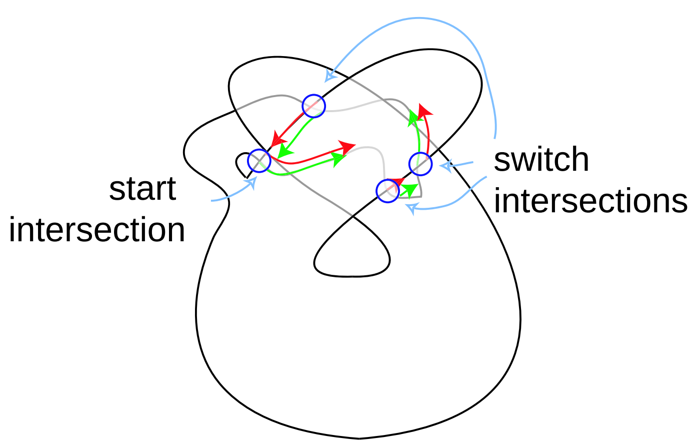
The following pseudocode outlines the FindBoundaries and Railhold procedures. FindBoundaries finds all the boundaries of an occlusion pair corresponding to some input boundary intersection. It uses Railhold to walk from one boundary intersection, following curves in projection, to the next.

```
struct SurfacePoint; //a point on a surface

class BoundaryPoint //a point on a boundary
{
  SurfacePoint p;
  BoundaryPoint Next (); //gets a point local to p on the boundary
                         //in the direction of the boundary
};

class BoundaryIntersection
{
  BoundaryPoint b1;
  BoundaryPoint b2;

  BoundaryPoint FollowPoint ();  //this gets the point on the
                                 //boundary that, if followed,
                                 //will go in the direction
                                 //outlining the +2 region
  SurfacePoint  ProjPoint ();    //this gets the other point

  BoundaryPoint Top ();          //returns the top point
                                 //of this intersection
  BoundaryPoint Bottom ();       //returns the bottom point
};

//this function returns the projection of p1
//onto the local area of p2
SurfacePoint LocalProject ( SurfacePoint p1,
                            SurfacePoint p2 );

{Curve, Curve} FindBoundaries ( BoundaryIntersect biStart )
{
  Curve c1,c2;
  BoundaryIntersect biCur;

  biCur = biStart;

  do
  {
```

```
    biCur = Railhold ( biCur, c1, c2 );
    swap c1, c2;
  }
  while ( biCur != biStart );

  return c1, c2;
}

BoundaryIntersect Railhold ( BoundaryIntersect biStart, Curve c1, Curve c2 )
{
  SurfacePoint  sPoint;
  BoundaryPoint bPoint;

  bPoint = biStart.FollowPoint ();
  sPoint = biStart.ProjPoint ();

  do
  {
    bPoint = bPoint.Next ();
    sPoint = LocalProject ( bPoint, sPoint );

    add bPoint to c1;
    add sPoint to c2;
  }
  while ( sPoint is not on a boundary );

  return BoundaryIntersect ( sPoint, bPoint );
}
```



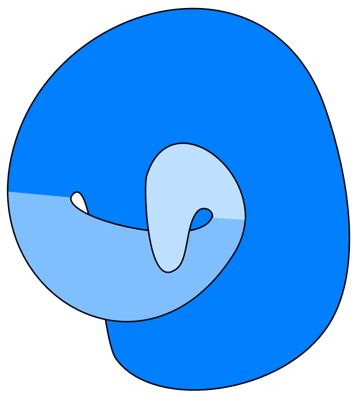
(a) A simple example



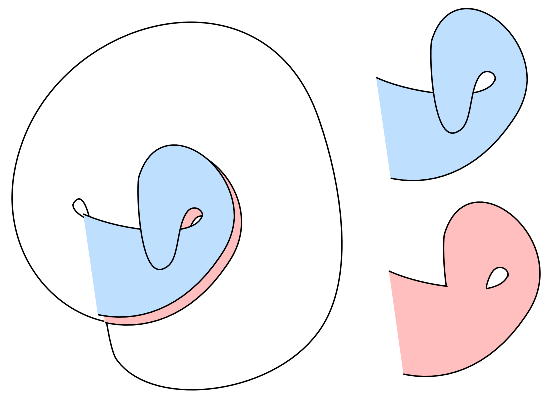
(b) A more complex example

Figure 2.29: Visualizations of the FindBoundaries algorithm on different surfaces. The arrows on the surface are the boundaries that are outlined; the circles enclose "switch" intersections.

Although we offer no proof of correctness for this algorithm, we have observed that in all the cases we have tried, this algorithm (1) correctly identifies a *superset* of the boundaries of the occlusion pair, and (2) the boundaries identified by the algorithm not on the boundary of an occlusion region are in the occlusion region itself. We will see that for our purposes this will suffice.



(a) An object



(b) An occlusion pair on the object

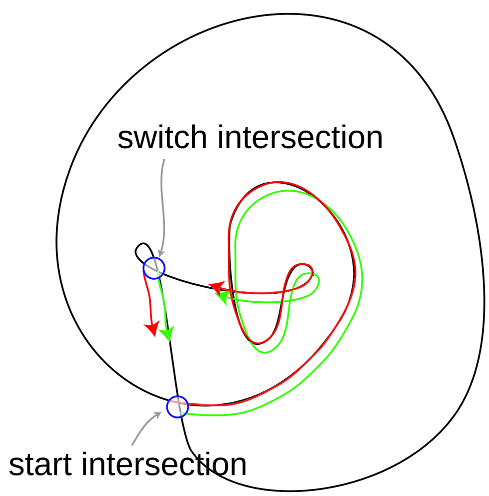


Figure 2.30: Notice that in the bottom occlusion region, part of the projection of the top boundary is not a boundary. In this case FindBoundaries identifies a superset of the boundaries of the occlusion regions.

2.6.2 Cutting

After identifying the boundaries of an occlusion region, constructing a partition that separates it from its mate is relatively easy.

Because our surfaces are discs, we can embed (paramaterize) any surface s as a region in R^2. In such an embedding, the occlusion regions are subregions of s.

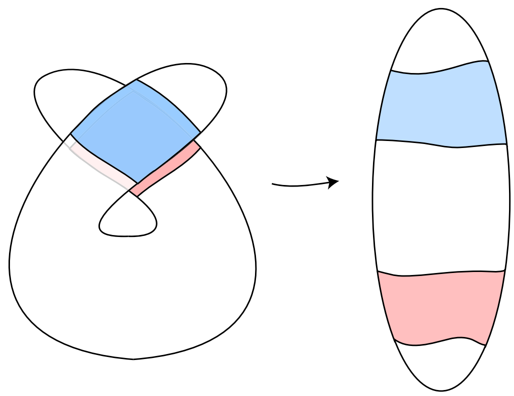


Figure 2.31: An embedding of a disc in the plane.

We could simply introduce cuts along all portions of the boundary of the occlusion region that are visibility curves on the surface. This would yield partitions like so:

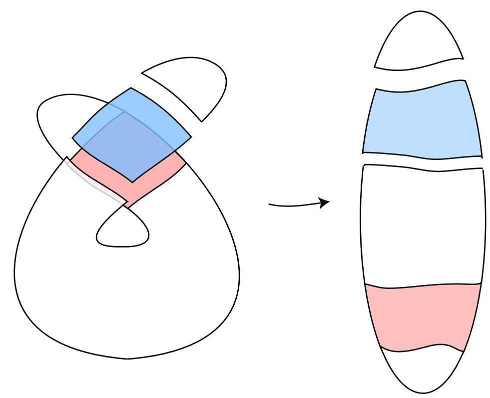


Figure 2.32: Cutting along all visibility curves for one occlusion region makes 2 cuts and creates 3 subsurfaces.

We notice, however, that all of these cuts are not necessary to accomplish our goal. The above figure can be successfully partitioned with one cut:

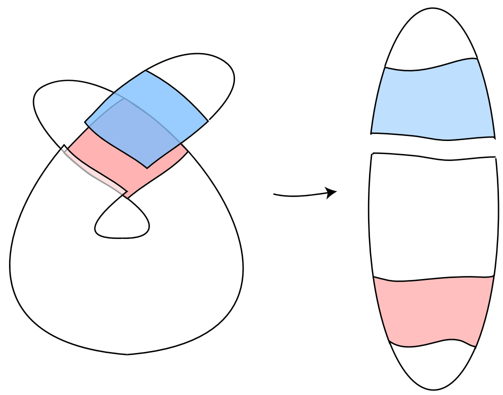


Figure 2.33: Cutting along only one of the visibility curves makes 1 cut and creates 2 subsurfaces.

2.6.3 Discifying Surfaces

To aid us in constructing such cuts on arbitrary surfaces, we first introduce cuts to make the surfaces all discs. We define a disc to be a region r that can be brought into one-to-one correspondence with a region r' that is simply connected (that is, r' can, through continuous contraction, be reduced to a point [9]). Intuitively, a disc is a region that has no "holes."

A disc is desirable because it is a region that has only one boundary; this will enable us to find good places to cut by a simple traversal of the boundary.

Central in this discussion will be the concept of the *winding number*. The winding number of an oriented closed curve measures the number of times the tangent vector of the curve does a full counter-clockwise rotation in one traversal of the curve (with our surface-to-the-left convention). Clockwise rotations are counted as negative turns [10].

This concept has been used in digital topology to distinguish internal boundaries (holes) from external boundaries of surfaces in digital images [10]. External boundaries have a winding number of +1; internal boundaries have a winding number of -1.

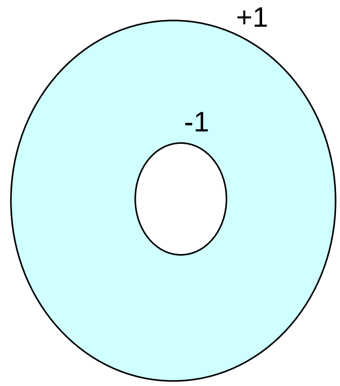


Figure 2.34: A torus with two boundaries: an external boundary of winding number +1, and an internal boundary of winding number -1.

Each of our discs, by definition, can have only one boundary, and that boundary must be external (have winding number +1). Thus, it must be the case that if there exists any boundary of winding number $k \neq 1$ then some surface is not a disc. Our approach to discification of surfaces, then, is to "fix" boundaries of winding number $k \neq 1$ by introducing cuts that connect those offending boundaries to other boundaries.

To do this, we distinguish between cases when the offending boundary intersects itself in projection and when it does not. When the boundary does not intersect itself in projection, then it must correspond to a simple hole of winding number -1. We remove these boundaries by picking the rightmost point on the boundary and cutting rightwards until a different boundary is hit:

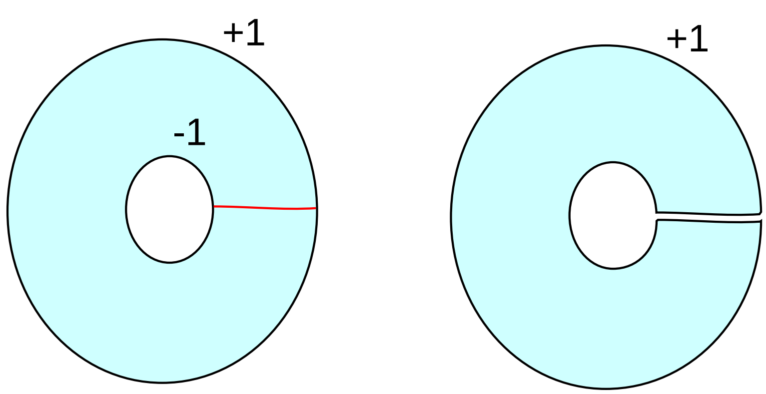


Figure 2.35: Cutting from the rightmost point of the -1 internal boundary of the torus.

When the boundary intersects itself, it may be the case that the "rightmost point" algorithm will not work because the extrema point that we choose may be an "outer" boundary segment. In these cases, we simply cut from the boundary self-intersection until another boundary is reached.

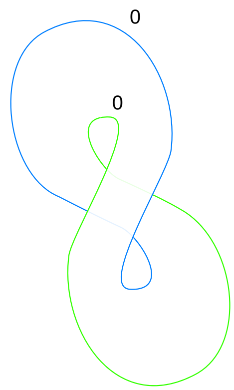


Figure 2.36: A figure eight has two boundaries, each with winding number 0.

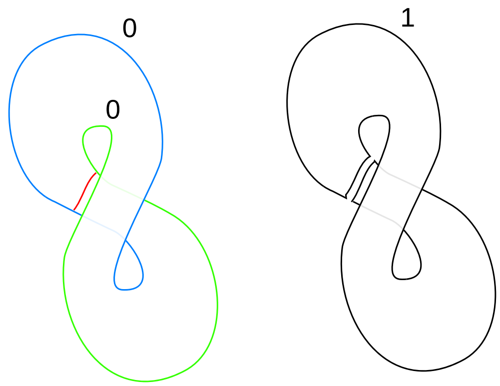


Figure 2.37: A cut from a boundary self-intersection makes the figure eight into a disc.

Although our treatment of this topic is not rigorous, we find that our cuts always result in a set of discs.

2.6.4 Cuts on Discs

Any cut that connects one point on a boundary of a disc to another will partition the surface into two discs. Because we partition along boundaries of the occlusion regions, and occlusion region boundaries must be either boundaries of the surface or visibility curves, our task is to choose a visibility curve along which to introduce a cut. Because any visibility curve must end in the boundary, such a cut will partition the disc into two child discs. Our goal is to choose a visibility curve so that the cut separates as much of the two occlusion regions as possible.

Given a disc-shaped surface s and an occlusion pair (r_1, r_2) on s, we must identify a visibility curve that, when cut along, separates r_1 and r_2. As we saw in the previous section, a cut on a visibility curve that starts “between” r_1 and r_2 will suffice.

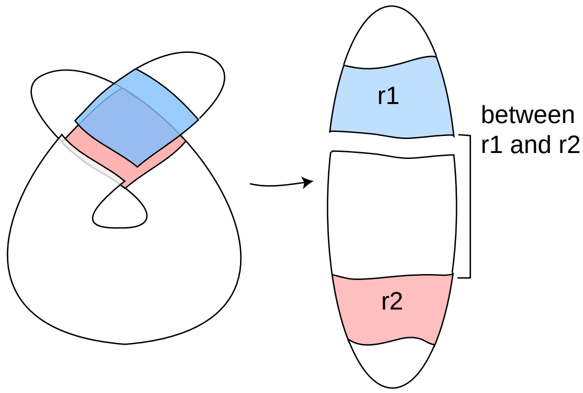
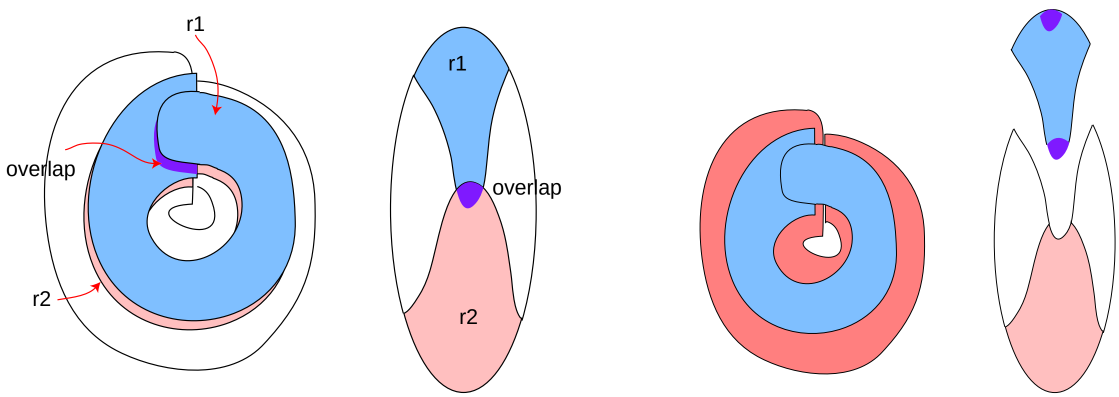


Figure 2.38: A successful cut separates r_1 and r_2. Thus, it must lie on a visibility curve bordering the "between" region.

If r_1 and r_2 do not overlap (as in the figure above), this cut will partition apart the two regions r_1 and r_2 into two different surfaces. If they do overlap, such a cut constitutes our "best guess". In both situations the cut makes measurable progress, as described in the previous sections.



(a) A complex object and its embedding

(b) A good cut on that object and the resulting embedding

Figure 2.39: Because the two regions overlap, no separating cut is possible. However, cutting along a "between" visibility curve constitutes our best guess.

We can find a suitable visibility curve by examining δs. When identifying the boundaries of an occlusion pair (the FindBoundaries procedure), repeated calls to Railhold ended in a set of boundary intersections. The boundary intersections mark the places where the boundary of an occlusion region "switched" from being on δs to being on a visibility curve on s. These "switch" points are points on δs. We simply need to find a switch point on r_1

such that the next switch point is on r_2. When this occurs, we are guaranteed that the cut will lie between r_1 and r_2:

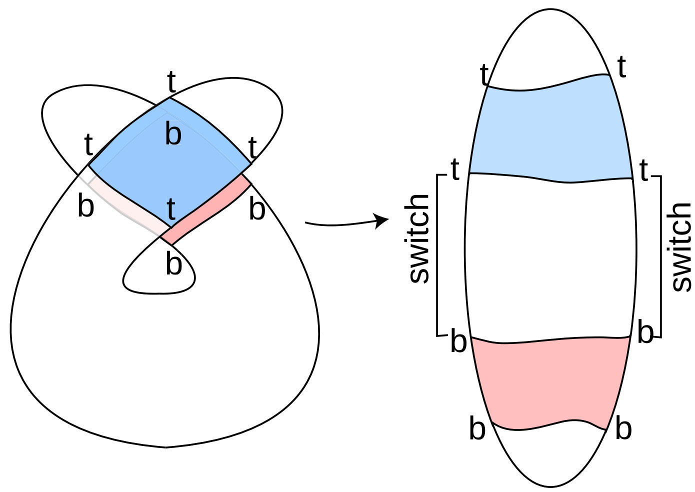


Figure 2.40: Marking each switch in the FindBoundaries procedure enables us to find a good visibility curve to cut by a traversal of the boundary. Top intersection points are marked with t; bottom intersection points are marked with b.

To accomplish this, we modify the Railhold algorithm to report the boundary intersections. We then mark those points on the boundary with the occlusion region (either *top* or *bottom*) that they correspond to. Then, we traverse the boundary of s, looking for a place that switches from *top* to *bottom*.

The following psuedocode outlines the CutStartPoint and FindIntersections procedures. CutStartPoint finds a good place to cut from to remove the occlusion pair corresponding to some input boundary intersection. It uses FindIntersections to find all the boundary intersections for some occlusion pair.

```
enum Mark
{
  TOP, BOTTOM, NONE
};

struct MarkStruct
{
  Mark m;
  BoundaryIntersect bi;
};

//modified version of FindBoundaries that
```

```
//returns the intersections found
IntersectionList FindIntersections ( BoundaryIntersect biStart )
{
  BoundaryIntersect biCur;
  IntersectionList list;

  biCur = biStart;

  do
  {
    list.Append ( biCur );

    biCur = Railhold ( biCur );
  }
  while ( biCur != biStart );

  return list;
}

//this function takes a boundary intersect
//and then gives a boundaryIntersection
//from which cutting along the visibility
//curve adjacent will separate the regions
BoundaryIntersect CutStartPoint ( BoundaryIntersect biStart )
{
  IntersectionList list;
  BoundaryIntersection bi;

  list = FindIntersections ( biStart );

  foreach bi in list
  {
    mark bi.Top () with {TOP, bi};
    mark bi.Bottom () with {BOTTOM, bi};
  }

  BoundaryPoint p;
  Mark lastMark;

  lastMark = NONE;
  p = any point on boundary;

  while(1)
  {
    if ( p is marked &&
         lastMark == NONE )
    {
      lastMark = p.Mark ();
    }
    else if ( p.Mark () != lastMark )
    {
      return p.Mark ().bi;
    }
  }
}
```

2.6.5 Partitioning the Surface

We now have a point to start cutting from. To execute the cut, we simply use Railhold from the boundary intersection, except, instead of adding points to boundary curves, we partition the surface along the curve projected onto the surface.

We use the same procedure to handle occlusion cycles. Remember, the goal was to separate the over regions from the under regions of some surface. To do this, we first identify the over regions, and then the under regions, and then choose a cut that starts between some over and under region. This is guaranteed to separate one over region from one under region. We repeat this until either no over or no under regions remain.

2.7 Overview

We now have a complete algorithm for producing a set of surfaces that can be visibility-ordered from an arbitrary set of input surfaces. This proceeds in several steps.

1. Remove cusps
   (a) Cut along a visibility curve induced by one of the boundaries incident on the cusp
2. Discify all surfaces
3. Remove non-cusp self-occlusion
   (a) Identify a boundary intersection
   (b) Mark the switch points of the occlusion pair for this boundary intersection
   (c) Cut along a proper visibility curve
4. Remove multi-surface cycles
   (a) Identify a locally minimal cycle
   (b) Cut over or under regions in some surface in the cycle until all are removed

The VLD algorithm is general in the sense that it can be implemented for any surface that supports the basic operations defined above. In the next chapter, we will describe our adaptation of the VLD algorithm for triangle meshes.

Chapter 3

Triangle Meshes

3.1 Introduction

To demonstrate the algorithm developed in the previous chapter, we have adapted it for triangle meshes. In this chapter, we will discuss the theoretical and implementational issues encountered in doing so.

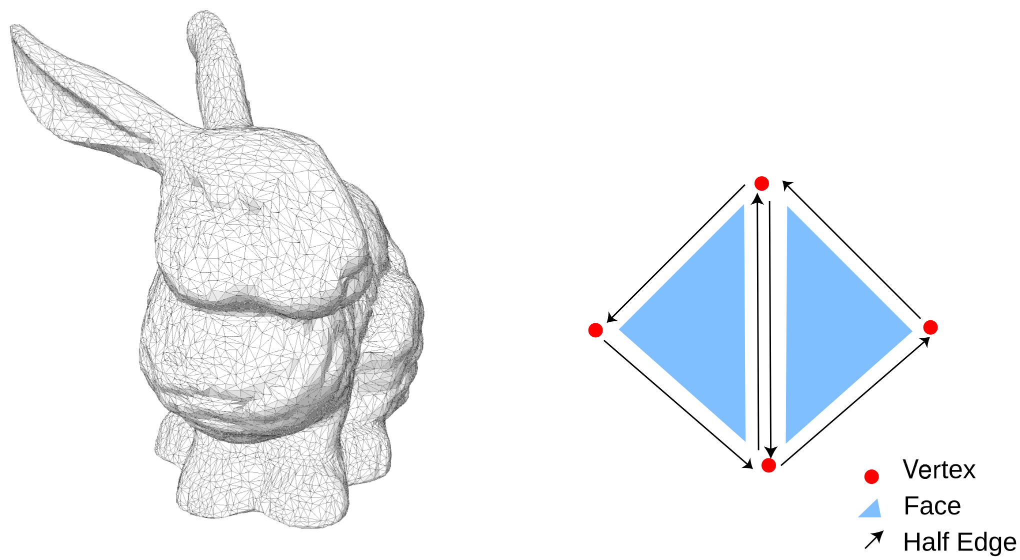
Triangle meshes are the common currency of modern computer graphics. Nearly all of today's graphics pipelines are based on triangle rendering. As detailed in the introductory chapter, Z-buffer is the most commonly used method for visibility resolution in these settings. However, if object-space visibility resolution is required, few options exist; NNS or NNS-like algorithms are most commonly used.

We showed in the previous section that visibility partitioning for non-self-intersecting manifold surfaces can be done by analysis of the boundaries of the surfaces, both on the surface and in projection. The surface is treated as a unit, without an arbitrary pre-partitioning. Our algorithm applied to triangle meshes can thus analyze a mesh as a coherent object instead of a set of individual triangles. This is desirable primarily because the number of silhouette edges on a mesh is usually significantly less than the number of faces in the mesh (Sander et al. reported that silhouette edges are $O(\sqrt{\text{\# of faces}})$ [7]). Thus, our algorithm touches only a small fraction of the faces in the mesh.

In adapting the VLD algorithm for triangle meshes, we have developed or identified ways of accomplishing the basic tasks of the arbitrary manifold algorithm on triangle meshes. The most significant component of this was a way of accomplishing Railholding and partitioning. We will present those first, and discuss issues associated with the solutions we implemented. We will then give an overview of all the steps of the algorithm and discuss results.

3.2 Triangle Meshes

A triangle mesh is a surface made up of triangles (or faces) f_i. Each triangle is defined by a set of three vertices v_1, v_2, v_3, and three *half-edges* between each pair of vertices. A half-edge is a directed line segment connecting two vertices. The vertices are points in R^3. The triangles can meet at their half-edges. In this case, the half-edges that meet are called *mates* of each other:



(a) A mesh (b) Local connectivity

Note that our half-edges are oriented to point counter-clockwise around front-facing triangles. We assume that all meshes we receive as input are closed manifolds; this means that the full neighborhood of any element of the mesh (a vertex, face, or half-edge) is defined. While our algorithm can be applied to arbitrary front-facing manifolds with boundary, our system begins with closed manifolds and uses the silhouette curves to partition the manifolds into front-facing regions.

3.3 Railholding

To successfully implement the algorithm, we must provide a version of Railhold that operates on triangle meshes. Instead of tracking points on boundaries and their projection onto the surface, we instead track a half-edge on the boundary and a face onto which the half-edge projects. The half-edge on the boundary is called the *cut edge* and the face that it projects onto is called the *active face*. Starting from some boundary intersection or a cusp, we set an initial cut edge and active face. The basic idea is that we can track the cut only by looking at the intersections between the edges of the active face and the cut edge in projection.

The following pseudocode outlines the Railhold procedure for meshes.

```
FaceList Railhold ( BoundaryEdge cutEdge, Face sheet face )
{
  FaceList returnList;
  point cutPoint = first intersection of sheetFace and cutEdge;

  while ( !WalkDone ( cutEdge, sheetFace ) )
  {
    if ( there is an intersection of cutEdge and
         some edge e of sheetFace after cutPoint )
    {
      cutPoint = point of intersection;
      sheetFace = face on other side of e from sheetFace;
      add sheetFace to returnList;
    }
    else
    {
      cutEdge = cutEdge->next;
      cutPoint = start of cutEdge;
    }
  }

  return returnList;
}
```

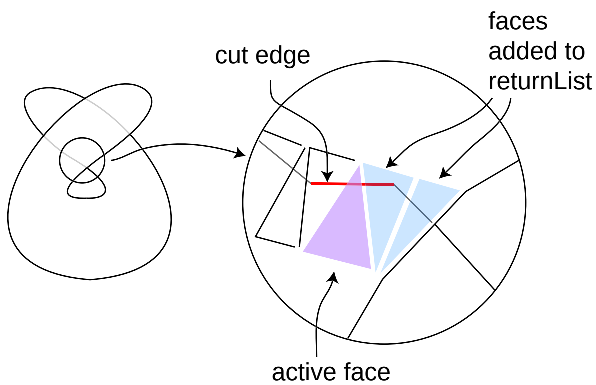


Figure 3.1: A step in the Railhold algorithm.

The output of the mesh Railhold is a set of faces that are touched by the visibility curve. We will show how to construct an approximating cut from these faces in later sections.

3.3.1 Implementation Issues: Numerical Precision

There are several cases we must consider for a robust implementation of the mesh Railhold algorithm. By the algorithm's very nature, it is susceptible to numerical precision problems. In the Railhold algorithm, numerical issues are most dangerous when a vertex of the cutedge projects exactly onto an edge or vertex of the active face. Such situations are problematic because intersections between the cutedge and the active face are used to determine the next active face. If the intersection occurs at an edge or a vertex, it is unclear what the next active face should be.

Consider the following situation:

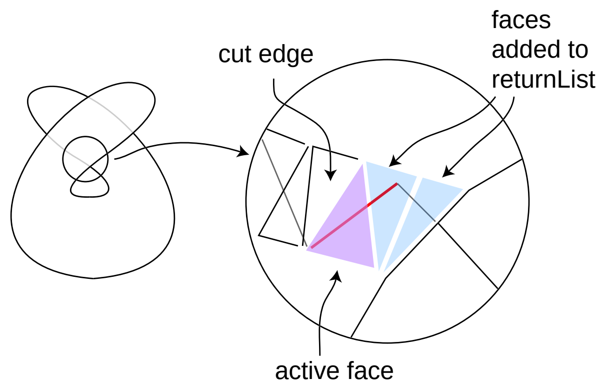


Figure 3.2: A vertex of the cutedge projects exactly onto a vertex on the active face. The next active face cannot be determined.

Here the endpoint of the cutedge lies exactly at a vertex of the active face. There is no intersection between the active face and the cutedge to indicate a new active face; the algorithm cannot progress. A solution to this problem is to keep an *active face set* instead of just a single active face. In most situations, the active face set simply holds the active face given in the old algorithm; but, in the situation in figure 3.2, all the faces in the neighborhood of the vertex of the active face would be added. Intersections would be tested with the outer boundary of the neighborhood.

A similar problem exists when the cutedge ends exactly on an edge on the active face:

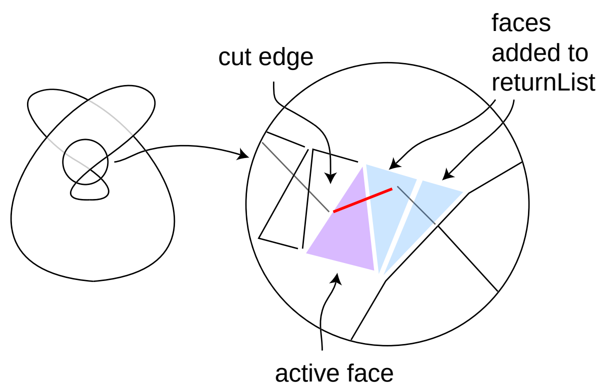


Figure 3.3: A vertex of the cutedge projects exactly onto an edge on the active face. The next active face cannot be determined.

Here we add both faces adjacent to the edge of the active face, and test intersections with the outer boundary of the two faces.

In practice, these problems hardly ever occur and so these modifications have not been implemented in our system.

3.3.2 Implementation Issues: Vertex Intersections

In the analysis in the previous chapter, we assumed that boundaries could not meet at points; boundaries that were coincident were separated by an epsilon distance. However, this is not possible on meshes because we do not actually alter the geometry of the mesh. Thus, it is possible to have multiple boundaries meeting at a single point. In some cases, this meeting of boundaries constitutes an "intersection":

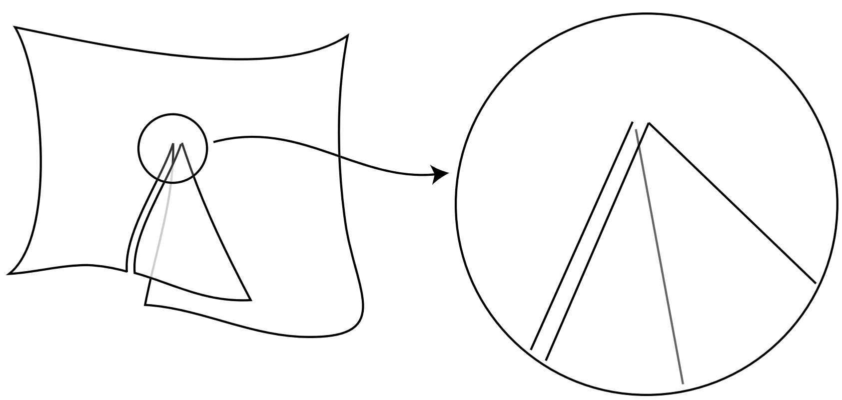


Figure 3.4: A previously cut cusp becomes a vertex with more than two boundary edges incident to it. In this case, this is an "intersection" exactly at the vertex

At other times, two boundaries simply meet at a vertex that would correspond to boundary "tangency":

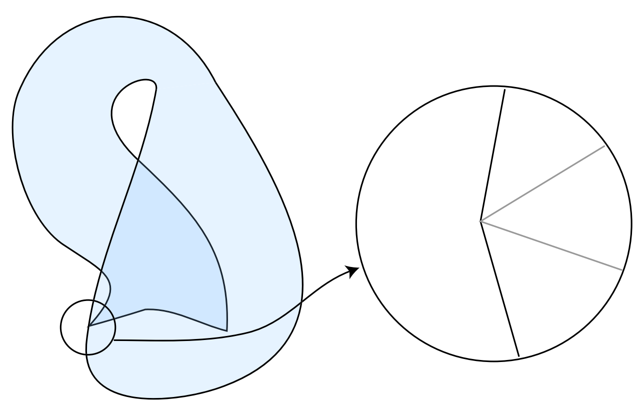


Figure 3.5: In this case, two boundaries meet at a vertex.

All of these cases must be handled specially by all parts of the algorithm that deal with (1) walking along the mesh and (2) boundary intersections. Most importantly, the Railhold algorithm needs to be modified to recognize and deal with these cases.

3.4 Non-exact Cuts

In the analysis in the previous chapter, we assumed that all of our cuts on the surface occurred on arbitrary curves. Doing so on meshes would be very expensive and also unnecessary; a cut that does not lie on an edge in the mesh would require changes in the connectivity of the mesh. Doing this on a large scale would greatly affect performance.

To address this, we will allow the cuts we actually make on the surface to only approximate the visibility curves on the surface. This will allow us to cut along the edges of the mesh that most closely match the cut. The basic idea is that we choose a cut that lies on the "outside" of the faces marked by a single call to Railhold that corresponds to the cut:

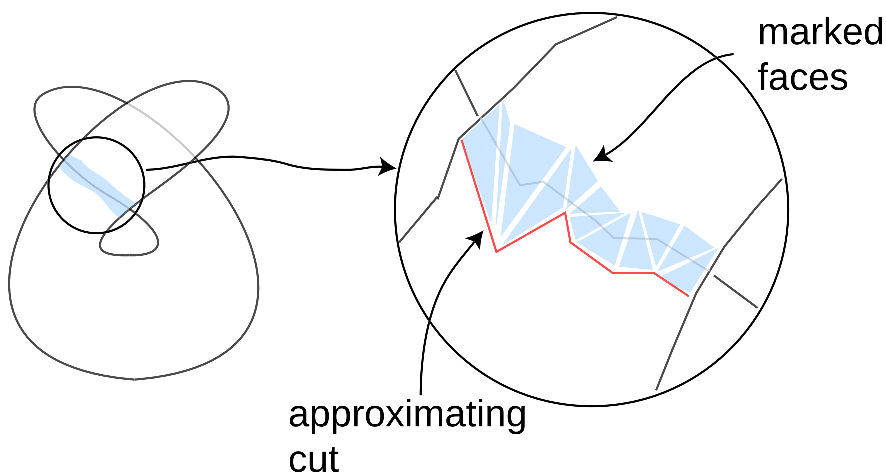


Figure 3.6: An inexact cut.

Note that doing so requires special care; the cut must be constructed to lie on non-boundary edges of the mesh only. This requires, among other things, that the "outside" edges of the marked faces that lie on boundaries be skipped and not cut along. This also implies that the cut can end prematurely, which will be discussed in the next section.

3.4.1 Discussion of Inexactitude

Because our cuts are inexact, many of our theoretical guarantees no longer exactly apply. Perhaps the most pressing is that we can no longer guarantee that no new cells in the depth map are created by a partition. This breaks our basis for showing that our self-occlusion removal procedure was guaranteed to terminate. We observe, however, that any non-intersecting mesh can be partitioned into non-self-occluding surfaces simply by partitioning each triangle into its own surface (this is a new guarantee for termination in the triangle mesh case). Even with this guarantee, our experience with our system shows that it is rare that individual triangles are partitioned apart for this reason.

The other main concern about inexact cuts is that it might not be possible to partition along the entire outer boundary of the marked faces from the Railhold procedure:

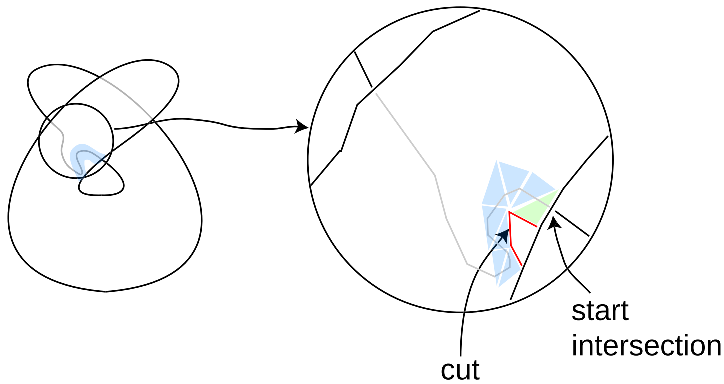


Figure 3.7: A cut that prematurely hits a boundary.

Our solution to this in our implementation is to simply make the first cut we can. If the other cut has to be made, it will be made in another iteration of the occlusion resolution algorithm.

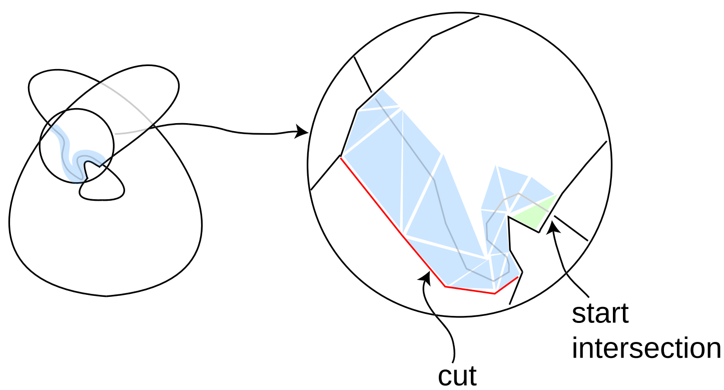


Figure 3.8: More cuts will complete the approximating cut.

In the case of occlusion cycles, it might be the case that in order to remove the cycle making cuts across triangles is necessary (in fact, any occlusion cycle consisting only of single triangles is such a cycle):

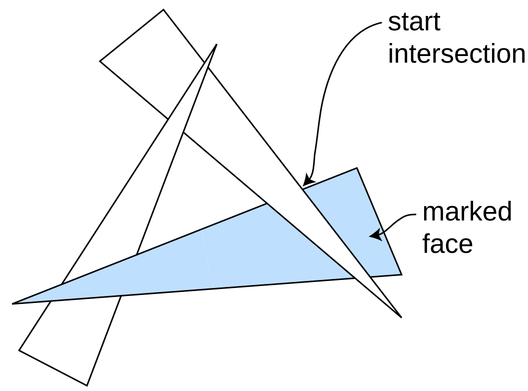


Figure 3.9: This cycle cannot be cut along mesh edges.

Our system currently will not cut the triangles in this situation; we simply ignore the cycle and choose some (incorrect) depth-ordering for the triangles in the cycle.

3.4.2 Finding Boundary Intersections

We used lemma 2.4.1 to show that self-occlusion has been removed if there are no boundary self-intersections left in a surface boundary. We will see that boundary intersection queries are also very important to multi-surface cycle finding. To facilitate general boundary intersection queries, we use a bin data structure that divides the relevant portion of the image plane into an array of bins. Each bin keeps track of which boundary half-edges in projection touch the bin. To test whether some half-edge h intersects any edges in the bin, we only need to test intersections with the edges in the bins that h touches.

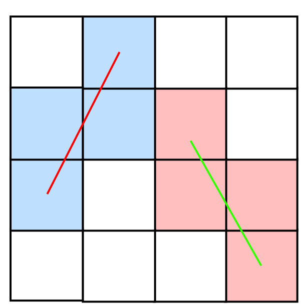


Figure 3.10: These two edges will not have to be intersected against one another because they do not overlap in the edge bin.

In practice, we maintain an intersection cache that holds for each boundary half-edge which other boundary half-edges it intersects. This data structure is built through the edge bin. Whenever a boundary half-edge is created, it first is added to the bin. Then the bin reports which other boundary half-edges it intersects; this information is added to the intersection cache. To test intersections between some boundary half-edge h and all others, the cache simply looks up the intersection information for h.

Queries to this cache data structure have the option of intersecting against only edges that belong to a particular boundary. This enables fast testing of whether a boundary intersects itself, or whether one specific boundary intersects another specific boundary.

3.5 The Algorithm

The algorithm implemented proceeds in several steps:

1. Silhouette Identification and Boundary Creation
2. Winding Number Computation
3. Cusp Removal
4. Discification
5. Self-occlusion Removal
6. Multi-Surface Cycle Removal

We will review these steps in order.

3.5.1 Silhouette Identification and Boundary Creation

All of the surfaces we given the cycle removal stages must be regions of front-facing polygons. Because our initial object contains both front-facing and back-facing triangles, we must find the silhouette of the object. The silhouette forms the initial boundaries of the surfaces we will pass to the later stages of the algorithm.

To identify silhouettes, we use tree-based fast silhouette extraction as presented by Sander et al.[7]. This gives us "edge soup;" that is, the silhouette edges are unordered. We keep track of boundaries as doubly-linked lists of half-edges, where each edge is linked to the next and previous boundary half-edge. Because of this, we need to first construct the boundary data structures from the silhouette edges. The boundaries are directed so that the front-facing surface lies to their left.

Our algorithm is to pick a silhouette half-edge out of the edge soup and walk along the surface, building the boundary for that surface, marking all the silhouette edges we add to the boundary. When we have built the entire boundary (when we reach the first edge), we start from the next unmarked edge in the edge soup, until all edges in the edge soup are marked.

3.5.2 Winding Number Computation

To discify surfaces, we need to know the winding numbers of all the boundary curves identified in the silhouette extraction step. To do this, we notice that in the discrete case the winding number of a curve in projection can be calculated as:

$$\frac{\sum_{edges} \pi - \text{ subtended surface angle between this edge and the next}}{2\pi}$$

To calculate the subtended angle, we sum the projected angle of the portion of the triangle fan on the endpoint of the edge on the surface:

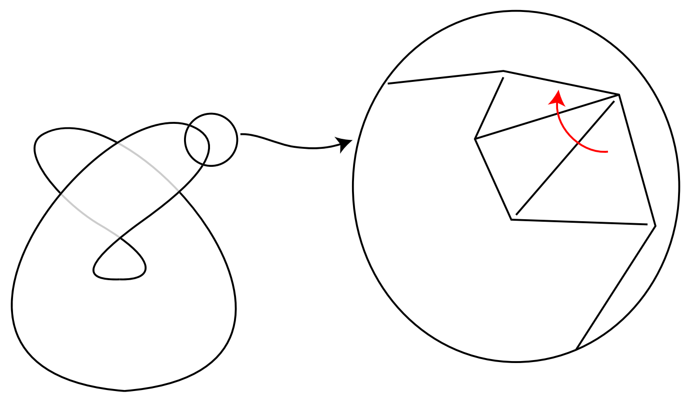


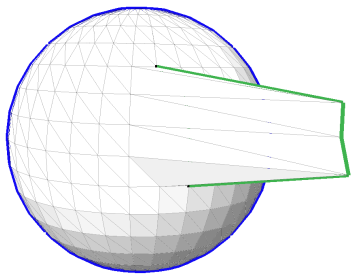
Figure 3.11: The projected angle of a triangle fan around a vertex is used for winding number calculation.

We store this value at the edge for later calculations, since after making cuts, winding numbers may have to be recalculated. When we make a cut, the winding numbers of the boundaries are updated to reflect the new boundaries created by the cut.

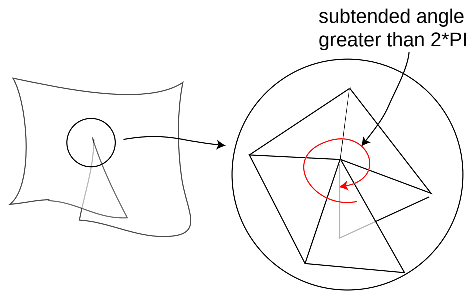
3.5.3 Cusp Removal

We can identify cusps as vertices where the subtended surface angle of the triangle fan on the surface is greater than 2π. Our method for removing cusps was to cut along the visibility curve of one of the boundaries on the surface. We will do this using the Railhold algorithm, started with one of the boundaries as a cut edge and a face in the triangle fan that contains the edge in projection as the active face. We will then make the cut along the outer boundary of the marked faces.

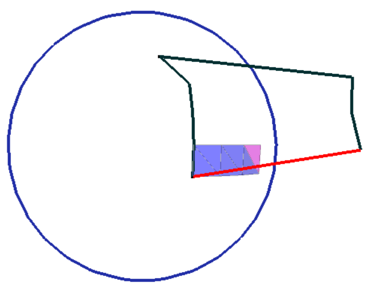
We find the initial active face by walking around the triangle fan from a boundary edge until the subtended angle is greater than 2π. This is guaranteed to provide a face that contains the projection of the boundary edge.



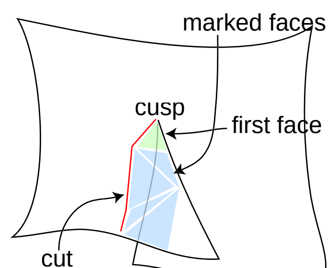
(a) A cusp on a mesh



(b) A diagram of a cusp on a mesh



(c) Cutting the cusp



(d) A diagram of a cusp cutting

Figure 3.12: Cutting a cusp. The left figures are screen captures from intermediate stages of our system.

3.5.4 Discifying Surfaces

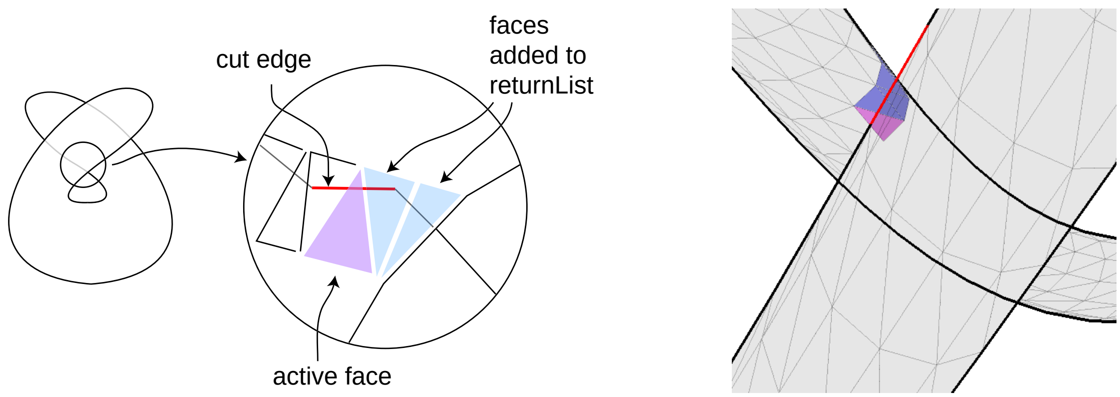
We then discify all surfaces. This proceeds as explained in the previous chapter; either we create a cut that leads rightwards from the rightmost point of a internal boundary, or we cut from some boundary intersection on an internal boundary.

3.5.5 Non-Cusp Self-Occlusion Removal

Now we move to non-cusp self-occlusion removal. We know that any self-occlusion implies a boundary self-intersection; we use the edge bin to find boundary self-intersections.

Once we have found a boundary intersection, we find the occlusion region associated with that boundary intersection. This is accomplished (as detailed in the previous chapter) by repeated calls to the Railhold algorithm.

To Railhold from an boundary intersection, the cut edge is set as the half-edge of the intersection pointing into the other surface; the active face is set as the face adjacent to the other edge:



(a) Diagram of Railhold (b) Railhold on a mesh

Figure 3.13: Railholding. The left figure is a screen capture from an intermediate stage of our system.

As we railhold, we log the intersections we reach in a linked list. After we have found all the intersections, we then traverse the boundary and look for a point that switches between an "under" intersect point and an "over" intersect point. We then do one last Railhold and than make the associated cut from the intersection found in the previous step.

This process is repeated until no boundary self-intersections remain.

3.5.6 Multi-Surface Cycle Removal

The last step is to remove multi-surface cycles. To do this, we first build the occlusion graph induced by the surfaces. To do this quickly, we notice that because we are only interested in locally minimal cycles we only need to be concerned with occlusion that occurs at boundary intersections between surfaces.

Lemma 3.5.1 *No locally minimal cycle can contain an arc $s_1 \rightarrow s_2$ where δs_1 and δs_2 do not intersect in projection.*

Proof We will prove this by contradiction.

Assume a locally minimal cycle C contains a link $s_i \rightarrow s_{i+1}$, where s_i and s_{i+1} have no boundary intersection. It must then be the case that either s_i must be completely contained within s_{i+1} in projection, or vice versa. If s_i is completely contained within s_{i+1} we know the arc $s_{i-1} \rightarrow s_i$ exists in the graph; this implies the existence of p_{i-1} and p_i such that p_{i-1} occludes p_i. But, because s_i is completely contained within s_{i+1}, then there must also exist a point p_{i+1} such that p_i occludes p_{i+1}. Thus, we know that p_{i-1} occludes p_{i+1}. This implies that $s_{i-1} \rightarrow s_{i+1}$ exists in the graph, which means that the cycle:

$$s_1 \rightarrow \cdots \rightarrow s_{i-1} \rightarrow s_{i+1} \rightarrow \cdots \rightarrow s_n \cdots \rightarrow s_1$$

exits in the graph, which has size $|C| - 1$, which means that C is not a minimal cycle.

The case where s_{i+1} is completely contained within s_i can be proven similarly. ■

We use this fact to simplify cycle finding. All boundary intersections have already been calculated and put into the intersection cache; we can build the subset of the occlusion graph we need by simply iterating through all the entries in the intersection cache and recording the boundary occlusion relationships they imply.

We use a simple depth-first search to find cycles in the graph. The cycle we find is not neccessarily locally minimal, however. Therefore, we then do a breadth-first search from some element in the cycle to find the smallest cycle containing that element.

In the system currently implemented, we simply identify a visibility curve lying at a switch between an "under" and "over" region, and cut along it. If this is repeated, it will eventually remove all cycles, although a final implementation would no doubt attempt to operate on a single element of a cycle repeatedly until the cycle is broken.

Note that we ignore cycles which are unbreakable by cuts along edges. We detect this by noting when no cut exists that does not lie on boundaries already extant on the surface.

Finding cycles in this manner for large scenes is prohibitively expensive, and this algorithm is intended only to process the relatively few layers created by the decomposition of one object. For a real-time system, using an algorithm such as IVS[8] to identify cycles would be much faster. In such a framework, this research would be used to resolve problematic visibility events such as self-occlusion and multi-surface cycles (cases where IVS simply aggregates objects), while IVS would actually identify those problematic events.

3.5.7 Visibility Ordering

After removing all multi-surface cycles, we have a set of surfaces that can be visibility-ordered. Now we must create a visibility ordering of the surfaces. To do this, we notice that we can quickly identify all arcs in the occlusion graph due to surfaces whose boundaries intersect in projection simply by looking at the intersection cache (much as we did for cycle finding). However, we must also identify arcs in the occlusion graph due to surfaces whose boundaries do not intersect in projection.

For each pair of boundaries that do not intersect, we first check to see if the extents of one of the boundaries is completely contained within the extents of the other. If not, then this pair has no arc between them. If there is complete containment, we then must apply more aggressive tests. Call the surface whose extents contains the other s_b and call the smaller surface s_s. We choose some random point on s_s called p_s. If p_s projects inside the projection of s_b, then it must be the case that a ray shot in projection in any direction must hit an odd number of edges of the other surface. If this is not the case, then no arc exists between the two surfaces.

If it is the case, then there is some arc between the two surfaces. We must now find out which surface is the occluder and which is the occludee. From p_s, we pick the first edge on s_b directly to the right. Then, from that edge, we walk along the surface to the left until we are on the face that contains p_s in projection. Since post-projection z is linear, it is simple to calculate which surface occludes which at $\phi(p_s)$. This is the arc we must add to the occlusion graph.

Once we have added all the arcs to the occlusion graph, a simple topologial sort will yield a visibility ordering.

Again, this is an area where an algorithm like IVS could be used to allow this work to scale to large scenes.

3.6 Overview

In this chapter we presented our adaptation of the VLD algorithm for triangle meshes. We discussed how to implement the basic elements of the algorithm for meshes, and also special issues encountered in doing so. We also presented a modification of the algorithm to allow for inexact cuts that greatly increases the efficiency of the algorithm.

Chapter 4

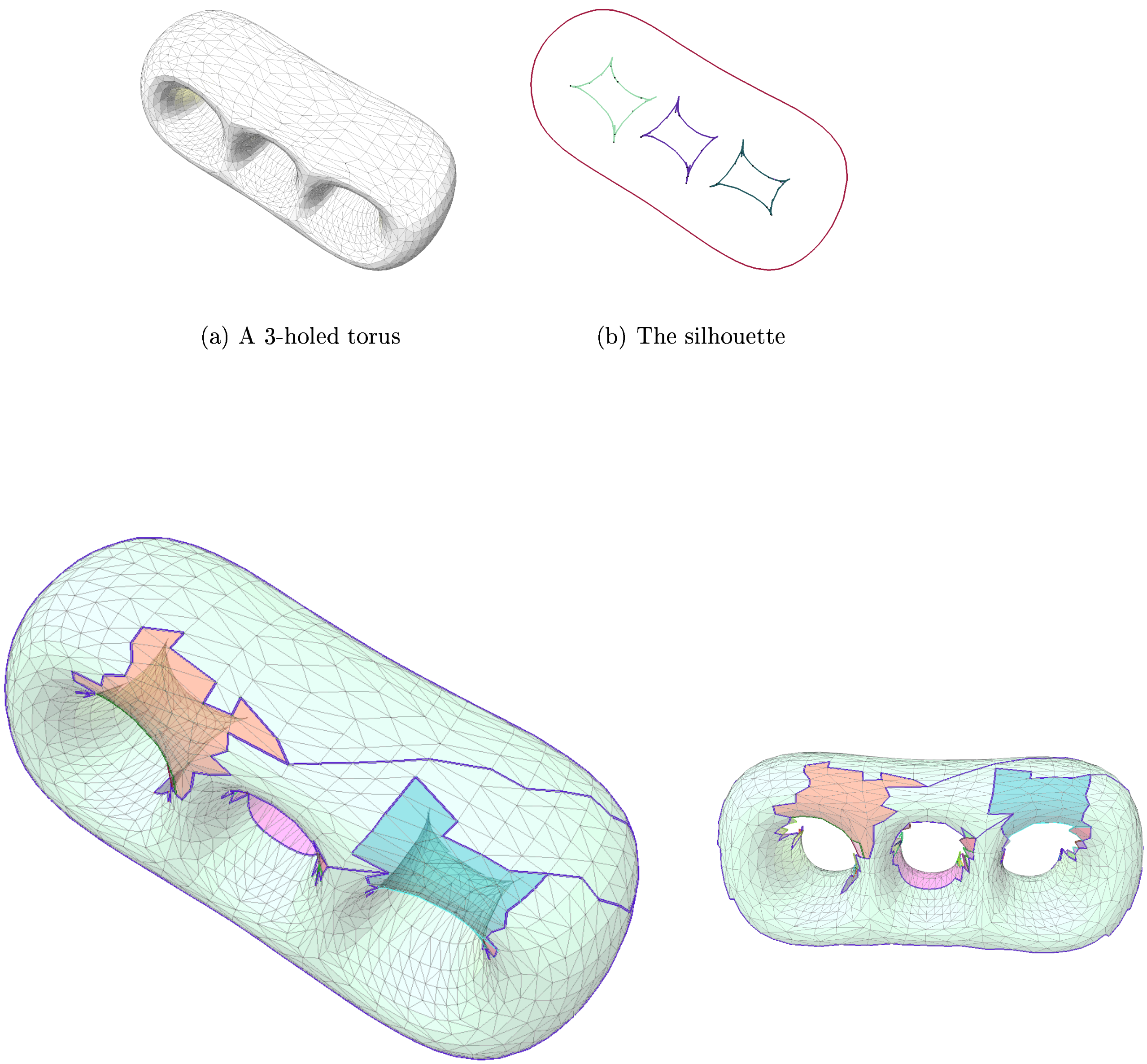
Results and Discussion

We have presented an algorithm for partitioning a set of surfaces S into a new set of surfaces S' such that S' can be visibility ordered. In doing so, we introduced the concept of the *occlusion set*, which enabled us to reason about coherent areas of occlusion. With this definition, we devised methods for resolving problem features of the occlusion graph. The resulting algorithm is general, and can be used on any type of surface so long as the basic tools can be developed.

We also discussed how to adapt this algorithm for triangle meshes, and a relaxation of the "exact cut" assumption that enabled us to do most of the algorithm while maintaining the connectivity of the mesh. From this, we devised an algorithm that was suitable for real-time applications.

4.1 Implementation Results

The development of the triangle mesh VLD system is ongoing. Our system currently fully supports the steps up to and including removal of self-occlusion as described above. Our system has support for removal of occlusion cycles, although cases involving vertex-vertex intersections are not always handled correctly. We also do not yet support creation of a visibility ordering from our occlusion-independent sheets. However, while the system is not completely robust, it still produces a set of visibility-independent sheets for most inputs.



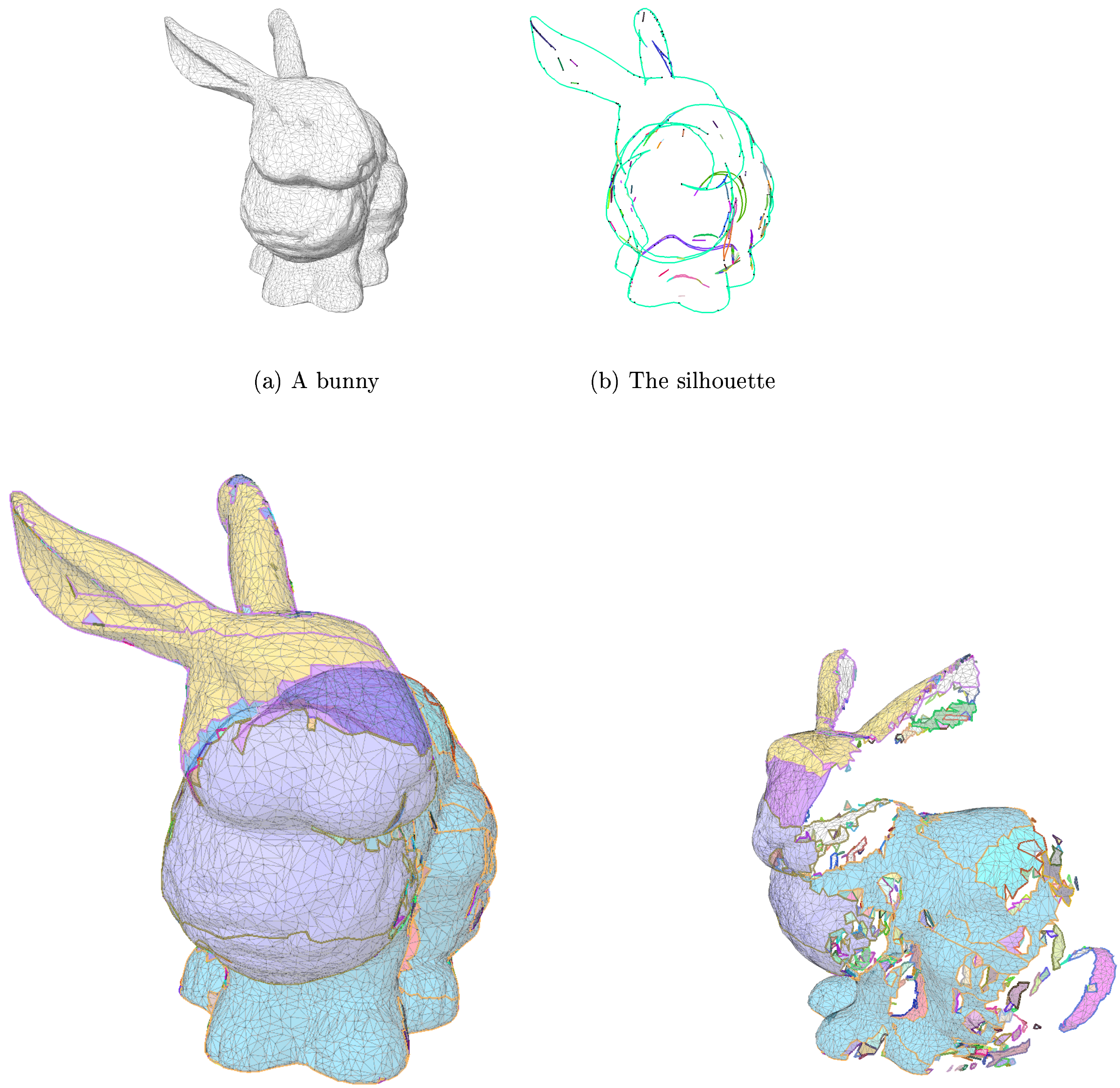
(a) A 3-holed torus

(b) The silhouette

(c) The VLD

(d) A side view of the VLD

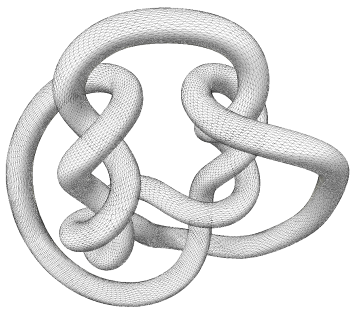
Figure 4.1: The VLD of the 3-holed torus. Notice that the self-occluding sections in the "holes" of the torus have been cut out. On the left and right "holes", the top part of the surface was cut out, and on the middle "hole", the bottom part of the surface was cut out. The cuts on the torus going across the surface were made in the discifying phase. The many small partitions around the concave boundaries were caused by cusps (to be discussed later).



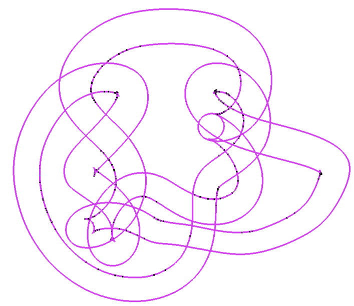
(a) A bunny (b) The silhouette

(c) The VLD (d) A side view of the VLD

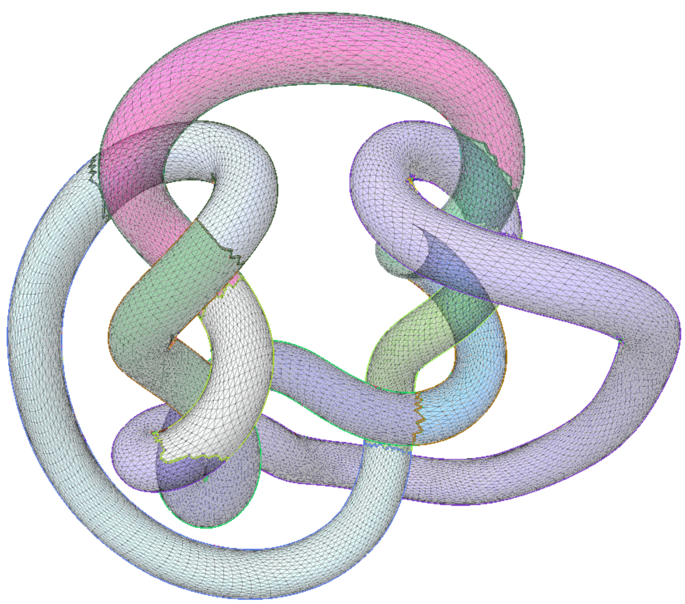
Figure 4.2: The VLD of the bunny. Notice that the bunny is decomposed into a small number of large layers, corresponding to the top of the head, the torso, and the back. On the side view, notice the general irregularity of the decomposition on areas near the silhouette; in most models the silhouette is generally not well-behaved (to be discussed later).



(a) A complex knot



(b) The silhouette



(c) The VLD

Figure 4.3: The VLD of a knot. Notice that the cuts occur on approximate visibility curves. Also notice that, despite the high depth and topological complexity of this model, the number of subsurfaces induced by the VLD is relatively small.

For all steps except cycle removal, our system attains nearly interactive frame rates for complex models. For the bunny (20000 faces) the frame rate averages 2.3 frames per second. For the knot model (46464 faces), the frame rate averages 1.1 frames per second. For the 3-holed torus model (4000 faces), our system averages 12.2 frames per second. These timing numbers are without multi-surface occlusion cycle breaking; including that portion of the code increases running time tenfold, because of our inefficient implementation. If an efficient system is used for finding occlusion cycles, however, we expect that the running time per frame would no more than double over the time to resolve self-occlusion.

Afters implementation is complete, we plan on incorporating this system in a full layer-based rendering system, such as the one presented in [4].

4.2 Future Work

There are many avenues of future work to pursue from this research.

4.2.1 Silhouette Simplification

The silhouettes of an object form our initial partition. For triangle meshes, we have noticed that this initial partition and the resultant surfaces can be extraordinarily complex even for simple models. On "bumpy" models, this complexity can manifest as many tiny surfaces near the area of the silhouette, as we saw on the tilted view of the bunny model.

Even for smooth models, areas of high concavity can contain many cusps as artifacts of the polygonization (as seen in the 3-holed torus model). Each cusp requires the creation of a new layer to remove the cusp. These complexities are not "fundamental" to the visibility decomposition of the basic shape of the model; they are for the most part polygonization artifacts. Silhouette simplification has been investigated in non-photorealistic rendering contexts [2],[6]. These approaches approximate the silhouette of a polygonal model with a series of smooth curves that capture the overall structure of the silhouette. If such a scheme could be employed as a prepass to remove these artifacts, then the savings in running time would be significant. However, it is not obvious how to run the algorithm using the modified silhouette while still respecting the geometry and/or connectivity of the original mesh.

4.2.2 Towards Optimal Partitioning

Our algorithm is by nature greedy. For self-occlusion, we identify an occlusion pair and attempt to resolve it, and for occlusion cycles we find a locally minimal cycle and attempt to break it. Firstly, we notice that even within our framework we can do much better. For self-occlusion, if we identify *all* occlusion pairs and then choose a cut that separates the maximal number of occlusion pairs, the cut we make will generally be more effective in removing occlusion. The same idea can be applied to multi-surface cycles. If we identify all cycles first, then we can choose a link that participates in the most cycles. Cutting that link will generally break more cycles than choosing a link at random.

In general, there are many different methods of choosing cuts which tradeoff quality of cut for computation time per cut. The key feature of our analysis is that it can lead to many such algorithms, differing in how much computation goes into deciding cuts. We believe that different types of surfaces, depending on the low-level algorithms used, will have much different performance characteristics regarding the key parts of the analysis/quality tradeoff:

- cost of finding and analyzing occlusion pairs/cycles
- cost of cutting and partitioning
- cost per resultant subsurface

For instance, cutting a triangle mesh along edges present in the mesh is very fast, and thus our running time is dominated by finding and analyzing occlusion pairs and cycles. As such, we make cuts after analyzing only one occlusion pair, because analyzing all occlusion pairs to save just several cuts would greatly increase running time.

These insights lead us to a question of optimality. The above modifications to the algorithm let us choose between varying degrees of quality in the cuts that we make and cost of computation of the cuts. However, this will not neccessarily lead us to an algorithm that finds the minimal partition of the surface. We suspect that an algorithm for computing a minimal partition (measured either by number of cuts or number of induced subsurfaces) is NP-hard.

4.3 Conclusion

In this thesis, we presented an algorithm for partitioning a surface into a set of visibility-independent regions based on a novel characterization of occlusion on surfaces. This algorithm is general and thus our results can be applied to any surface representation. We have also presented a system that implements this algorithm for triangle meshes that attains near interactive framerates for complex models.

Acknowledgements

I would first of all like to thank Shlomo, who has taught me so much and given me so many opportunities in my time here; John for being so amazingly helpful in getting this thing out the door; Pedro, for coming up with some great ideas through the course of this whole thing.

I would like to thank my readers, for taking the time to read this. It makes it all worthwhile.

I'd also like to thank my parents and sisters, without whom nothing would be possible (even in the non-trivial sense). Just so you know, this monstrosity is the reason why I haven't called you in months...

I must thank my roommates, Adam and Alex, and those other people, Liesje, Emily, and Gwyneth, for being all around good people; Christian, for being The C; DDR for providing excercise-like physical activity; and all those great of mine friends that have tolerated me in these past few months.

And last but not least, Christina, without whom (1) I would have spelled "visibility" wrong every time it appeared in this thesis, and (2) I would have gone insane long, long ago...

Chapter 5

Appendix

5.1 Occlusion Cycle Proofs

Definition An occlusion cycle C consisting of surfaces $s_1, s_2, \cdots, s_k$ is *locally minimal* if there exists no cycle that contains only surfaces in c and has size smaller than $|c|$.

Lemma 5.1.1 *In a locally minimal cycle C ordered like so:*

$$s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n \rightarrow s_1$$

every s_k can only occlude s_{k+1} (s_n occludes s_1).

Proof Say in cycle C with surfaces $s_1, s_2, \cdots, s_n$, s_k occludes s_j where $j \neq k+1$. Say $j > k+1$. Then, the cycle

$$s_1 \rightarrow \cdots s_k \rightarrow s_j \rightarrow \cdots s_n \rightarrow s_1$$

exists, which has length $n - (j - k - 1)$, which, if $j > k+1$, is less than n, so C was not a locally minimal cycle.

If $j < k+1$ then the cycle

$$s_j \rightarrow s_k \rightarrow s_j$$

exists, which has size $k - j + 1$, which must be less than n, so C was not a locally minimal cycle. The lemma is true by contradiction. ∎

Lemma 5.1.1 implies that for every s_k in a locally minimal cycle C, $\iota_{s_k}^{s_{k-1},s_k}$ and $\iota_{s_k}^{s_k,s_{k+1}}$ are the only occlusion regions on s_k in occlusion sets with other surfaces in C. We call $\iota_{s_k}^{s_{k-1},s_k}$ the *under regions* on s_k in C and $\iota_{s_k}^{s_k,s_{k+1}}$ the *over regions* on s_k in C.

The import of lemma 5.1.1 is that in a locally minimal cycle, at a surface s_k we then need only concern ourselves with how s_k interacts with its immediate neighbors s_{k-1} and s_{k+1}. This simplifies our analysis considerably.

Lemma 5.1.2 *If C is a locally minimal cycle, then for every s_k, there can be no point p that is both in an over region of s_k in C, and in an under region of s_k in C.*

Proof Say C is a locally minimal cycle of size n, and that there exists a point p is in some region in $\iota_{s_k}^{s_{k-1},s_k}$ and also in some other region in $\iota_{s_k}^{s_k,s_{k+1}}$. Then, there exists a point p_{k-1} on s_{k-1} that occludes p and point p_{k_1} on s_{k+1} that is occluded by p. We know then that p_{k-1} occludes p_{k+1}, which implies that s_{k-1} occludes s_{k+1}. Thus, the cycle

$$s_1 \rightarrow \cdots \rightarrow s_{k-1} \rightarrow s_{k+1} \rightarrow \cdots s_n \rightarrow s_1$$

exists, which is of size $n-1$, which means that C is not a locally minimal cycle. By contradiction, the lemma is true. ∎

Lemma 5.1.2 implies that union of the over regions on any surface in a minimal cycle must be disjoint from the union of the under regions. Because of this, it is always possible to construct a partition of any surface in the cycle that separates the over regions from the under regions. We call such a partition a *cycle dividing partition.*

Lemma 5.1.3 *If some s_k in a locally minimal cycle C is cycle dividing-partitioned into a set of surfaces S_k, there exists no cycle containing only surfaces in C and surfaces in S_k.*

Proof Say a smaller such cycle exists after s_k is partitioned into S_k. Either the cycle contains a surface in S_k or it does not.

If the cycle does not contain a surface in S_k, then it must not have contained s_k. Thus, there must have been a cycle containing only surfaces in C that did not contain s_k; this means C was not locally minimal.

If the cycle does contain some surface in $s' \in S_k$, then the link $s_i \rightarrow s' \rightarrow s_j$ must exist in the cycle, where s_i and s_j are in c. It cannot be the case that $i = k-1$ and $j = k+1$, because then s' must contain a point under s_{k-1} and a point over s_{k+1}. In that case s_k was not dividing-partitioned. But, this means that s' must either be occluded by some surface that is not $s_{k=1}$, or occlude some surface that is not s_{k+1}. By Lemma 5.1.1 this cannot be possible. By contradiction, there can be no cycle after the partition. ∎

Lemma 5.1.3 suggests a method for breaking any locally minimal cycle; simply pick some surface s in the cycle, identify the under and over regions on s, and partition s so that the under and over regions are separated (lemma 5.1.2 assures that this is always possible).

In this way, we can remove all cycles; after removing a locally minimal cycle C, the surfaces in C can no longer contain a cycle. Thus, by identifying minimal cycles and cutting them until no cycles remain, we can remove all cycles in a set of non-self-occluding surfaces, and this process is guaranteed to terminate, because each time a surface is partitioned the number of surfaces that can participate in a minimal cycle is reduced.

5.2 Self-Occlusion Proofs

Lemma 5.2.1 *If point c is in a occlusion region r in some occlusion set S, then there exists a generator point set that generates S such that c generates r.*

Proof We will construct a point set $(c_1, c_2, \cdots, c_k)$ such that:

- The c_i all project to the same point on the image plane
- The c_i are unique
- We can construct a path from each c_i to each point in the occlusion region r_i

This will show the existence of a generator point set that satisfies the lemma.

Say some k-occlusion set $S = (r_1, r_2, \cdots, r_k)$ is generated by $(p_1, p_2, \cdots, p_k)$. Say c is in r_i. There exists a path d from p_i to c such that for every p_j there exists a full preimage of $\phi(d)$ containing p_j (by definition of the k-occlusion region). Say that the path that is the full preimage of $\phi(d)$ containing p_j is d_j. Call c_j the endpoint of d_j. It must be the case that for every i, j $\phi(c_i) = \phi(c_j)$ (because $\phi(d_i) = \phi(d_j)$, their endpoints must project to the same place). Thus, we know that $\phi(c_1) = \phi(c_2) = \cdots = \phi(c_k)$.

The c_i must be unique. If there was some $c_i = c_j$, $i \neq j$, then there would be a path from p_i to c_i (call it e_i), and a path from p_j to c_i (call it e_j) that both project to the same path. Because these are two distinct paths, this means that locally around c_i there is self-occlusion, which cannot happen because c_i is a point on the interior of a surface, so $c_i \neq c_j$ for $i \neq j$.

We have shown that the c_i all project to the same point, and that they are all unique. We must now show that the regions they define are in fact $(r_1, r_2, \cdots, r_k)$. Consider some point $q_i \in r_i$. There exists a path from d_i from q_i to p_i such that a full preimage of $\phi(d_i)$ containing p_j exists for every p_j. There is also a path d'_i from p_i to c_i such that a full preimage of $\phi(d'_i)$ containing p_j exists for every p_j. Then, we can always construct a path e_i that is the union of d_i and d'_i that goes from c_i to p_i such that the a preimage of $\phi(e)$ containing c_j exists for every c_j. Call this path e_j. Because $d_i \neq d_j$ and $d'_i \neq d'_j$ when $i \neq j$, it must be the case that $e_i \neq e_j$ when $i \neq j$. The mapping M_{e_i, e_j} is simply the obvious concatenation of mapping M_{d_i, d_j} and $M_{d'_i, d'_j}$. ∎

Lemma 5.2.1 defines a function from a point on one region r in a k-occlusion set S to the set of points P that generates that occlusion region. We notate this as:

$$Generators(c, r, S) = P$$

Because of this lemma, we can choose any point in an occlusion region to generate that region in some generator set that generates the occlusion set.

Lemma 5.2.2 *Say a k-occlusion set $S = (r_1, r_2, \cdots, r_k)$ is generated by $(p_1, p_2, \cdots, p_k)$, where p_i generates r_i. If there exists some other generator set $(c_1, c_2, \cdots, c_k)$, such that $c_i = p_i$, then for all j, $c_j = p_j$.*

Proof We will prove this by contradiction.

Assume that $c_i = p_i$ but that some other generator $p_j \neq c_j$. Consider any point x_i in the region r_i. There exists a path d_i from p_i to x_i such that a projection of that path touching p_j ends at a point x_j in r_j. Call the path touching p_j d_j. Because c_j also generates r_j, in the same generator set as p_i (because $p_i = c_i$), there exists a point x'_j that is the endpoint of the similar path that contains c_j. Call that path d'_j.

We will now show that the depth ordering between x_j and x'_j must be the same as the depth ordering between p_j and c_j.

Assume without loss of generality that p_j occludes c_j. We can construct a smooth mapping $C = cM_{d_j,d'_j} = M_{d_i,d'_j} \circ M_{d_j,d_i}$ from points on d_j to corresponding points on d'_j. This mapping is one-to-one, and for all $q \in d_j$, $C(q) \neq q$ and $\phi(C(q)) = \phi(q)$. Moreover, $C(p_j) = c_j$ and $C(x_j) = x'_j$.

Consider the function $\delta(q) = z(q) - z(C(q))$ for $q \in d_j$. Assume without loss of generality that p_j occludes c_j. Then, $\delta(p_j) > 0$. If x'_j occludes x_j, then $\delta(x_j) < 0$. δ must be continuous along e_j, so this implies that at some q, $\delta(q) = 0$. This implies that $z(q) = z(C(q))$. Because $\phi(q) = \phi(C(q))$, but q and $C(q)$ are distinct points on the interior of the surface, this implies that the surface self-intersects at q. This is impossible by assumption. Thus, the depth ordering between x_j and x'_j must be the same as the depth ordering between p_j and c_j.

Thus, for every $x_j \in r_j$, there exists a different point $x'_j \in r_j$, such that x_j occludes x'_j. This is clearly impossible for finite surfaces, so, by contradiction, $p_j = c_j$. ∎

Lemma 5.2.3 *Say a k-occlusion set S is generated by $(p_0, p_1, \cdots, p_k)$ and also by some other set $(c_1, c_2, \cdots, c_k)$, such that if r_i is generated by p_i then r_i is generated by c_i. If p_i occludes p_j then c_i occludes c_j.*

Proof Consider the construction offered in the proof of lemma 5.2.1. For some generator set $(p_1, p_2, \cdots, p_k)$ of $S = (r_1, r_2, \cdots, r_k)$ and a point $c_i \in r_i$, we built an occlusion set $(c_1, c_2, \cdots, c_k)$ that generates S. In the construction, we built a path e such that for each p_j the preimage of $\phi(e)$ containing c_j (called e_j) connected c_j and p_j (that in fact was the basis for the construction). For every i and j, $i \neq j$, we know that $\phi(e_i) = \phi(e_j)$.

Because of lemma 5.2.2, we know that the generator set constructed as above must be the sole generator set where c_i generates r_i. Thus, given two generator sets $(p_1, p_2, \cdots, p_k)$ and $(c_1, c_2, \cdots, c_k)$, simply construct the generator set of S where c_1 generates r_1. This is guaranteed to construct $(c_1, c_2, \cdots, c_k)$, and the results from the construction can be applied.

We will now show that if p_i occludes p_j, then c_i occludes c_j. This will complete the proof. Our proof method will be similar to that used in proof of lemma 5.2.2.

Assume without loss of generality that p_i occludes p_j. We know there exists a mapping $C = M_{e_i,e_j}$ from points on e_i to corresponding points on e_j (where e_i and e_j are defined as above). This mapping is one-to-one, and for all $q \in e_i$, $C(q) \neq q$ and $\phi(C(q)) = \phi(q)$. Moreover, $C(p_i) = p_j$ and $C(c_i) = c_j$.

Consider the function $\delta(q) = z(q) - z(C(q))$ for $q \in e_i$. Assume without loss of generality that p_i occludes p_j. Then, $\delta(p_i) > 0$. If c_j occludes c_i, then $\delta(c_i) < 0$. δ must be continuous along e_i, so this implies that at some q, $\delta(q) = 0$. This implies that $z(q) = z(C(q))$. We know that $\phi(q) = \phi(C(q))$, but q and $C(q)$ are distinct points on the interior of the surface, this implies that the surface self-intersects at q. This is impossible by assumption.

Thus, it must be the case that c_i occludes c_j. ∎

The previous lemma implies that the depth ordering of every generator set for a given k-occlusion set is the same. This is the results which lets us reason about how occlusion sets behave. Notice that this means that between occlusion regions in a set there is a natural conception of which occlusion set is "above" which others (even though they might overlap).

Lemma 5.2.4 *Consider the region in the image plane of $\phi(r_1)$, where r_1 is some region in an occlusion pair $S = (r_1, r_2)$. For every point $c \in \phi(r_1)$ there must exist a point p_1 in r_1 but not in r_2, and a point p_2 in r_2 but not in r_1, and $\phi(p_1) = \phi(p_2) = \phi(c)$.*

Proof We will prove this by contradiction.

By lemma 5.2.3, we know that for every generator set (g_1, g_2) of S, either g_1 is above g_2 or vice versa. Without loss of generality, say that g_1 is always above g_2.

First we will prove there must be a p_1 in r_1 but not in r_2. Consider the set of points P that are the preimages of c on r_1 and r_2. Take the topmost point of P and call it p_a. If p_a is in r_1 only, then we are done; p_a is the point that is in r_1 only.

If p_a is in r_2 only, or in r_1 and r_2, then we have a contradiction. Consider the point pair $G_a = Generators(p_a, r_2, S)$. In G_a, p_a must generate r_2. This implies the existence of a p'_a that generates r_1 in G_a. We know that $p'_a \in r_a$ must be above p_a by lemma 5.2.3. This is a contradiction, because p_a is the highest point that projects to $\phi(c)$.

Now we will prove there must be a p_2 in r_2 but not in r_1. Consider the set of points P that are the preimages of c on r_1 and r_2. Take the bottommost point of P and call it p_b. If p_b is in r_2 only, then we are done; p_b is the point that is in r_b only.

If p_b is in r_1 only, or in r_2 and r_1, then we have a contradiction. Consider the point pair $G_b = Generators(p_b, r_1, S)$. In G_b, p_b must generate r_1. This implies the existence of a p'_b that generates r_2 in G_b. We know that $p'_b \in r_b$ must be below p_b by lemma 5.2.3. This is a contradiction, because p_b is the lowest point that projects to $\phi(c)$. ∎

Bibliography

[1] Catmull, E. A subdivision algorithm for computer display of curved surfaces. Tech. Rep. UTEC-CSc-74-133, Salt Lake City, 1974.

[2] Hertzmann, A., and Zorin, D. Illustrating smooth surfaces. In *Siggraph 2000, Computer Graphics Proceedings* (2000), K. Akeley, Ed., ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 517–526.

[3] Krishnan, S., and Manocha, D. Global visibility and hidden surface algorithms for free form surfaces, 1994.

[4] Lengyel, J., and Snyder, J. Rendering with coherent layers. *Computer Graphics 31*, Annual Conference Series (1997), 233–242.

[5] Newell, M. E., Newell, R. G., and Sancha, T. L. A solution to the hidden surface problem. In *Proceedings of the ACM Annual Conference* (Boston, Massachusetts, 1972), vol. I, pp. 443–450.

[6] Northrup, J., and Markosian, L. Artistic silhouettes: A hybrid approach, 2000.

[7] Sander, P. V., Gu, X., Gortler, S. J., Hoppe, H., and Snyder, J. Silhouette clipping. In *Siggraph 2000, Computer Graphics Proceedings* (2000), K. Akeley, Ed., ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 327–334.

[8] Snyder, J., and Lengyel, J. Visibility sorting and compositing without splitting for image layer decomposition. *SIGGRAPH '98* (July 1998), 219–230.

[9] Struik, D. Lectures on classical differential geometry, 1961.

[10] Xia, F. Normal vector and winding number in 2d digital images with application for hole detection.