Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines

Aaron Hertzmann
Media Research Laboratory
Department of Computer Science
New York University
http://www.mrl.nyu.edu/hertzmann/

1 Introduction

In these notes, we survey some of the basic tools used for non-photorealistic rendering of 3D scenes. We will primarily focus on detecting outlines of object shape: silhouettes, boundaries, and creases. (Hatching and shading, which are also very important for communicating shape, are discussed elsewhere in the course notes.) The algorithms in this section can be divided into algorithms that operate in the image space (2D), and algorithms that operate in world space (3D).

2 Outlines in Image Space

The easiest way to detect silhouettes is to let your existing graphics packages do all the hard work. By rendering images in different ways and then post-processing, you can quickly produce pleasing results.

Image-space non-photorealistic rendering algorithms use rendered (and, thus, sampled) images, and are limited by the precision of the images. However, image space silhouette detection is sufficient for many applications. Exact algorithms that operate in 3D are described in Section 3.

Some of the simplest silhouette algorithms were introduced by Gooch et al. [12], and are described in Chapter 10 of the course notes. A related method by Raskar and Cohen [17] allows variable-length line segments; see their paper for more details.

2.1 Outlines with Edge Detection

A very simple way to generate a line drawing of a 3D scene would be to render the scene from the desired point of view, detect edges in the image, and display the edges. However, the edges of a

photograph do not typically correspond to the silhouette edges that we want to illustrate [19]. For instance, highly textured surfaces will generate many edges that are irrelevant to the object shape; likewise, no edge is detected between two overlapping objects with the same color.

2.1.1 Depth Map

A better way to generate silhouettes is to render the image, extract the depth map, and apply an edge detector to the depth map [18, 6, 5] (Figure 1(a,b)). The depth map is an image where the intensity of a pixel is proportional to the depth of that point in the scene. The idea behind this method is that the variation in depth between adjacent pixels is usually small over a single object, but large between objects. Edge detection is described in Appendix A. Most graphics packages provide some way of extracting the depth map from an image[1].

2.1.2 Normal Map

A problem with this method is that it does not detect the boundaries between objects that are at the same depth, nor does it detect creases. (In more formal terms, it can only detect C^0 surface discontinuities.) We can augment the silhouette edges computed with the depth map by using surface normals as well.

We will do this by using a normal map, which is an image that represents the surface normal at each point on an object. The values in each of the (R, G, B) color components of a point on the normal map correspond to the (x, y, z) surface normal at that point.

To compute the normal map for an object with a graphics package, we can use the following procedure [6]. First, we set the object color to white, and the material property to diffuse reflection. We then place a red light on the X axis, a green light on the Y axis, and a blue light on the Z axis, all facing the object. Additionally, we put lights with negative intensity on the opposite side of each axis. We then render the scene to produce the normal map. Each light will illuminate a point on the object in proportion to the dot product of the surface normal with the light's axis. (If negative lights are not available, then two rendering passes will be required.) An example is shown in Figure 1(c,d).

We can then detect edges in the normal map. These edges detect changes in surface orientation, and can be combined with the edges of the depth map to produce a reasonably good silhouette image (Figure 1(e)). These methods together detect C^0 and C^1 discontinuities in the image.

[1] In OpenGL, for example, the depth map can be extracted by calling **glReadPixels** with the GL_DEPTH_COMPONENT argument.

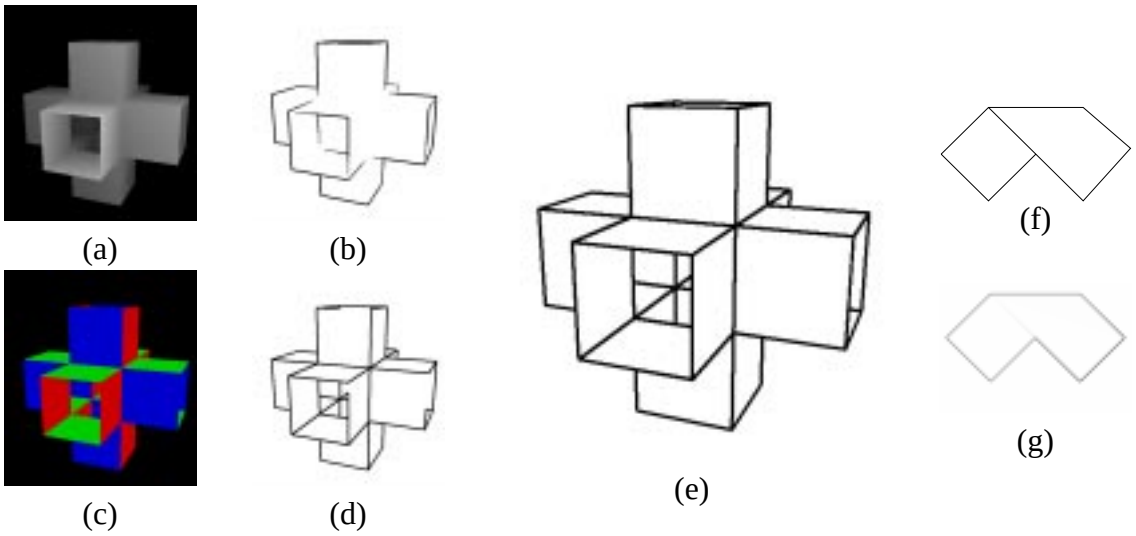


Figure 1: Outline drawing with image processing. (a) Depth map. (b) Edges of the depth map. (c) Normal map. (d) Edges of the normal map. (e) The combined edge images. (f) A difficult case: folded piece of paper (g) Depth edges. (See also the Color Plates section of the course notes.)

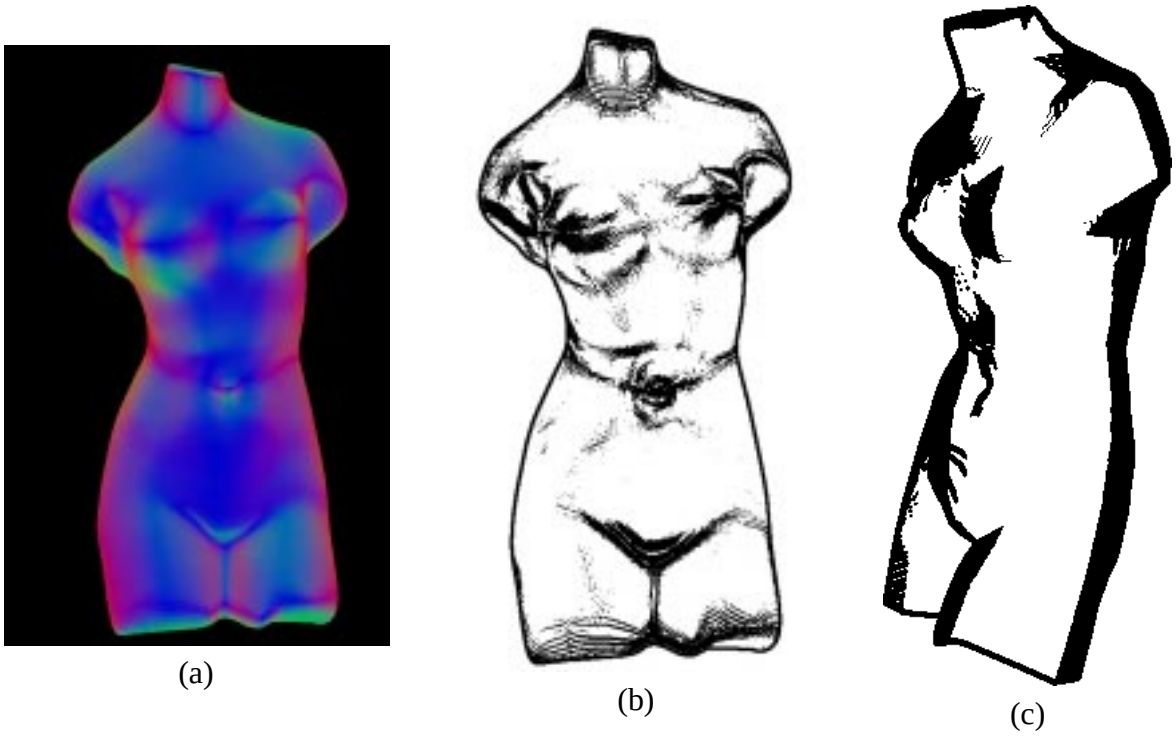
2.1.3 Other Types of Textures

We can generalize the above methods to render the image with any smoothly-varying surface function, and then detect edges in the resulting image. For example, we can texture-map the object with a smooth image. However, special care must be taken if the texture map is repeated on the surface [4]. Different methods, such as environment map textures and volumetric textures, can be explored for locating different classes of object lines.

2.2 Rendering

If we intend simply to render the image as dark lines, then these edge images are sufficient. If we wish to render the silhouettes as curves with some kind of attributes, such as paint texture or varying thickness, or if we wish to modify the line qualities, then we must somehow extract curves from the edge map. Methods for doing this are described by Curtis [5] and Corrêa et al. [4]. Saito and Takahashi [18] also discuss some heuristics for modifying the appearance of the edge images.

For many applications, the techniques described in this section are sufficient. However, they do suffer the fundamental limitation that important information about the 3D scene is discarded during rendering, and this information cannot be reconstructed from the 2D image alone. For example, a highly foreshortened surface will appear to be discontinuous. This means that you must manually tweak the image processing parameters for each model and scene. Furthermore, there



(a) (b) (c)

Figure 2: Serendipitous effects using edge detection. (a) Normal map of a subdivided Venus model. (b) Woodcut-style image generated from edges of the normal map. Sharp edges in the mesh generate hatch marks. (c) A sumi-e style image. Streak lines are due to sampling in the depth buffer. (See also the Color Plates section of the course notes.)

are fundametal limitations; Figure 1(f) shows a pathologically difficult case for these algorithms: none of the surface functionals or textures we have described will locate the corner of fold. In the next section, we describe algorithms that compute outlines precisely.

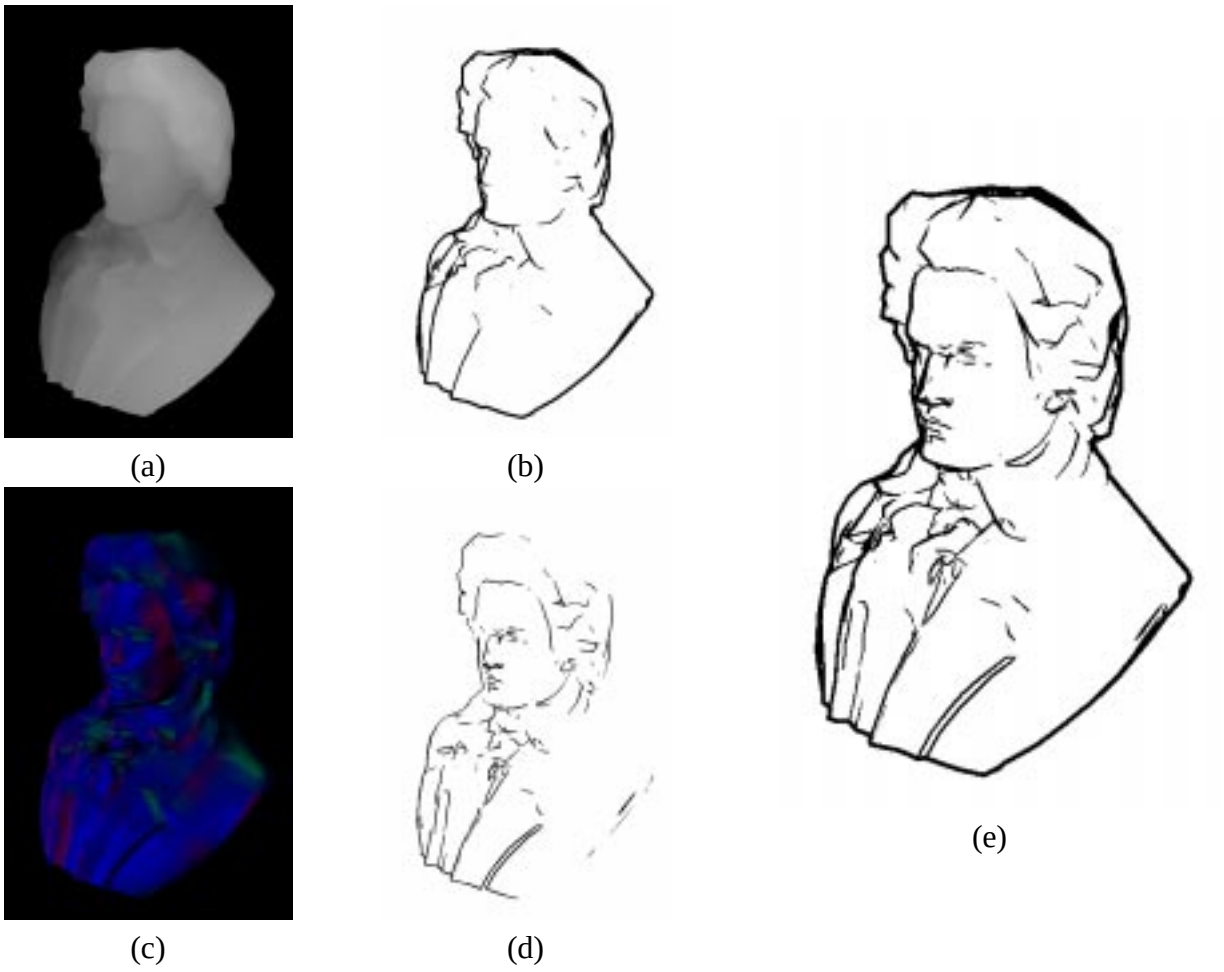


Figure 3: Outline detection of a more complex model. (a) Depth map. (b) Depth map edges. (c) Normal map. (d) Normal map edges. (e) Combined depth and normal map edges. (See also the Color Plates section of the course notes.)

3 Object Space Silhouette Detection

In this section, we will describe methods for finding silhouettes of models in 3D. These methods are more involved than the image space methods, but can produce curves with much higher precision. These curves are also suitable for additional processing; for example, they can be rendered with natural brush strokes.

There are several types of curves we are interested in:

- Silhouettes
- Surface boundaries
- Creases. A crease is a discontinuity on an otherwise smooth surface.
- Self-intersections. Some surfaces that arise in mathematics intersect themselves. Such surfaces are seldom used for typical graphics applications.
- Other surface curves, such as isoparametric curves.

We will primarily discuss detection of silhouettes. Detecting boundaries and creases is straightforward, and can be done in advance for each model. For discussion of isoparametric lines and other kinds of hatch marks, see [21, 7].

3.1 What is a Silhouette?

Before we proceed, we need to make formal the definition of silhouette. For polygonal meshes, the silhouette consists of all edges that connect back-facing (invisible) polygons to front-facing (possibly visible) polygons. For a smooth surface, the silhouette can be defined as those surface points $\mathbf{x}_i$ with a surface normal $\mathbf{n}_i$ perpendicular to the view vector (Figure 4):

$$\mathbf{n}_i \cdot (\mathbf{x}_i - \mathbf{C}) = 0 \tag{1}$$

where $\mathbf{C}$ is the camera center. Note that this definition makes no mention of visibility; a point that is occluded by another object may still be a silhouette point. In almost every case, we are only interested in rendering the visible segments of silhouette curves, or in rendering the invisible sections with a different line quality. Hidden line elimination will be discussed in Section 3.4.

In this discussion, we will focus on perspective projection. For orthogonal projection, all view vectors are parallel, and the equation can be written as $\mathbf{n}_i \cdot \mathbf{v} = 0$, where $\mathbf{v}$ is the view vector.

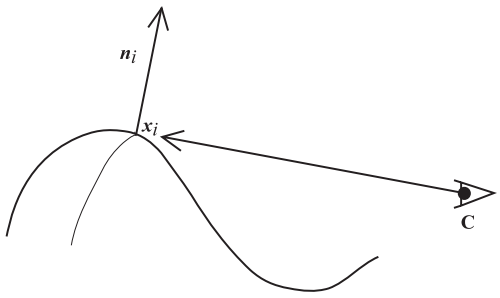


Figure 4: For smooth surfaces, the silhouette is the set of points for which the surface normal is perpendicular to the view vector.

3.2 Polygonal Meshes

For a polygonal mesh, the silhouettes are exactly the set of mesh edges that connect a front-facing polygon and a back-facing polygon. The usual method for computing the silhouette is to iterate over every mesh edge and check the normals of the adjacent faces. This loop must be performed every time the camera position changes.

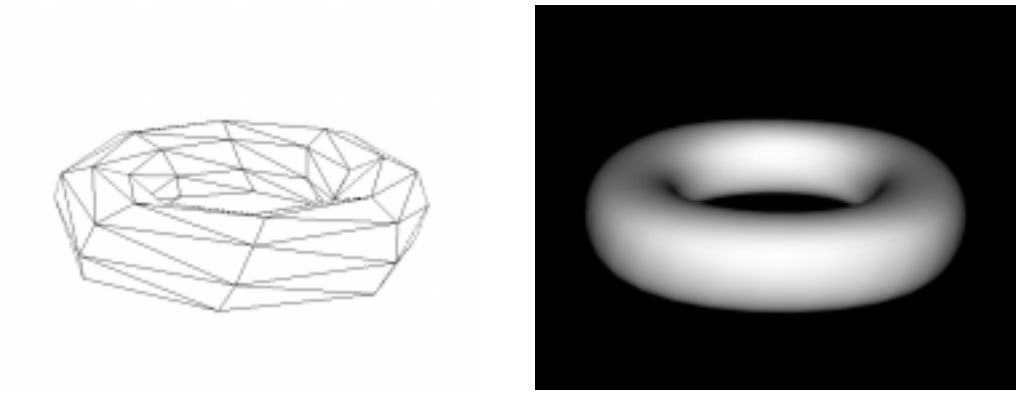
Testing every mesh edge can be quite expensive. We can speed this up by testing only a few of the edges. Markosian et al. [15] describe a randomized algorithm that can find the majority of visible silhouette edges at interactive rates. Their algorithm is based on the observation that only a small fraction of mesh edges are visible silhouette edges [13]. Thus, we can randomly select a few edges for testing, rather than testing the entire mesh. Please see their paper for details about which edges to test. This algorithm is not guaranteed to find every silhouette edge, but it will find most of them.

For orthographic projection, it is possible to efficiently locate silhouettes using a Gaussian map [12, 2]. See Chapter 10 of the course notes for details.

3.3 Smooth Surfaces

In this section, we describe methods for computing silhouettes of smooth surfaces [11, 8]. It is often desirable to generate pictures of surfaces that are smooth, rather than just polyhedral. Two of the most popular surface representations are NURBS [10] and subdivision surfaces [20]. Both of these representations are based on polyhedral meshes; the actual smooth surface approximates (or interpolates) a mesh. Furthermore, it is possible to compute the surface normal and principle curvatures at any surface point[2]. Our discussion here will focus on these surface representations. For discussion of implicit surfaces, see [3].

[2]Non-stationary subdivison surfaces are an exception. Little is known about the limit behavior of these surfaces.



(a) (b)

Figure 5: (a) A polygonal mesh (b) A smooth surface defined by the mesh

For simplicity, we will only discuss surfaces approximated by a triangular mesh (Figure 5). A quadrilateral mesh can be converted into a triangular mesh by splitting each quadrilateral into two triangles. Alternatively, these ideas can be extended to polygonal meshes; however, many more cases must be handled [11].

The first step is to compute the normalized dot product d_i of the normal $\mathbf{n}_i$ of the smooth surface with the view vector at every mesh vertex:

$$d_i = \frac{\mathbf{n}_i \cdot (\mathbf{x}_i - \mathbf{C})}{||\mathbf{n}_i||||\mathbf{x}_i - \mathbf{C}||} \tag{2}$$

We then compute the sign of the dot product for each vertex:[3]

$$s_i = \begin{cases} +, & d_i \geq 0 \\ -, & d_i < 0 \end{cases} \tag{3}$$

Recall that our goal is to locate surface points with $d_i = 0$. Because these quantities vary smoothly over the surface, we can simply look at all pairs of adjacent vertices that have different signs. Given a mesh edge between points $\mathbf{x}_i$ and $\mathbf{x}_j$, if $s_i \neq s_j$, then there must be a silhouette point on the edge. (This can be shown by noting that the surface normal and the view vector are both continuous over the surface; therefore, their dot product must also be continuous. There must exist a zero-crossing between a positive and a negative value of a continuous function.) We can approximate the position of the silhouette point by linearly interpolating the vertecies:

$$\mathbf{x}' = \frac{|d_j|}{|d_i| + |d_j|}\mathbf{x}_i + \frac{|d_i|}{|d_i| + |d_j|}\mathbf{x}_j \tag{4}$$

[3]By treating 0's as positive, we avoid some complicated bookkeeping later on.

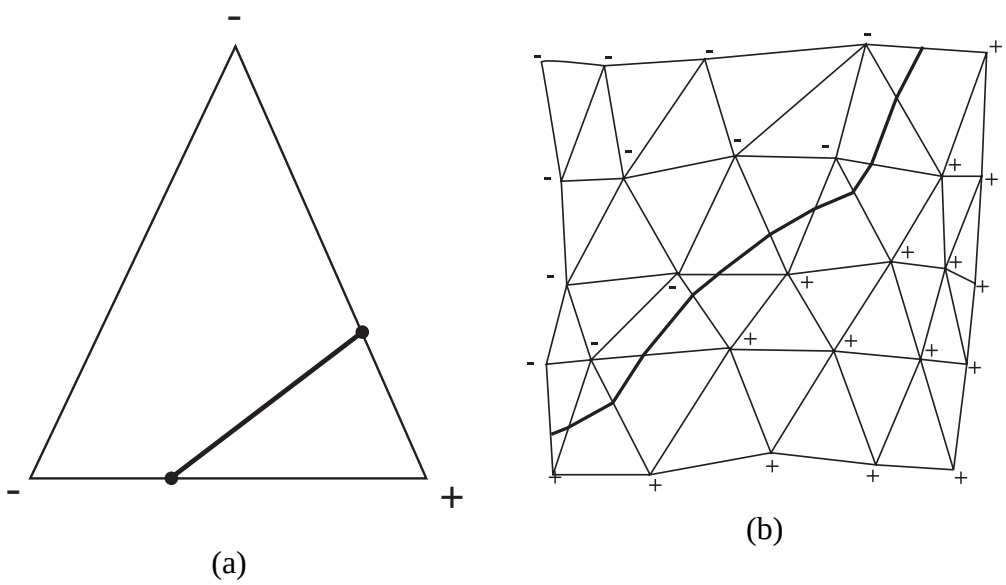
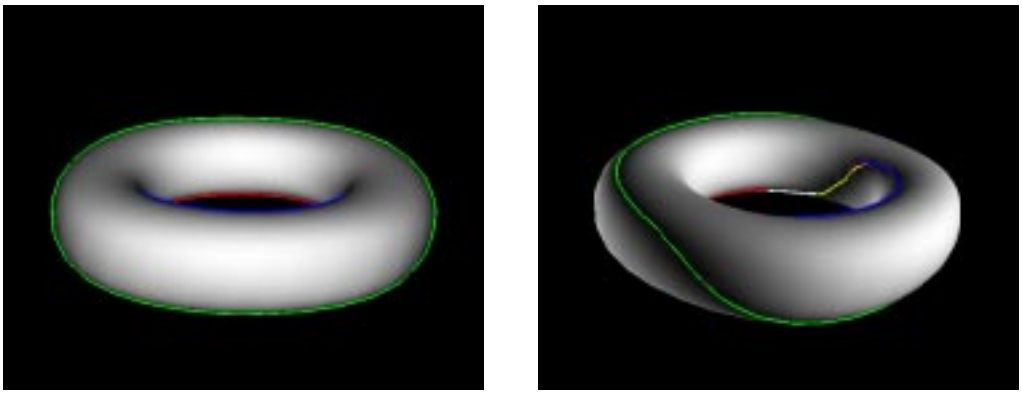


Figure 6: Approximating the silhouette curve for a smooth surface. The smooth surface is represented by a triangle mesh that is close to the surface. (a) Silhouette points are computed on each edge by linear interpolation. A line segment is created by connecting the points. (b) Line segments are connected to form a piecewise-linear curve.

We can then connect the silhouette points into silhouette curves. Suppose a mesh triangle contains sign changes. Since a triangle only contains three vertecies, there is only one unique case (Figure 6(a)): one vertex has an opposite sign from the other two vertecies. In this case, two triangle edges have a sign change, and the other does not. To approximate the silhouette in this triangle, we can simply compute silhouette points on the two edges, and connect them with a line segment. We then repeat this for every triangle with a sign change, and connect all pairs of line segments that share an end point (Figure 6(b)). This produces a piecewise-linear approximation to the silhouettes of the smooth surface.

If the mesh has large triangles, then the silhouette approximation will be very coarse. It is also possible for silhouettes of the smooth surface to be missed entirely by this method. The solution to this is to refine the mesh; in other words, to produce a mesh with smaller triangles that corresponds to the same smooth surface. The specific method of subdivision depends on the type of surface being used. Subdivision can be applied repeatedly, until the desired resolution is reached.

Furthermore, if we are only interested in mesh silhouettes, then we can use adaptive subdivision to refine the surface only near the silhouette. The obvious subdivision criteria is to subdivide every triangle that already contains a silhouette. However, this will miss silhouettes that are contained entirely within a single triangle. A sufficient condition is to subdivide a triangle if there is a sign



(a) (b)

Figure 7: (a) Silhouette of a smooth surface. (b) Side view of the silhouette. Changes in visibility in this figure are due to cusps. (See also the Color Plates section of the course notes.)

change anywhere in the control mesh for a triangle. (Otherwise, the control mesh is 1-1 in the image plane, and cannot produce a silhouette.)

3.4 Visibility

Once we have curves of interest from a surface, we would like to determine which portions of these curves are visible. The visibility process is more or less the same for most types of curves (silhouette, boundary, crease, isoparametric line, etc.) Hidden line visibility is one of the oldest problems in computer graphics [1].

The basic algorithm for computing visibility is to break all curves at potential changes in visibility, producing a new set of curves where each curve is entirely visible or entirely invisible. We then determine the visibility of each new curve by ray tests. [1, 8, 15].

There are three situations where the visibility of a surface curve can change (Figure 8):

1. It passes under a silhouette, boundary or crease in the image plane.
2. It intersects a silhouette, crease, or self-intersection curve on the surface.
3. It is a silhouette or boundary and has a cusp.

Note that these are *potential* changes of visibility; for example, a curve completely hidden by another object will be entirely invisible, regardless of any of these cases.

The first case occurs when part of one object obscures another object. This case can easily be detected by projecting curves onto the image plane, and then computing all curve intersections. The intersections can be performed by the sweep-line algorithm [16] or by scan-converting the

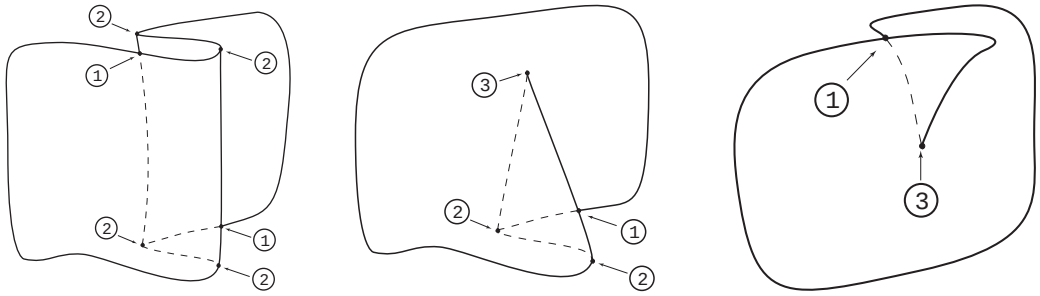


Figure 8: Examples of the three types of potential changes in visibility (See text).

edges of the curve into a grid, and computing intersections for all pairs of edges in each grid cell [15].

The second case is straightforward, and could be detected using the projection method above. However, if one of the curve terminates at another curve, then the intersection might be missed due to numerical precision errors, depending on the representation. It is probably more convenient to detect these cases on the surface.

The third case, a cusp, occurs when the projection of a curve contains a discontinuity. A silhouette or boundary curve may become invisible at a cusp. For polygonal meshes, cusps may only occur at mesh vertecies. A mesh vertex may only be a cusp if it is adjacent to both a front-facing silhouette edge and a back-facing silhouette edge, or if it lies on a boundary. An edge is front-facing if and only if the nearer of the adjacent faces is front-facing. For smooth surfaces, cusps occurs on silhouette curves wherever the projection of the curve has a C^1 discontinuity. Boundaries and creases may only have cusps at tagged corner vertices, i.e. vertices marked as discontinuous.

Once we isolate every potential change in visibility, we can split each curve into smaller curves. We then have a collection of curves, each of which is entirely visible, or entirely invisible. We can then test the visibility of each curve by performing ray tests.

To perform a ray test on a curve, we cast a ray in the direction of the view vector. If the ray intersects any surfaces before the curve, then the curve is invisible; otherwise, it is visible. The intersection test may be accelerated using conventional techniques, such as a BSP tree. For smooth surfaces, the ray test is complicated by the fact that we do not have an explicit surface representation, and so care must be taken when casting rays with respect to an approximating mesh. Kobbelt et al. [14] describe a more elaborate and reliable method for ray tests.

It is possible to avoid many ray tests by taking advantage of visibility coherence on the surface [15, 1, 8]. However, some visibility relationships for curves on smooth surfaces are not the same as for their piecewise-linear approximations, and such approximations should be used with caution.

Once the ray tests are complete, we have determined the visibility of every curve of interest,

and the curves may be rendered.

Acknowledgments

Portions of these notes describe joint work with Denis Zorin.

References

[1] Arthur Appel. The Notion of Quantitative Invisibility and the Machine Rendering of Solids. In *Proc. ACM National Conference*, pages 387–393, 1967.

[2] F. Benichou and Gershon Elber. Output Sensitive Extraction of Silhouettes from Polygonal Geometry. ftp://ftp.cs.technion.ac.il/pub/misc/gershon/papers/sil_extrac.ps.gz.

[3] David J. Bremer and John F. Hughes. Rapid approximate silhouette rendering of implicit surfaces. In *Proc. The Third International Workshop on Implicit Surfaces*, June 1998.

[4] Wagner Toledo Corrêa, Robert J. Jensen, Craig E. Thayer, and Adam Finkelstein. Texture Mapping for Cel Animation. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 435–446. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

[5] Cassidy Curtis. Loose and Sketchy Animation. In *SIGGRAPH 98: Conference Abstracts and Applications*, page 317, 1998.

[6] Philippe Decaudin. Cartoon-Looking Rendering of 3D-Scenes. Technical Report 2919, INRIA, June 1996.

[7] Gershon Elber. Line Art Illustrations of Parametric and Implicit Forms. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), January – March 1998. ISSN 1077-2626.

[8] Gershon Elber and Elaine Cohen. Hidden Curve Removal for Free Form Surfaces. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 95–104, August 1990.

[9] Hany Farid and Eero P. Simoncelli. Optimally Rotation-Equivarient Directional Derivative Kernels. In *7th Int'l Conf Computer Analysis of Images and Patterns*, Kiel, Germany, September 1997.

[10] Gerald Farin. *Curves and Surfaces for Computer-Aided Geometric Design: A Pratical Guide.* Academic Press, Inc., Boston, third edition, 1993.

[11] Amy Gooch. Interactive Non-Photorealistic Technical Illustration. Master's thesis, University of Utah, December 1998.

[12] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Richard Riesenfeld. Interactive Technical Illustration. In *Proc. 1999 ACM Symposium on Interactive 3D Graphics*, April 1999.

[13] Lutz Kettner and Emo Welzl. Contour Edge Analysis for Polyhedron Projections. In W. Strasser, R. Klein, and R. Rau, editors, *Geometric Modeling: Theory and Practice*, pages 379–394. Springer Verlag, 1997.

[14] Leif Kobbelt, K. Daubert, and Hans-Peter Seidel. Ray tracing of subdivision surfaces. In *Eurographics Rendering Workshop '98 Proceedings*, 1998.

[15] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 415–420. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[16] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[17] Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. In *Proc. 1999 ACM Symposium on Interactive 3D Graphics*, April 1999.

[18] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 197–206, August 1990.

[19] T. Sanocki, K. Bowyer, M. Heath, and S. Sarkar. Are real edges sufficient for object recognition? *Journal of Experimental Psychology: Human Perception and Performance*, 24(1):340–349, January 1998.

[20] Peter Schröder and Denis Zorin, editors. *Subdivision for Modeling and Animation*. SIGGRAPH 99 Course Notes, 1999.

[21] Georges Winkenbach and David H. Salesin. Rendering Parametric Surfaces in Pen and Ink. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 469–476. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

A Edge Detection with Convolution

Hundreds of papers have been written about detecting edges in images. Fortunately, the kinds of images we are concerned with here — synthetic depth and normal maps — are amenable to reasonably simple edge detectors. In this section, we describe edge detection with the Sobel filter, and a simple variant.

The Sobel kernels are:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{5}$$

Let $I(x, y)$ be a grayscale image. Vertical and horizontal edge images of I are computed by discrete 2D convolution: $I_x(x, y) = I(x, y) \otimes S_x$, $I_y(x, y) = I(x, y) \otimes S_y$. (This operation is an approximation to differentiation of the image in the continuous domain.) Finally, to create an edge image, we compute the magnitude of the derivative, and then theshold the edges by some threshold T:

$$I_{mag}(x, y) = \sqrt{I_x^2(x, y) + I_y^2(x, y)} \tag{6}$$

$$Edge(x, y) = \begin{cases} 1 & I_{mag}(x, y) \geq T \\ 0 & I_{mag}(x, y) < T \end{cases} \tag{7}$$

The Sobel filter sometimes produces noisy results. For better performance, you can replace the S_x and S_y with the following "optimal" 5x5 filters F_x and F_y [9]:

$$\begin{array}{rcl} p_5 & = & [0.036470, 0.248968, 0.429123, 0.248968, 0.036470] \\ d_5 & = & [-0.108385, -0.280349, 0.0, 0.280349, 0.108385] \\ F_x & = & p_5^T d_5 \\ F_y & = & d_5^T p_5 \end{array} \tag{8}$$