Real-Time Nonphotorealistic Rendering

Lee Markosian *Michael A. Kowalski* *Samuel J. Trychin* *Lubomir D. Bourdev*
Daniel Goldstein *John F. Hughes*

Brown University site of the
NSF Science and Technology Center for
Computer Graphics and Scientific Visualization
Providence, RI 02912

Abstract

Nonphotorealistic rendering (NPR) can help make *comprehensible* but simple pictures of complicated objects by employing an economy of line. But current nonphotorealistic rendering is primarily a batch process. This paper presents a real-time nonphotorealistic renderer that deliberately trades accuracy and detail for speed. Our renderer uses a method for determining visible lines and surfaces which is a modification of Appel's hidden-line algorithm, with improvements which are based on the topology of singular maps of a surface into the plane. The method we describe for determining visibility has the potential to be used in any NPR system that requires a description of visible lines or surfaces in the scene. The major contribution of this paper is thus to describe a tool which can significantly improve the performance of these systems. We demonstrate the system with several nonphotorealistic rendering styles, all of which operate on complex models at interactive frame rates.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation — Display algorithms;

Additional Key Words: non-photorealistic rendering

1 Introduction

Computer graphics is concerned with the production of images in order to convey visual information. Historically, research in computer graphics has focused primarily on the problem of producing images which are indistinguishable from photographs. But graphic designers have long understood that photographs are not always the best choice for presenting visual information. A simplified diagram is often preferred when an image is required to delineate and explain. Lansdown and Schofield [10] make this point in the context of a repair manual, asking, "How much use is a photograph to mechanics when they already have the real thing in front of them?" Strothotte et al. [17] note that architects often trace over computer renderings of their initial designs to create a sketchier look, because they want to avoid giving their clients a false impression of completeness. In general, the question of whether to use photorealistic imagery depends on the *visual effect intended by the designer*.

A growing body of research in computer graphics has recognized the power and usefulness of nonphotorealistic imagery [21, 20, 13, 22, 10, 17, 11, 7, 16, 3]. Until now, though, nonphotorealistic rendering (NPR) methods have primarily been batch-oriented rather than interactive. (An exception is Zeleznik's SKETCH system [22], which makes crude nonphotorealistic renderings using tricks in the standard polygon-rendering pipeline). One obstacle to achieving real-time nonphotorealistic rendering is the problem of determining visibility, since a straightforward use of z-buffering may give incorrect results. This can occur when what is drawn on the screen does not correspond literally to the geometry of the scene. For example, a line segment between two vertices of a triangle mesh may be rendered in a wobbly, hand-drawn style. Any part of the wobbly line which does not directly correspond to the original line segment may be clipped out during z-buffering.

This paper presents a new real-time NPR technique based on an *economy of line* – the idea that a great deal of information can be effectively conveyed by very few strokes. Certain key features of images can convey a great deal of information; our algorithm preferentially renders silhouettes, certain user-chosen key features (e.g., creases), and some minimal shading of surface regions. To accomplish this at interactive rates, we rely on approximate data: not every silhouette is rendered in every frame, although large silhouettes are rendered with high probability. The key ideas that support this scheme are

- rapid (probabilistic) identification of silhouette edges,
- using interframe coherence of silhouette edges, and
- fast visibility determination using improvements and simplifications in Appel's hidden-line algorithm [1].

We demonstrate the use of these techniques to support a variety of rendering styles, all of which are produced at interactive rates. These include a spare line-rendering style suitable for illustrations (including optional rendering of hidden lines), a variety of sketchy hand-drawn styles suitable for approximate models, and a technique for adding shading strokes to basic visible-line renderings in order to better convey 3D information while preserving an artistic effect. This last technique uses a method for determining hidden surfaces which is a simple extension of the hidden-line algorithm. Using the methods we describe, our renderer is able to produce basic visible line drawings of free-form (tesselated) surfaces at an effective rate of over 1 million model polygons per second on a modern workstation.

The overall structure of our algorithm is: (i) determine the silhouette curves in the model, (ii) determine the visibility of silhouette and other feature edges by a modified Appel's algorithm, (iii) render the silhouette and feature edges. The basic algorithm can be extended to perform some shading over surface regions, in which case step (ii) is extended to determine visibility of surfaces. We explain the first two parts in detail, and then describe the final part in Section 5.

2 Assumptions and Definitions

First, we assume the model to be rendered is represented by a non-self-intersecting polygon mesh, no edge of which has more than two adjacent faces – i.e., the mesh is a topological manifold. To make our second assumption precise, we need some definitions:

Definition 1 *A polygon is* **front-facing** *if the dot product of its outward normal and a vector from a point on the polygon to the camera position is positive. Otherwise the polygon is* **back-facing**. *A* **silhouette edge** *is an edge adjacent to one front-facing and one back-facing polygon. A* **border edge** *is an edge adjacent to just one face. A silhouette edge is* **front-facing** *if its adjacent face nearest the camera is front-facing. Other silhouette edges are* **back-facing**.

Our second assumption is that in every image that we render, the view is *generic* in the following sense:

Definition 2 *A view is* **generic** *if (i) the multiplicity of the image of the silhouette curves is everywhere one, except at a finite number of points where it is two; (ii) these multiplicity-two points do not coincide with the projection of any vertices of the mesh; and (iii) their number is invariant under small changes in viewing direction.* [1]

Our method may fail for non-generic views, but we have not observed this in practice when computations are performed with double-precison (64 bit) floating point numbers.

3 Appel's Algorithm

Appel's hidden-line algorithm, as well as those of Galimberti [6] and Loutrel [12], is based on a notion of *quantitative invisibility* (QI), which counts the number of front-facing polygons between a point of an object and the camera. The algorithm is applied to the entire mesh of edges in a polyhedral model to determine QI at all points; those with QI = 0 are visible and are drawn. Good descriptions of the basic algorithm can be found in [5, 2, 18]. We summarize a few key ideas here.

The algorithm first identifies all silhouette edges, because as we traverse the interior of an edge, QI changes only when the edge crosses behind a silhouette. In a generic view, QI can also change at a vertex, but only when the vertex lies on a silhouette edge. This fact is characterized by several authors [18, 5] as a "complication" of the algorithm; we'll discuss this further below.

The algorithm proceeds by determining (via raytracing, for example), the QI of some point in each connected set of edges, and then propagating QI out from this point, taking care to note changes as the edge along which it is propagated passes behind silhouettes, or when a vertex through which it is propagated lies on a silhouette. In this way the number of ray tests is minimized by exploiting "edge coherence."

4 Improving Appel's algorithm

4.1 A fast randomized algorithm for finding silhouettes

Since we focus primarily on rendering silhouettes, and because of their prominence in Appel's algorithm, it's important to find them quickly. But the straightforward approach to finding silhouettes requires an exhaustive search, which conflicts with our goal of achieving interactive frame rates while rendering complex models.

We therefore compromise, and have developed a randomized algorithm for rapidly detecting silhouette edges. We examine a small fraction of the edges in the model, and if we find a silhouette edge, it is easy (by stepping along adjacent silhouette edges) to trace out the entire silhouette curve. If a typical silhouette has 100 edges, we are likely to detect it if we examine only 1% of the edges in the object. Thus the likelihood that a silhouette will be detected is proportional to its length, so that long ones, which are more significant, are more likely to be detected. [2]

If we order the edges by dihedral angle θ, and assign probabilities that decrease as θ increases, we can increase our chances of finding silhouette edges, because given a randomly chosen view, the probability that an edge is a silhouette is proportional to $\pi - \theta$ (in radians).

Given sufficiently small changes in camera position and orientation, it's often the case that a silhouette curve in one frame contains edges that were also silhoutte edges in the previous frame. We exploit this frame-to-frame coherence of silhouettes by always checking every silhouette edge of the previous frame. To further increase the chance of finding silhouettes in the current frame, we select a small fraction of silhouette edges from the previous frame as the starting point for a limited search, traversing edges toward or away from the camera depending on whether the start point lies within a back-facing or front-facing region of surface, respectively. The search stops when a silhoutte edge is found or when the number of edges traversed exceeds a pre-set bound.

When we remove the "seed-and-seek" approach to silhouette-finding and instead check every edge of the model, we observe up to a five-fold increase in total running time for finely-tessellated models (see Section 6).

4.2 Silhouettes and cusps

The "complication" in Appel's algorithm arises because the mapping from the surface to the plane is singular along silhouette edges. Understanding this complicated case better allows us to avoid some unneeded computation. To this end, we first (following [4]) redefine QI to be the number of layers of surface (front- and back-facing) obscuring a point. We then observe that, for generic views, QI along a silhouette curve can change at a vertex only if that vertex is of a special type, which we call a *cusp*:

Definition 3 *A vertex is called a* **cusp vertex** *(or* **cusp***) if one of the following holds (see figure 1):*

1. *it is adjacent to exactly 2 silhouette edges, one front-facing and the other back-facing,*
2. *it is adjacent to more than 2 silhouette edges, or*
3. *it is adjacent to a border edge*[3].

The QI along a non-silhouette curve which intersects a silhouette curve at a vertex can change as it passes through the vertex. Appel's algorithm thus requires a local test at every vertex belonging to a silhouette. But we are interested primarily in propagating QI *along*

[1] This definition is adopted from [19].

[2] Suppose an object's tessellation is refined using a scheme with the following properties: with each subsequent refinement, the total number of edges quadruples, the number of distinct silhouette curves (connected sets of silhouette edges) remains constant, and the number of edges in each silhouette curve doubles. Then it is not hard to show that a constant probability of detecting a silhouette is maintained while checking $O(\sqrt{n})$ edges, where n is the number of edges in a given refinement of the object.

[3] This case is necessary, as shown by figure 1(c), which contradicts corollary 5.1.5 of [4].

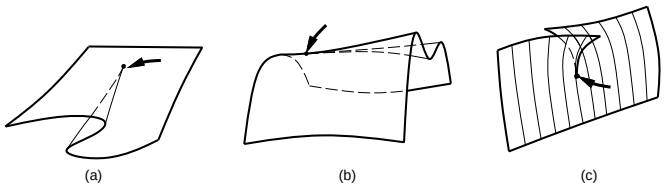


Figure 1 Arrows indicate cusps. (a) A typical cusp. (b) A more exotic cusp. (c) A border cusp (the two edges meeting at the center of the sheet are border edges).

silhouette curves, so testing for changes in QI just at cusp vertices provides a significant savings in computation time. [4]

4.3 Avoiding ray tests

Next, we show how to avoid some of the ray tests required by Appel's algorithm. First, if we assume that all objects in the scene are completely in view of the camera, then any edge which touches the 2D bounding box (in image space) of all silhouettes does not require a ray test – it is automatically visible. Hence, no ray test is required for any connected set (or **cluster**) of silhouette edges containing such an edge.

Appel's algorithm would now proceed with (1) a ray test to establish QI at some distinguished point on each cluster, followed by (2) the propagation step in which QI is assigned to the remaining points of each cluster. By reversing this order, we can sometimes eliminate the need for a ray test altogether, since the second step is often sufficient to determine that an entire silhouette curve is occluded. (See figure 2).

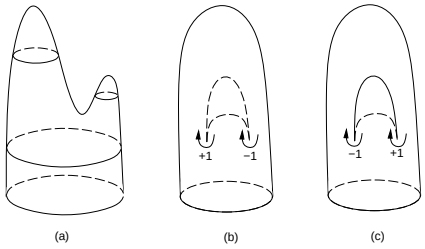


Figure 2 (a) A surface: side view. (b) Smaller branch is in rear. (c) Smaller branch is in front. The change in QI at cusps is indicated. Traversing the inner silhouette in (b) is sufficient to determine that the silhouette is totally occluded.

For each connected cluster of silhouette edges, we first choose an edge and a point on it infinitesimally close to one of its vertices. We call this point the **base point** of the cluster. Let b denote QI at the base point. QI at all other points of the cluster will be defined via offsets from b. We assign a preliminary lower-bound value of 0 to b. We then calculate the offsets with a graph search, taking into account image space intersections of edges of the current cluster with any silhouette edges, as well as cusp vertices encountered in the traversal. (A curve's QI increases by two when it passes behind a silhouette, and may change by an arbitrary, locally measurable amount at a cusp vertex). We record the minimum QI, m, encountered during the search. If $m < 0$, we may safely increment b by $-m$. It's easy to show that on a closed surface, front-facing silhouette edges must have even QI and back-facing silhouette edges must have odd QI. For such surfaces, we add 1 to b if needed to correct its parity. (In that case the cluster is totally occluded – figure 2 shows an example of this).

Finally, we examine each intersection involving edges from different clusters. In this situation, if n is the QI of the occluding edge, and m is the QI of the occluded edge along its unobscured portion, then we must have $m \geq n$. If we find that for our estimated QI values $m < n$, we can increment the base QI of the cluster containing the occluded edge by $n - m$, and propagate this information to other clusters as well.

In practice, these observations often account for all clusters, and consequently no ray tests are required in the current frame. In the remaining cases we perform the needed ray tests efficiently through a technique we call **walking**.

4.4 Walking

Once relative QI values at all points of a silhouette cluster have been determined with respect to the QI b at the base point, we must determine the correct value of b. The following technique does this, assuming all objects in the scene are in front of the camera. (We briefly discuss how to render immersive scenes below).

When one silhouette cluster is totally enclosed by another (in image space), the enclosing silhouette may be the boundary of a region which may totally obscure the enclosed silhouette. (See figure 2 (b) and (c). In (b), the enclosed silhouette is totally occluded, in (c) it is not). We detect such enclosures and their consequent occlusions as follows. First, we disregard silhouette curves which are already known to be totally occluded. We also disregard any silhouette curve which touches the image space box, B, that bounds all silhouette edges (as it can't be totally enclosed). On each remaining silhouette curve, we choose a point U with currently assigned QI of 0. Let U_p denote the projection of U. We identify enclosing silhouettes by tracing a path in image space from U_p toward the boundary of B. From each enclosing silhouette curve S encountered at an image space point V_p, we find the corresponding point V on S. We choose a branch of surface adjacent to S at V along which we can begin tracing a path whose projection heads back toward U_p, if such a branch of surface exists. (Either both branches of surface satisfy this condition or both do not – in which case we proceed to the next enclosing silhouette). We then traverse the surface from V along the path whose projection retraces (in reverse direction) the original path from U_p. If this surface walk succeeds in arriving at a point which projects to U_p, a depth test determines whether U is occluded by that portion of surface.

Our walking method does not work in general for immersive scenes in which geometry may surround the camera. An alternative approach is to perform ray tests efficiently with the use of an octree data structure which can be used to find intersections of a line segment with any triangles in the scene. One problem with this approach is that if there are any silhouette curves in the scene which have gone undetected by the randomized algorithm for finding silhouettes, it's possible for a small region of occlusion in a detected silhouette to be propagated (incorrectly) throughout the entire silhouette. This can occur since intersections with the undetected silhouettes are not taken into account, but the ray test may count occlusions due to surfaces bounded by the undetected silhouettes. (The walking method does not count such surfaces). Taking steps to decrease the probability of missing silhouette curves that lie within the viewing frustum is one approach for minimizing this problem.

The discussion to this point has tacitly assumed that edges of interest are all silhouette edges. These methods easily accomodate border edges and other non-silhouette edges (such as creases or decorative edges) as well. Border edges cause a change of ± 1 in

[4] Note that front-facing and back-facing silhouette edges (used in identifying cusps) can be detected according to whether the surface along the edge is convex or concave; the convexity of each edge can be determined in a pre-process step once-and-for-all.

QI of edges passing behind them. Other edges cause no change in QI of edges passing behind them.

4.5 Implementation details

We follow Loutrel's [12] approach of projecting the silhouette edges into image space and finding their intersections there. This can be done with a sweep-line algorithm in $O(k \log k)$ time, where k is the number of silhouette edges (see e.g. [15]). We found it more convenient to use a spatial subdivision data structure which divides the image space bounding box of the silhouette edges into a grid of cells. Each silhouette edge is "scan converted" into the grid; only edges which share a cell need be tested for intersection. This method has worst-case complexity of $O(k^2)$ but performs well on average. We re-use the spatial subdivision grid in the walking step, in order to find enclosing silhouettes whose projection intersects the image-space path from U_p.

5 Rendering visible lines and surfaces

We demonstrate the use of our visibility algorithm to produce several styles of nonphotorealistic renderings at interactive rates. The accompanying video shows our system in action, and still images produced by the renderer are included at the end of this paper.

World-space polylines to be rendered are first projected into the film plane. Artistic or expressive strokes are then generated by modifying the resulting 2D polylines. We use three techniques for generating expressive strokes: drawing the polylines directly, with slight enhancements such as variations in line width or color (see figure 3(a)); high-resolution "artistically" perturbed strokes defined by adding offsets to the polyline (figure 3(b)); and texture-mapped strokes which follow the shape of the polyline (figure 3(c)). A variation on the first method is to render occluded lines in a style which depends on the number of layers of surface occluding them (figure 3(e)).

In the second method we first parameterize the polyline by arc length. We then define a new parametric curve $\mathbf{q}(t)$ based on the original parametric curve $\mathbf{p}(t)$ by adding a vector offset $\mathbf{v}(t)$ defined in the tangent-normal basis, i.e.:

$$\mathbf{q}(t) = \mathbf{p}(t) + v_x(t)\mathbf{p}'(t) + v_y(t)\mathbf{n}(t).$$

The use of vector offsets allows $\mathbf{q}(t)$ to double back on itself or form loops. Using the tangent-normal basis allows perturbation patterns to follow silhouette curvature. These offset vectors can either be precomputed and stored in lookup tables or computed on the fly. (We have implemented both techniques).

For precomputed offsets, we use a file format which specifies vector offsets. This format also incorporates "break" tags which signal the renderer to leave selected adjacent vertices unconnected in $\mathbf{q}(t)$, allowing strokes to incorporate disconnected shapes, such as circles, dashes, or letters. Variations on a small number of fundamental stroke classes (sawtooth, parabolic undulations, noise) produce a wide variety of stroke styles: high frequency sawtooth curves produce a charcoal style; low-frequency parabolic curves produce a wandering, lazy style; high-frequency, low-magnitude noise applied along the stroke normal and tangent directions produces a jittery hand-drawn style; low-frequency, high-magnitude offsets along the stroke tangent produces a jerky, rough-sketched look.

An alternative method for computing offsets is to use a spatially-coherent noise function indexed by screen-space location. We use a Perlin noise function [14] to define displacements along visible lines. [5]

The third method builds a texture-mapped mesh using the polyline as a reference spine. Each texture map represents a single brushstroke. We repeat the texture along the reference spine, approximately preserving its original aspect ratio. In order to generate the mesh, we walk along the spine adding a perpendicular crossbar at each vertex in the polyline and at each seam between repeating brushstrokes. Additionally, the width of the stroke can be made to vary with lighting computed at the polyline vertices, becoming thicker in darker areas and thinner in lighter areas. Our simple implementation does not handle self-intersections of the texture map mesh due to areas of high curvature.

Lastly we demonstrate a technique for generating curved shading strokes in order to produce a richer artistic effect and to better convey 3D information. (See figure 3(d)). Here, the principle of "economy of line" supports both the esthetic goals and that of maintaining interactive frame rates. We use an extension of the hidden line algorithm which allows us to derive visibility information across surface regions. This method was described by Hornung [9].

We place shading strokes (or *particles*) in world space (on the surface) rather than define them in screen space. This is the approach used by Meier [13] in her "painterly rendering" system. One advantage of this approach is that it maintains frame-to-frame coherence. We make the simplifying assumption that lighting comes from a point source located at the camera position. This greatly simplifies the task of computing stroke placement and density to achieve a target tone. An even distribution of strokes on the surface produces higher apparent densities in regions slanting away from the light – which is exactly where we want a darker tone. Our initial implementation assigns one stroke particle to the center of each triangle, which assumes a sufficiently even triangulation. Strokes are not drawn when occluded or when the computed gray value (using a lambertian shading model) falls below a threshhold.

Stroke directions are defined by the cross product of local surface normal and the ray from the camera to the stroke location, so that strokes line up with silhouette lines. Strokes have a preset world-space length; those with sufficiently large screen-space length are drawn as polylines. The direction of each segment of the polyline is computed as above, with local surface normal taken as a blend of normals at the vertices of the triangle at which the stroke is centered. Finally, we render the strokes using any of the artistic rendering methods described above.

6 Performance

We treat our models as subdivision surfaces, which allows us to refine a given mesh so that it approximates a smooth surface with an arbitrary degree of accuracy. (See [8] for a description of the type of subdivision surfaces we use). The following tables list performance statistics for our renderer operating on models which have been subdivided to the indicated number of polygons. Our test machine is a 200 MHz Sun Ultra ^{TM} 2 Model 2200 with Creator 3D graphics.[6] Our method performs particularly well on smooth meshes, since these have fewer cusps and intersecting silhouette edges than irregular or bumpy surfaces.

In contrast, Winkenbach's [21] pen-and-ink rendering system produces decidedly finer images, but takes several minutes per frame to do so. (Over half that time is spent on visibility determination).

[5] We thank Paul Haeberli for this rendering method and the source code for implementing it.

[6] We use the graphics capabilities for rendering lines only.

Model	Triangles	Frames/sec	Triangles/sec
Two torii	65,536	30.58	2,004,091
Mechanical part	64,512	14.69	947,681
Venus	90,752	17.83	1,618,108

Table 1 Performance of basic visible-line renderer. Times were measured on a 200 MHz Ultrasparc.

Model	Triangles	Frames/sec	Slowdown
Two torii	65,536	5.27	5.8
Mechanical part	64,512	4.30	3.4
Venus	90,752	3.47	5.1

Table 2 Performance of basic line renderer when checking all edges each frame – the slowdown is in comparison with the same models listed in table 1.

Model	Triangles	Frames/sec	Triangles/sec
Blobby teddy bear	7,776	6.37	49,588
Venus	5,672	9.2	52,195

Table 3 Performance of shaded line renderer. Models are those shown in the accompanying video.

7 Future Work

We envision several avenues for future work. Our handling of shading strokes is restrictive and could be generalized to support arbitrary lighting conditions and to better control the density of strokes in screen space to match a target gray value. The shaded stroke renderings would be further enhanced by the addition of cast shadows, which our visibility algorithm can easily be extended to find. (The technique for computing shadow regions is straightforward, and was used in [21]).

More generally, we feel that our exploration of rendering styles can be developed much further. The rendering styles we have demonstrated in this paper take a simple, automated approach in which renderings are produced without regard to the content of the 3D scene, or to the intent of its designer. A rich, unexplored area for future research in NPR is the use of additional information in model definitions which can be used to produce nonphotorealistic renderings which reflect information about a model beyond basic geometric attributes, or which target particular esthetic effects.

8 Acknowledgments

We thank Mark Oribello and Seung Hong for help with images and video, Christine Waggoner for help with modelling, Loring Holden for lending Dan a shell, and Paul Haeberli for the idea and source code for displacing lines with Perlin noise. Also thanks to our sponsors: NSF Graphics and Visualization Center, Alias/Wavefront, Autodesk, Microsoft, Mitsubishi, NASA, Sun Microsystems, and TACO.

References

[1] A. Appel. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of ACM National Conference*, pp. 387–393, 1967.

[2] J. Blinn. *Jim Blinn's Corner*, chapter 10, pp. 91–102. Morgan Kaufmann, 1996.

[3] D. Dooley and M. Cohen. Automatic illustration of 3d geometric models: Lines. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, pp. 77–82, March 1990.

[4] G. Elber and E. Cohen. Hidden curve removal for free form surfaces. In *Proceedings of SIGGRAPH '90*, pp. 95–104, August 1990.

[5] J. Foley, A. van Dam, S. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*, chapter 15, pp. 666–667. Addison-Wesley, 1992.

[6] R. Galimberti and U. Montanari. An algorithm for hidden line elimination. *Communications of the ACM*, 12(4):206–211, April 1969.

[7] P. Haeberli. Paint by numbers: Abstract image representations. In *Proceedings of SIGGRAPH '90*, pp. 207–214, August 1990.

[8] H. Hoppe, T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. Piecewise smooth surface reconstruction. *Proceedings of SIGGRAPH '94*, pp. 295–302, July 1994.

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

[10] J. Lansdown and S. Schofield. Expressive rendering: A review of nonphotorealistic techniques. *IEEE Computer Graphics and Applications*, 15(3):29–37, May 1995.

[11] W. Leister. Computer generated copper plates. *Computer Graphics Forum*, 13(1):69–77, 1994.

[12] P. Loutrel. A solution to the hidden-line problem for computer-drawn polyhedra. *IEEE Transactions on Computers*, C-19(3):205–213, March 1970.

[13] B. Meier. Painterly rendering for animation. In *Proceedings of SIGGRAPH '96*, pp. 477–484, August 1996.

[14] K. Perlin. An image synthesizer. In *Proceedings of SIGGRAPH '85*, pp. 287–296, July 1985.

[15] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*, chapter 7. Springer-Verlag, 1985.

[16] T. Saito and T. Takahashi. Comprehensible rendering of 3d shapes. In *Proceedings of SIGGRAPH '90*, pp. 197–206, aug 1990.

[17] T. Strothotte, B. Preim, A. Raab, J. Schuman, and D. Forsey. How to render frames and influence people. *Computer Graphics Forum*, 13(3):455–466, September 1994.

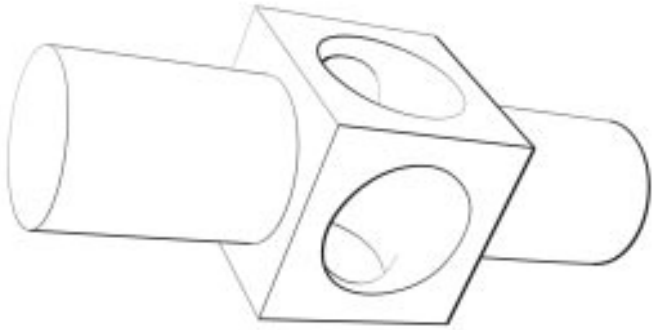
[18] I. Sutherland, R. Sproull, and R. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, March 1974.

[19] L. R. Williams. Topological reconstruction of a smooth manifold-solid from its occluding contour. Technical Report 94-04, University of Massachusetts, Amherst, MA, 1994.

[20] G. Winkenbach and D. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of SIGGRAPH '94*, pp. 91–100, July 1994.

[21] G. Winkenbach and D. Salesin. Rendering parametric surfaces in pen and ink. In *Proceedings of SIGGRAPH '96*, pp. 469–476, August 1996.

[22] R. Zeleznik, K. Herndon, and J. F. Hughes. Sketch: An interface for sketching 3d scenes. In *Proceedings of SIGGRAPH '96*, pp. 163–170, August 1996.

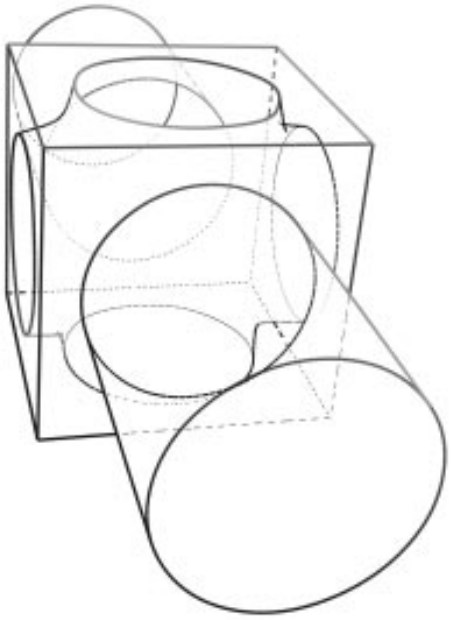


(a)

(b)

(d)

(c)



(e)

Figure 3 (a) A mechanical part (model courtesy of the University of Washington). (b) Mechanical part rendered with sketchy lines. (c) A charcoal-like rendering of terrain with texture-mapped strokes. (d) Human figure with expressive outline and shading strokes. (e) Mechanical part with hidden lines in varied styles.