Rapid Silhouette Rendering of Implicit Surfaces

by
David Bremer

Brown University
Department of Computer Science
Master's Project
May 1998

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University.

[illegible]

Signature (John F. Hughes, Project Advisor)

5/12/98

Date

Contents

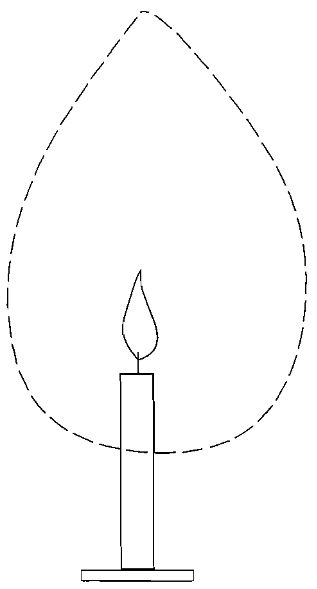

Figure 1: Example implicit surface consisting of the set of all points in space whose temperature is exactly 110 degrees.

1. BACKGROUND

An implicit surface is a 2D surface within a 3D space. The surface is defined as the set of all 3D points that satisfy an equation of the form $f(x, y, z) = 0$, for some function f. For example, suppose that there is a function $t(x, y, z)$ which reports the temperature at a point (x, y, z) in a room, and suppose that there is a burning candle sitting on a table in the room. The set of all points in space at which the temperature is exactly 110 degrees would form a surface around the flame of the candle. See Figure 1. So the function defining this implicit surface would be $f(x, y, z) = t(x, y, z) - 110 = 0$.

Implicit surfaces have several characteristics that make them desirable as a representation for models in computer graphics. If the defining function is continuous, then the surface itself will generically be continuous. Many types of implicit surfaces are very smooth, which makes them the best representation of certain kinds of models. They are also useful in certain types of animation, because simple techniques exist for nicely squashing and stretching the models, and for topology-changing deformations.

However the length of time needed to render implicit surfaces often makes their use impractical. Current rendering methods include ray-tracing, volume rendering, polygonization followed by polygon rendering, and particle rendering [2], none of which can be done very well at interactive rates. Currently, the best choice among these for interactive rendering is sampling the function at a low resolution and then rendering a few polygons or particles on the surface. However, this method is unsatisfactory because it gives a very rough approximation to the surface's true shape. See the SIGGRAPH '97 video conference proceedings for [12] for an example of both a polygonizer and a particle renderer that run at interactive rates for small models.

2. SILHOUETTE RENDERING

Inspired by nonphotorealistic rendering (NPR) techniques, [9, 10, 3, 14, 11, 13, 7] especially interactive NPR for rapid display of complex polyhedral objects [7], we applied an alternative style of rendering to implicit surfaces.

Our basic approach concentrates computational resources on drawing the surface's silhouette quickly and accurately from a given viewpoint, with hidden lines removed. Additionally, a little curvature information is displayed near the silhouette and on the interior of the rendering. Example results of the algorithm, showing off the available features, are shown in Figure 2.

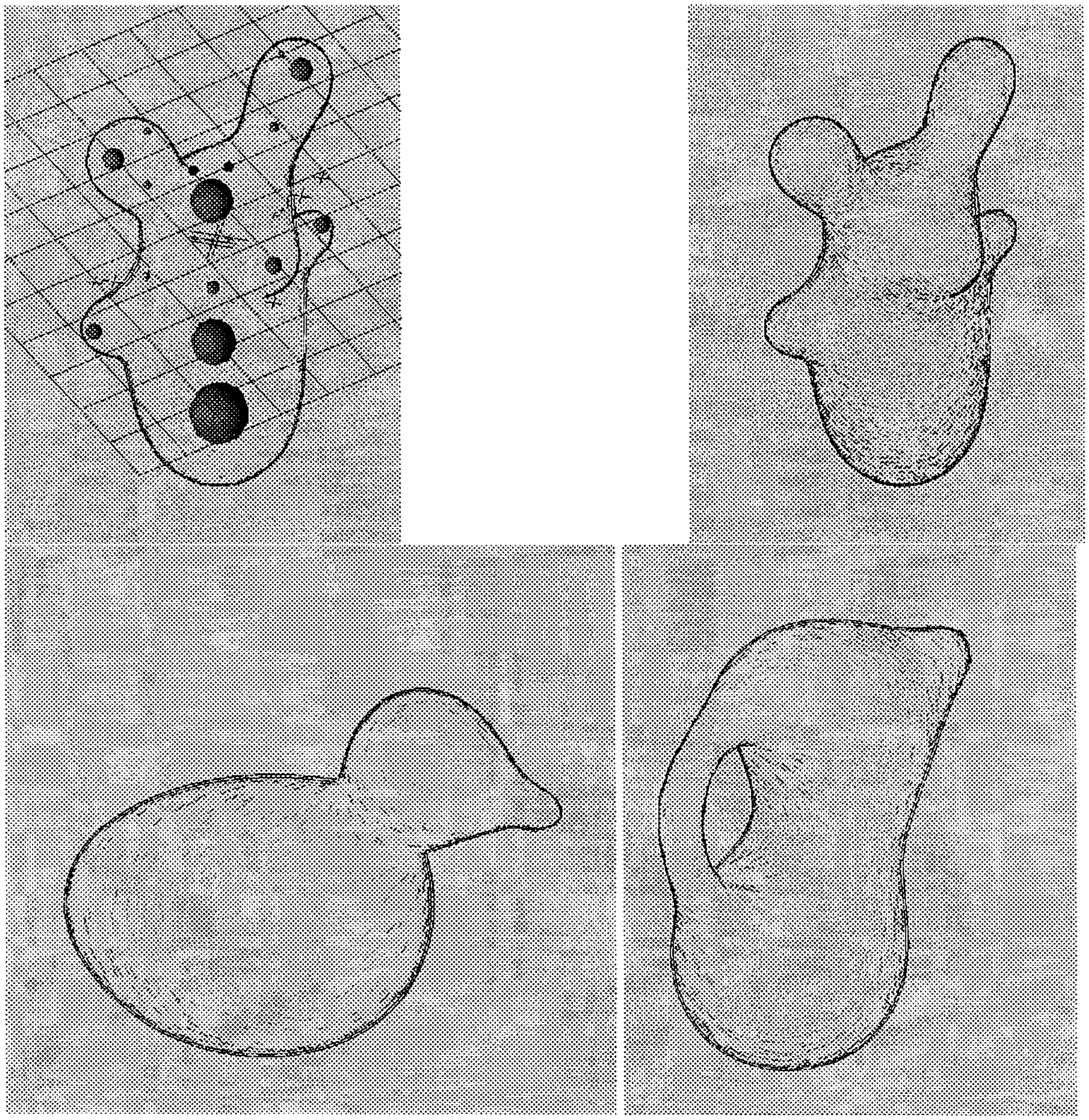

Figure 2: Several models rendered with our nonphotorealistic renderer. (a) A bunny, in the modeler. (b) The bunny rendered without the modeling aids. (c) A duck. (d) A "vase."

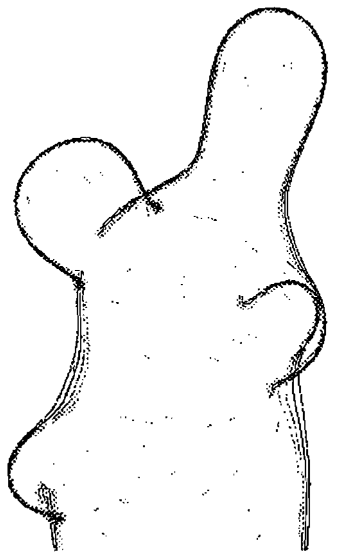
Figure 3: The effects of failing to removing occluded silhouettes on our bunny model.

While standard rendering techniques require that at least the entire surface, if not a whole 3D volume, be traversed to make a rendering, our silhouette edge-drawing approach needs only to trace the silhouette curve or curves, allowing the rendering to take place much faster, and/or to be done more carefully than, say, a rough polygonization. With this algorithm, good rendering can often occur at interactive or near-interactive rates.

This new approach makes good use of computational resources. Although we only draw a few lines on the screen to represent a whole surface, we are giving the eye the visual cues which seem most important to it in perceiving shape. These cues are the position and size of each part of the object relative to others, as well as a bit of curvature information near the silhouettes and across the whole surface. The silhouette lines provide a way to distinguish the surface from the background and to show a little of the surface's shape, with relatively little computational effort. In addition, occluded portions of silhouette edges are culled, because occlusion is among the most important cues our visual system gets regarding the position of parts of the surface relative to one another. It seems that everyone realizes from an early age that if one part of an object is in front of another, the closer part occludes the farther [6, 5], so we felt it absolutely necessary to cull out occluded parts of the silhouette. See Figure 3. In addition, we show a little information about the surface curvature near the silhouette and on the interior of the rendering. It does give a little shape from shading, which is filled in slowly instead of immediately.

For small models, our algorithm's rendering speed is comparable to that of a coarse polygonizer or point renderer, such as the one shown by Stander and Hart [12]. For large models the algorithm should prove to be asymptotically faster, although no longer real-time. For similar reasons, our algorithm would also be asymptotically faster than a volume renderer, which must traverse a whole 3D volume, or at least the whole surface, before producing an image. Because of the asymptotic difference in rendering time, future research might focus on better methods of fairly fast non-interactive rendering of complex implicit surfaces.

The last common method of implicit surface rendering is ray tracing. Ray tracing does seem to give a good feeling of object shape and position, but it is just not fast enough to be useful as a tool for visualizing a model–such as is needed in a modeling program. In the time it takes to generate one nice, ray-traced view of the model, it could be seen from many points of view using our algorithm.

3. PREVIOUS WORK

As mentioned to earlier, our work was primarily inspired by recent work in non-photorealistic rendering (NPR). NPR in general describes any method of drawing a simple abstraction of a complex model in order to highlight important details, or to add extra information or feeling [13]. There are at least two big benefits to rendering this way. The first is the perceptual gain–being able to get more information or feeling from an image or series of images. The other is related to efficiency–we can sometimes save computational expense by computing and rendering only a select portion of a model which is felt to be the most important. Since implicit surfaces are expensive to render at all, we focused on using NPR to make the algorithm fast. Future work in the other area might include shading the whole surface in an expressive way, such as with a pen-and-ink style, and providing a visual way to identify cusps, singularities, or other interesting parts of the model.

Our work was directly inspired by the work of Markosian et. al. [7] which produces a silhouette-rendering of a polygonal model. After seeing it, we tried to adapt the idea of rendering only outlines to the problem of implicit surface rendering.

But that work was not the first to produce line drawings of surfaces. An early line drawing application was developed by Scott Roth [8]. It draws just object outlines, but does so with a ray-tracing algorithm. Similarly, it was developed to give a speed improvement over regular ray-tracing, which it obtains because it is not required to evaluate lighting equations. Unlike the other work, it was designed to operate on CSG models constructed from a few implicitly defined primitive shapes (cubes, spheres, cones, and cylinders).

As a side note, there is a rendering style called contour line drawing which bears a superficial resemblance to ours. This method slices planes through the implicit surface and draws the curves formed by the intersection. [2] The big difference between that method and ours is that it uses curves across the whole surface, in order to show the surface shape, whereas ours draws far fewer curves and does so just to indicate the surface's outline.

4. THE ALGORITHM

4.1. Overview

We need algorithms to find the silhouette edges and information for shading, to test the edges for occlusion, and to draw the edges and shading info. However if while examining the model, the user pauses for a moment, the algorithm need not waste time recomputing silhouettes.

Figure 4 shows the process schematically. The pseudocode is as follows:

```
For each frame
   If the camera has moved or the model has changed
      Find new silhouette edges and shading info
      Test edge sections for occlusion
      Draw the edges and shading strokes
   Else
      Look for missed edges and additional shading info
      Draw all edges and shading strokes
```

4.2. Notation and assumptions

We assume that the surface model, S, that we are rendering is the zero-set of a twice continuously differentiable function f on R^3. To make the explanation simpler, suppose that the solid bounded by the surface is the region in which $f < 0$, which causes gradients to point away from the surface rather than inward.

In order to make the algorithm work with a wide variety of model representations, it treats the implicit function defining the model as a "black box" from which it only needs to be able to get,

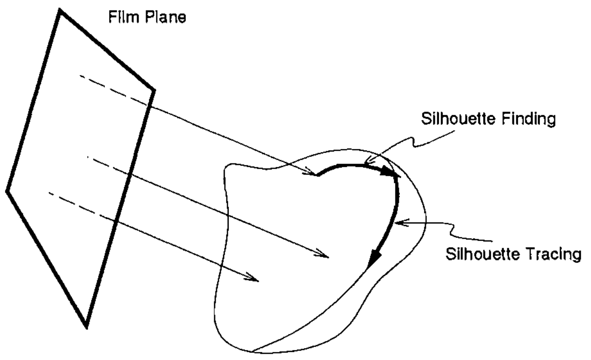

Figure 4: Rays from the film plane are traced along the view direction until they hit the surface. Then we trace in the direction of the view-plane projection of the gradient to try to find a silhouette. Once one is found, we trace along it.

at any point, the function value, the gradient, and the Hessian (matrix of second derivatives) of the function.

We further assume that the virtual camera is orthographic with view direction $\mathbf{v}$, that all points on the film plane are outside the object (i.e., $f > 0$ on the film plane), and that the surface S lies entirely within some known region of space (a sphere of radius 20 about the origin in our particular implementation).

We denote points and vectors by boldface letters; the point $\mathbf{x}$ has coordinates x_1, x_2 and x_3.

We also assume, to make ray-surface intersection easier, that there is a constant $K > 0$ such that at every point $\mathbf{x}$, the gradient of f,

$$\nabla f(\mathbf{x}) = (\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \frac{\partial f}{\partial x_3}(\mathbf{x}))$$

has magnitude bounded by K. Finally, we denote the *Hessian* of f, the matrix of second partial derivatives, by Hf, so that

$$Hf(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_1 \partial x_3}(\mathbf{x}) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_2 \partial x_2}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_2 \partial x_3}(\mathbf{x}) \\ \frac{\partial^2 f}{\partial x_3 \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_3 \partial x_2}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_3 \partial x_3}(\mathbf{x}) \end{bmatrix}.$$

4.3. Silhouette Finding

Silhouette finding is a three-step process:

1. Locate a point on the surface through ray-surface intersection
2. Trace along the surface to a point on a silhouette
3. Trace out the silhouette

4.3.1. Ray-surface intersection

We apply a modification of Kalra and Barr's [2] implicitization algorithm to do ray-surface intersection: the idea is that for functions with bounded gradients, we can search for ray-surface intersections by stepping along a ray and be guaranteed to miss no intersection: if, while searching for an intersection along the ray $\mathbf{p} + t\mathbf{v}$, we are at location $\mathbf{x}$, then we can take a step of size $f(\mathbf{x})/K$ to location $\mathbf{x}' = \mathbf{x} + (f(\mathbf{x})/K)\mathbf{v}$ and be confident that $f(\mathbf{x}') \geq 0$. We search along rays from the eye until the function value is nearly zero, and call the resulting point a surface point. If the search proceeds far enough, our assumption that the surface lies within a bounded region of space lets us terminate the search. In this case, the silhouette finding process does not continue with silhouette point finding, but rather with ray-surface intersection using a different ray.

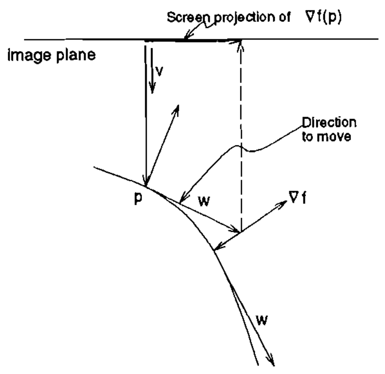

Figure 5: When we reach a point **p** of the surface, we find a tangent vector whose screen projection is in the same direction as that of the gradient (that tangent vector's called w here), and move in that direction to find a silhouette.

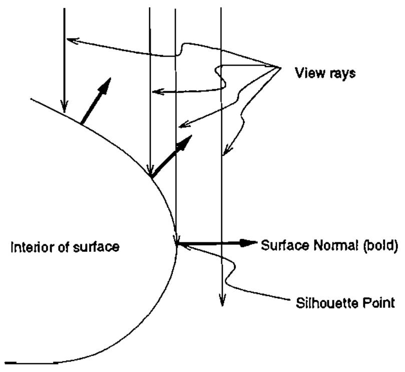

Figure 6: When a view ray hits the interior of the surface (shown in cross-section here), the surface normal (show in bold) points back towards the eye. When the view ray grazes the silhouette of the surface, the view ray and the surface normal are orthogonal.

4.3.2. Silhouette point finding

When we find a ray-surface intersection, we try to use it to locate a silhouette by walking along the surface in the direction of the screen-projection of the gradient at each point (see Figure 5).

We take the ray-surface intersection **p**, compute the function gradient at **p**, and use this to find a tangent vector in the plane spanned by **v** and ∇f, i.e., we let: [1]

$$F(\mathbf{p}) = \frac{\nabla f(\mathbf{p}) \times (\mathbf{v} \times \nabla f(\mathbf{p}))}{\|\nabla f(\mathbf{p})\|^2}$$

The vector field F is tangent to the surface and lies in the plane spanned by **v** and $\nabla f(\mathbf{x})$. We trace along this vector field until the dot product of **v** and ∇f changes sign, which indicates that we have passed a silhouette. A silhouette point is a point of the surface where the tangent plane contains the view direction; it may be obscured by some other part of the surface, but we call it a silhouette point nonetheless. In the mathematical literature, it's sometimes called a "fold point." See Figure 6.

So far, we have just described the vector field to be integrated, not the method of integration. See Section 5 for a description of the integration method.

Often, after a few silhouettes have been traced out, the point found will lie on one of the silhouettes already traced out. So before proceeding, we test to see if this is just such a point. See Figure 7.

[1] The division by the square of the gradient is designed to make the vector field independent of the scale of f: if we replace f with αf, the vector field is unchanged.

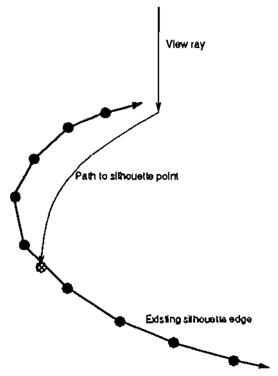

Figure 7: A new silhouette point, shown in grey, should be tested to see if it lies on a silhouette edge that was already traced out, to avoid recomputing the position of a silhouette edge.

Silhouettes are represented as 3D polylines. We do a proximity test between the found point and all the silhouette points which may be near it. (These points are stored in a hash table based on location in three-space.) All the points in the polyline are approximately ϵ units apart, so if the found point is within ϵ units of any found point, it is discarded, and the silhouette finding process starts over with a new ray-intersection.

4.3.3. Silhouette Tracing

A curve $h : \mathcal{R} \to \mathcal{R}^3$ lies on the silhouette of the surface S viewed along $\mathbf{v}$ if

- $h(t) \in S$ for every t, and
- The tangent plane to the level surface at $h(t)$ contains $\mathbf{v}$ for every t.

These two conditions can be rephrased as

$$\begin{aligned} f(h(t)) &= 0 \\ \mathbf{v}^t \nabla f(h(t)) &= 0. \end{aligned}$$

Rather than try to solve for $h(t)$ analytically, we instead use this implicit description to determine the tangent vector of h, from which we can determine h by numerical integration.

Differentiating each equation with respect to t, applying the chain rule, and using $\mathbf{w}$ to denote $h'(t)$, we get

$$\begin{aligned} \nabla f(h(t)) \cdot h'(t) &= 0 \\ \mathbf{v}^t H f(h(t)) h'(t) &= 0, \end{aligned}$$

i.e.,

$$\begin{aligned} \nabla f(h(t)) \mathbf{w} &= 0 \\ \mathbf{v}^t H f(h(t)) \mathbf{w} &= 0. \end{aligned}$$

Thus the tangent vector to a silhouette curve must be orthogonal both to the gradient at its basepoint, and to the product of the Hessian at the basepoint with the view direction. This makes it proportional to the cross product of these:

$$\mathbf{w} \propto \nabla f(h(t)) \times \mathbf{v}^t H f(h(t)).$$

We can therefore trace along a silhouette by computing

$$\mathbf{w} = \nabla f(\mathbf{p}) \times \mathbf{v}^t H f(\mathbf{p})$$

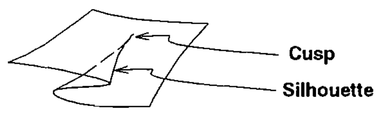

Figure 8: A cusp occurs at the end of a silhouette. A slight "hook" is conventionally drawn at the cusp to indicate its shape.

and finding an integral curve for this vector field. Of course, at locations where $\mathbf{w} = 0$ the tracing process stagnates. This happens, for example, at cusps like the one shown in Figure 8. Our silhouette tracing algorithm starts from a silhouette point, found previously, and traces out the silhouette until the tracing process stagnates or returns to the starting point; if it stagnates, we return to the starting point and trace in the other direction. Section 5 discusses the details of the tracing scheme.

We must also take care not to keep tracing past the starting point. We take a series of steps along the curve of size ϵ. After each step, we find the distance between the new point and the starting point. If the distance is less than ϵ, tracing stops and the 3D polyline is closed.

4.4. Occlusion Testing

The silhouette curves are recorded as 3D polylines. We test the vertices of the silhouette for occlusion by first checking every nth (4th in our implementation) vertex for occlusion, and then, for those between which occlusion status changes, testing the intervening vertices as well. To check occlusion of a single vertex of a silhouette polyline, we start at the vertex, move back towards the film plane from it, and do a ray-surface intersection test back into the scene. If the ray intersects the surface at a place much closer to the film plane than our vertex, we declare the vertex invisible; otherwise it's visible.

Note that because of numerical issues, the ray may not intersect the surface exactly at the silhouette vertex (which may, indeed, not actually lie exactly on the surface), so the "much closer" test is important. Unfortunately, if the true silhouette is just barely obscured by some nearer piece of surface, and the silhouette vertex happens to still seem to be visible, we draw still the silhouette.

4.5. Rendering

4.5.1. Silhouette Edges

The silhouette edges could be simply drawn as polylines, as would be done in a basic implementation. But, to convey extra information about the surface's shape near the silhouette, we alter the drawing style based on the local curvature. At a point $\mathbf{p}$ of the silhouette, the Hessian can be used to determine the curvature of the surface in the "heading-into-the-distance" direction. To be more precise, if we consider the plane through $\mathbf{p}$ spanned by the gradient and the view direction, its intersection with S is a curve. The gradient to S is normal to this curve, and the rate of change of this normal in the view direction is proportional to the curvature of the curve. But the rate of change of the normal as we move the basepoint in some direction $\mathbf{u}$ through the point $\mathbf{p}$ is simply $\mathbf{u}^t Hf(\mathbf{p})$; to compute its component in the view direction, we take the inner product with the view vector. We can therefore compute the curvature in the view direction as $\mathbf{v}^t Hf(\mathbf{p})\mathbf{v}$. Unfortunately, this computation depends on the scaled f; we normalize it by dividing by the magnitude of the

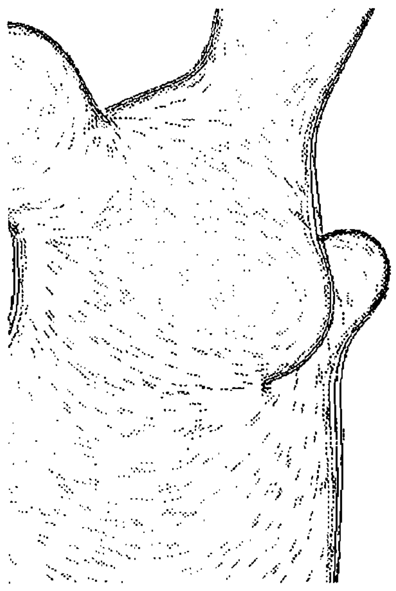

Figure 9: The silhouette drawing style varies with the surface curvature. The tip of the bunny's left arm is drawn with tightly spaced lines, whereas the lines defining its torso are widely spaced.

gradient of f. Hence we compute

$$\kappa(\mathbf{p}) = \frac{\mathbf{v}^t Hf(\mathbf{p})\mathbf{v}}{\|\nabla f(\mathbf{p})\|}$$

for points on the silhouette, and use it to help us draw shading near the silhouettes to indicate curvature.

So we draw not just the silhouette but several parallel copies of it, with the inter-copy spacing proportional to $1/\kappa(\mathbf{p})$. Thus tightly-curved sections get closely-spaced curves, and areas of shallow curvature get widely-spaced ones. See Figure 9.

We could also experiment with stroke styles as did Markosian et al [7].

4.5.2. Interior Shading

Interior shading strokes are drawn using a very simple lighting model – we assume that the light in the scene is arriving from behind the virtual camera, and that the surface is diffuse, so that the illumination is proportional to the dot-product of the view direction and the (unit) surface normal. At interior points, when a ray strikes the surface, we immediately compute the gradient at the intersection point so that we can start searching for a silhouette. We use this computed gradient to determine two additional things:

- the direction $v \times \nabla f$ that's tangent to the curve of constant illumination (isophote), and
- the lightness s of the surface $(\frac{v \cdot \nabla f}{\|\nabla f\|})$ at the intersection point.

We then pick a color $|s|bg + (1 - |s|)db$, where bg is the background color (a neutral gray) and db is a dark blue, and draw a short stroke in this color, tangent to the isophote.

These "free" shading lines accumulate as rays are shot at the surface in search of silhouette edges, and help convey the interior shape of the surface. See Figure 10.

5. EFFICIENCY CONSIDERATIONS

5.1. Approximate Tracing

In both silhouette finding and silhouette tracing, we need to "walk along" the implicit surface, guided by a vector field. In each case the general algorithm we use is Euler integration: $\mathbf{p}$ is

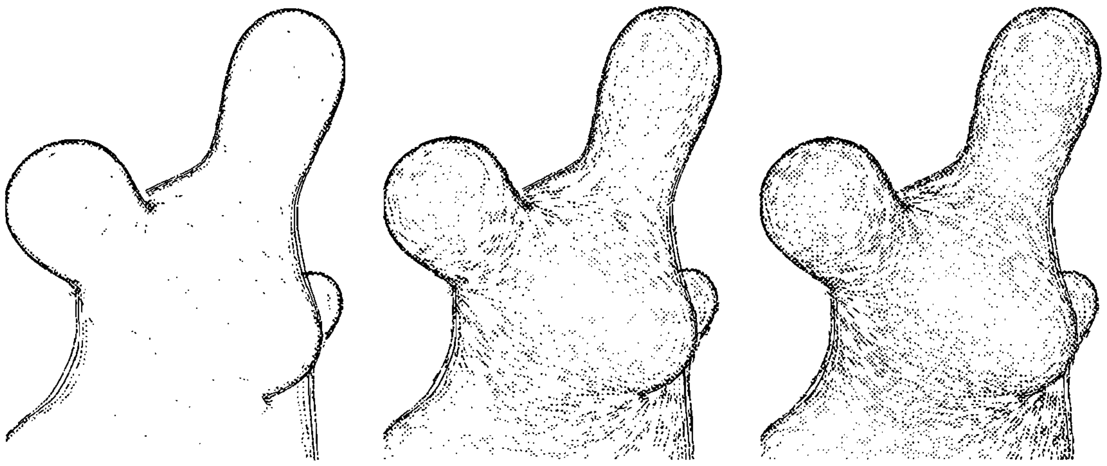

Figure 10: Several images of the same model, with progressively more strokes filled in. Note the light strokes on places perpendicular to the view direction, such as the bunny's nose, and dark strokes near silhouette edges.

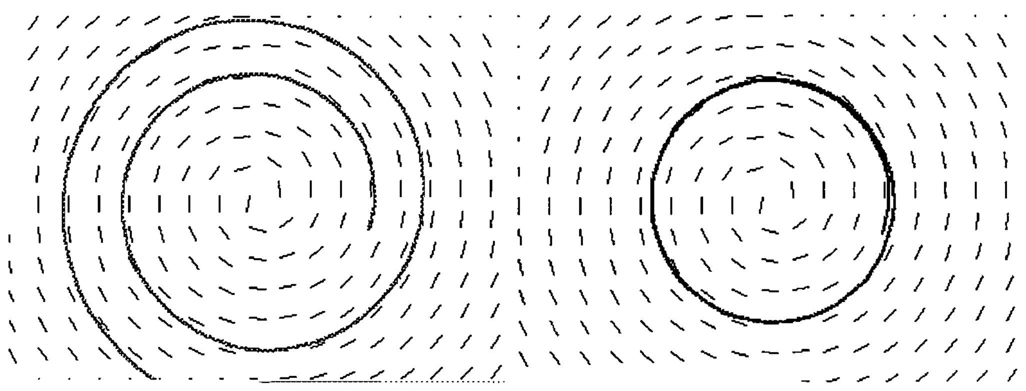

Figure 11: (a) When the tangent field to a family of circles is integrated with Euler integration, the result is a growing spiral. (b) When we add a penalty term for distance from the starting circle, the result is a non-diverging circle (albeit slightly displaced from the starting point's circle).

replaced by $\mathbf{p} + \epsilon F(p)$, where F is the vector field, and ϵ is some small number. This approach is notoriously unstable; using it to walk along the circumference of a circle (i.e., along the vector field $F(x, y) = (-y, x)$) leads to the sort of spiral shown in Figure 11(a). But if in addition to knowing that we want to be guided by a vector field, we have some other constraint, we can use this to help stabilize the process. For example, in silhouette finding, we know that we not only want to move along the surface in the direction determined by the gradient and view direction, we also want to remain on the surface. By adding a "penalty" term to the vector field – a term that's zero on the surface, but drives us back to the surface when we're off it – we can ensure that the integral curves don't wander too far. In the case of the tangent vector field to the circle, we can use the vector field $G(x, y) = K * (x^2 + y^2 - 1) * (-x, -y)$ as a "corrector" field; when we add this to F, the integral curves, even with Euler integration, now lie close to the circle rather than following a diverging spiral (see Figure 11(b)). This idea is closely related to the constraint-satisfaction method in Barzel and Barr's "Dynamic Constraints" work [1]. The constant K determines the degree of penalty for falling off the circle: if K is small, the curve will not stay close; if K is made too large, however, the curve can oscillate across the circle.

In the case of silhouette finding, we know that we want to remain on the surface as we search for a silhouette. Our first implementation took small steps and then, at the end of each step, did a ray-surface intersection to "fall back" onto the surface. Our revised version instead uses the stabilization method: instead of using the vector field

$$F(\mathbf{x}) = \frac{\nabla f(\mathbf{x}) \times (\mathbf{v} \times \nabla f(\mathbf{x}))}{||\nabla f(\mathbf{x})||^2}$$

defined on the surface, we define (on all of $\mathcal{R}^3$) the vector field

$$F(\mathbf{x}) = \frac{\nabla f(\mathbf{x}) \times (\mathbf{v} \times \nabla f(\mathbf{x})) - f(\mathbf{x})\nabla f(\mathbf{x})}{||\nabla f(\mathbf{x})||^2}.$$

The additional term – $\frac{-f(\mathbf{x})\nabla f(\mathbf{x})}{||\nabla f(\mathbf{x})||^2}$ – is a field that points towards the surface at all points of space. Hence when Euler integration takes our curve off the surface, the additional term tries to coax it back onto the surface. Just as in the case of the spiraling circle, the correction is imperfect: the "stabilized" path still does not lie exactly on the surface. But it does not diverge from it either, and the expensive ray-surface intersection is eliminated. By the way, this is just a specialized type of predictor-corrector integrator; the novelty is in its application to finding silhouettes for isosurfaces.

For the case of silhouette *tracking*, our initial corrector takes the predicted location and does a ray-surface intersection (moving in the negative gradient direction) to fall back to the surface, and then a silhouette-finding operation to fall back to the silhouette. In the current implementation, these two steps use the same algorithms that initially are used to find the surface and then a silhouette point. In this system, on reaching a cusp the corrector no longer returns the right value, which is why tracing stops at cusps.

We have since implemented a corrector like the one described for silhouette finding. The pictures here, however, use the original method, since we have not thoroughly tested the new corrector. For this new corrector, we have two additional constraints: we want to find integral curves of the vector field

$$G(\mathbf{p}) = \nabla f(\mathbf{p}) \times \mathbf{v}^t H f(\mathbf{p})$$

on the surface, but Euler integration will wander off the surface and off the silhouette curve. Again, we can add a correction of the form $\frac{-f(\mathbf{x})\nabla f(\mathbf{x})}{||\nabla f(\mathbf{x})||^2}$ to keep the curve on the surface. We can also add a correction to keep the curve running along the silhouette: just as adding $-f\nabla f$ tends to drive f to zero, we can add $-g\nabla g$, where $g(\mathbf{p}) = \mathbf{v} \cdot \nabla f(\mathbf{p})$ to drive g to zero, i.e., to drive us onto a silhouette. This expression simplifies to

$$-(\mathbf{v} \cdot \nabla f(\mathbf{p}))\mathbf{v}^t H f(\mathbf{p}).$$

Fortunately, ∇f and Hf are already computed in getting the basic vector field to walk along.

In summary, we find an integral curve of

$$\begin{aligned}\frac{1}{||\nabla f(\mathbf{p})||^2}(\nabla f(\mathbf{p}) \quad \times & \quad (\mathbf{v} \times \nabla f(\mathbf{p})) - f(\mathbf{p})\nabla f(\mathbf{p}) \\ & \quad -K * (\mathbf{v} \cdot \nabla f(\mathbf{p}))\mathbf{v}^t H f(\mathbf{p}))\end{aligned}$$

and it will not only follow the silhouette, but if it (because of Euler steps) wanders from the silhouette, will be driven back towards it.

In our initial tests, setting K to 1 has led to some oscillatory behavior; $K = 0.5$ seems to work well, however.

As an alternative to Euler integration, we could use Runge-Kutta integration. Runge-Kutta integration requires more computation for each step, but diverges much more slowly from the ideal curve than Euler integration, allowing bigger steps to be taken. But both methods still diverge, so regardless of which we use, we would still want to take advantage of the special correction information available to us, which lets us pull the curve directly back toward the surface or curve on which it should lie. Future work might include testing to see if, and how much, the speed gained from Runge-Kutta's bigger steps offsets the cost of extra computation.

5.2. Choosing good rays to shoot

Our algorithm begins by shooting rays from the film plane along the view direction (orthogonal to the film plane in our implementation) into the scene, hoping to find silhouettes. The silhouette-finding algorithm can easily get stuck in "valleys" in the surface, so some rays produce nothing of interest. On the other hand, a ray that falls near a silhouette will rapidly lead to productive results. Because we are trying to render at interactive speeds, we have some confidence that inter-frame differences in the image are small, so silhouettes in a frame are likely to be near their locations in the previous frame. Thus former silhouette points are good candidates for ray-starting-points in the current frame. If we displace these points slightly "inward" along the surface normal, then surface (or camera) translations are less likely to cause them to miss the surface when they're re-shot. We therefore, in choosing rays to shoot in each frame, preferentially select starting points that lie on silhouettes from previous frames; we also use some randomly chosen rays, in hopes of finding new silhouettes that may appear far from any previous silhouette.

6. TIMING

Initial timing tests for an execution that involves frequent camera motion suggest that the bulk (60%) of the algorithm's time is spent determining silhouette curves. Of this, half (30%) is spent shooting rays, many of which miss the surface (although this depends on the screen-area occupied by the surface), a quarter (15%) on silhouette-finding, and the remaining quarter on silhouette tracing.

Another 30% is spent on occlusion testing, virtually all of it in ray-surface intersection computations.

The remaining 10% is spent drawing the shapes, doing object-creation in Java3D, and handling thread synchronization and other tasks unrelated to the algorithm.

By contrast, during a model-creation session, about 80% of the time was spent determining silhouette curves (70% ray-shooting, 10% silhouette-finding, 20% silhouette-tracing), and about 20% doing occlusion testing. A small amount was spent creating a drawing shapes in Java3D.

7. LIMITATIONS AND FUTURE WORK

The algorithm described here has some serious limitations. We require that f, ∇f and Hf all be available at all points of the model that we render. For sampled data, these might be provided by performing some tricubic interpolation of the samples, although we have not implemented this. Further, our ray-surface intersection requires the bound on the gradient magnitude, although it could be replaced with some other method if no such bound is available.

The occlusion testing uses only samples of the silhouettes, hence is prone to small errors. If we knew that we had computed all silhouettes, and projected them to 2D, we could apply the methods used by Markosian et al. [7] instead. It may well turn out that this is more efficient, because it would drastically reduce the number of ray-surface intersection tests we need to perform. Furthermore, it would allow us to do 2D region-fill operations to make the surface interior be a different color from the background, which would presumably help in indicating the object's shape.

There are two additional cues to the shape that could probably be shown by using a perspective camera: motion parallax and the fact that objects diminish in size with increasing distance from the viewer. The first, and maybe also the second, seem most effective in a system like ours which views models at interactive rates. The current implementation uses only an orthographic camera; replacing it with a perspective camera is a small change, but the vector $\mathbf{v}$, which is constant for an orthographic camera, becomes dependent on the viewed point for a perspective camera, which would add some modest computation.

Our system cannot render texture maps on the surfaces, and indeed, since we sample as few points on the surface as we can, we see no way to include this.

We would also like to push the limits of NPR further. For example, the rendering near cusps, where silhouette edges disappear, has a disappointing (to us) appearance, with the "shading curves" fanning out. Hand-drawn cusps like the one in Figure 8 present a far more attractive appearance, and we'd like to somehow copy this. It would be nice to be able to find and detect other interesting features, such as sharp edges and singularities. In addition, we might experiment with a slower version of the algorithm which would draw shading across the whole surface, perhaps in a pen-and-ink style.

The tradeoff between step size and speed is only partly successful: if we increase the step size too much, we either spend excessive time in the explicit correctors (re-intersect surface, re-find silhouette) or the implicit correctors can fail because the assumption that the point is not far from the surface, so that gradient forces can bring it back on, fails.

As mentioned earlier, it would be good to try other methods of integrating the silhouette curve, such as with Runge-Kutta integration, to see if we can make a gain in efficiency.

Our use of Java3D is unsatisfactory: it seems foolish to create curves in 3-space so that a 3D renderer can redraw them for us in 2D. But with the optimizations in Java3D, it appears (at least on our Sun workstations) to be faster to do this than to draw directly in 2D.

8. FINAL NOTES

The project can be executed from the directory /u/djb/impl. Once there, type java Illus3D to produce an environment in which you can create a surface, or type java Illus3D [file].blob to view one of the sample models.

9. REFERENCES

[1] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constraints. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 179–188, August 1988.

[2] Jules Bloomenthal, editor. *Introduction to Implicit Surfaces.* Morgan Kauffman Publishers, Inc., 1997.

[3] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-generated watercolor. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 421–430. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[4] David P. Dobkin, Silvio V. F. Levy, William P. Thurston, and Allan R. Wilks. Contour tracing by piecewise linear approximation. *ACM Transactions on Graphics*, 9(4):389–423, 1990.

[5] E. Bruce Goldstein. *Sensation and Perception.* Brooks/Cole Publishing Company, 1996.

[6] Victoria Interrante. Perceiving and representing shape and depth. SIGGRAPH 97 Course Notes for Principles of Visual Perception and Its Applications in Computer Graphics, August 1997.

[7] 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.

[8] Scott D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, 1982.

[9] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen–and–ink illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 101–108. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[10] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable textures for image-based pen-and-ink illustration. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 401–406. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[11] Mike Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. Scale-dependent reproduction of pen-and-ink illustrations. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 461–468. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[12] Barton T. Stander and John C. Hart. Guaranteeing the topology of an implicit surface polyganization for interactive modeling. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 279–286. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[13] Georges Winkenbach and David H. Salesin. Computer–generated pen–and–ink illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 91–100. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[14] 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. SOFTWARE DESIGN

A.1. Important System Classes

`Function3D`

Abstract class used to define the function from which the implicit surface is derived. The function needs to be defined and twice differentiable at each 3D point. Exactly one instance of this class should be in the system at all times.

```
double    eval(Point3d)
```

Evaluates the function at a point in space.

```
void      gradient(Point3d p, Vector3d result)
```

Finds the gradient at point p, and places that value in `result`.

```
void      hessian(Point3d p, Matrix3d result)
```

Finds the hessian at point p and places it in the 3x3 matrix `result`.

`ModelTracer`

This is used to find and trace out a 3D silhouette curve. It has a `Function3D`, and it traces a surface defined by the set of points in the function that are equal to some target value. Normally, the model tracer uses a default target value that is stored with the function.

```
boolean   intersect(Point3d startpt, Vector3d dir,
                    double maxdist, Point3d result)
```

Shoots a ray from `startpt` in direction `dir`, looking for a point at which the function is approximately equal to the target value. If the ray travels for more than `maxdist` units, it stops and returns false. Otherwise it returns true.

```
boolean   findSilhPoint(Point3d startpt, Vector3d viewdir,
                        Point3d result)
```

Travels across the surface to find a silhouette point: a point at which the gradient vector is roughly perpendicular to the view direction.

It works by doing a series of predictions and corrections. See Section 4.3.2 for more about the vector field we follow, and Section 5.1 to find out how it is integrated.

It takes steps of uniform size, until it steps past the silhouette edge. Once it steps past the end, it reduces the step size, backs up one step, and starts tracing again, to get better accuracy.

`Silhouette traceSilh(Point3d startpt, Vector3d viewdir)`

Traces along the surface, always staying close to the ideal silhouette curve, using a series of predictions and corrections. It returns a sequence of 3D points that represent a silhouette.

Prediction Step:
Using Euler integration, just take a uniform size step in the direction described in Section 4.3.3.

Correction Step:
Current way (Being tested): First, use the vector field to fall back onto the surface, as described in Section 5.1. Then, use another vector field, described later in Section 5.1 to get back to the silhouette. This method successfully moves around cusps, but can drift a little from the ideal silhouette.

Old way: First, use the vector field to fall back onto the surface, as described in Section 5.1. Then, call `findSilhPoint` to travel across the surface back to the silhouette. When a cusps is reached, this process fails to follow the curve, so tracing must be stopped in the current direction.

`void        fillinSilh(Silhouette sil)`

This is used to continue tracing out a silhouette. It is useful because sometimes the process of tracing is stopped early due to time constraints. The algorithm is virtually the same as that used for `traceSilh`. It starts from each end of the passed in silhouette, and traces as far as possible in each direction, adding new points as it goes.

`EdgeFinder`

This object is responsible for finding silhouette edges and passing them along to the `EdgeProcessor`. Its search is guided to some extent by the `EdgeProcessor`, which gives it unfinished edges and promising world-space points to search near.

A separate thread executes most of the methods in this object.

`void run()`

The thread for this object repeatedly does the following:
If there are any `Heuristic` objects available, process them. For example, a `Heuristic` object might contain a partially traced out edge that should be traced further by calling `ModelTracer.fillinSilh(...)`. If there are no `Heuristic` objects left, do some default search, like shooting random rays.

`Silhouette findEdge(Point3d startpt, Vector3d dir, Vector3d viewdir)`

Finds a silhouette edge.

1. Call `ModelTracer.intersect` to get a point on the surface, by shooting a ray from `startpt` in direction `dir`.
2. Call `ModelTracer.findSilhPoint(...)` to get a point on the silhouette edge.
3. If this point is not on near any edges that were already found, Call `ModelTracer.traceSilh(...)` to get the edge

`void pause()`

This method is invoked by another thread, to force the `EdgeFinder` to stop searching or tracing an edge 'very soon', and to hand any partially traced edge to the `EdgeProcessor`. Works by setting a 'stop' switch in the `EdgeFinder`.

`void cont()`

Allows the `EdgeFinder` to continue, but does not wake it up if it is asleep. Works by unsetting a 'stop' switch in the `EdgeFinder`.

`EdgeProcessor`

This class is responsible for processing edges in various ways, such as doing occlusion tests, adding shading strokes, and putting the edges into a draw-able form. A separate thread executes most of the methods in this object. That thread is asleep until the `EdgeFinder` adds an edge.

`void run()`

The thread for this object repeatedly does the following:

1. Sleep until an edge is added.
2. Remove the edge and create a `SilhEdge` object, which encapsulates all the info from processing, such as world-space points, screen-space points (not used currently), occlusion info about points, and each edge in a draw-able form.

`void draw()`

This is called by the main thread, the one that does all the drawing for Java3D. It iterates through all the `SilhEdge` objects, telling each to draw itself.

`ImplCanvas3D`

This object is used by the Java3D rendering engine. It extends Java3D's Canvas3D object, which allows rendering in immediate mode.

Every time a frame is drawn, this method is automatically called.

`void renderField(int unimportant_parameter)`

1. Pause the `EdgeFinder`.

2. Wait for the `EdgeProcessor` to process all the edges it has.
3. Call `EdgeProcessor.draw()` to actually draw all the silhouettes and shading strokes.
4. Update the camera matrix, if necessary, and notify the `EdgeFinder` and `EdgeProcessor` of changes, so they can update their view directions, clear out the `SilhEdges`, add appropriate heuristics, etc.
5. Unpause the `EdgeFinder`.
6. Force this thread to go to sleep for a little while, to give the `EdgeFinder` and `EdgeProcessor` time to make new edges.

`Heuristic`

These objects encapsulate information that the `EdgeFinder` uses in its search for edges. Currently they can hold three types of information: a partially traced out edge which should be completed, a point in space at which a ray should be shot for a likely hit, or a point on the screen through which a ray should be shot.

A.2. Flow of Control

Flow of control through the main thread of the application–the one that executes the drawing of the lines at the right time.

Soon after previous frame is swapped in, the main thread calls the following method, in which the most complex synchronization takes place.

`ImplCanvas::renderField(int)`

1. Signals the `EdgeFinder` to stop "very soon", and puts itself to sleep until the `EdgeFinder` complies. The signaling is done by setting a boolean 'stop' switch in the `EdgeFinder` object, which the `EdgeFinder` thread checks periodically. When it is at a good stopping point, it will wake up the main thread and put itself to sleep. So now the `EdgeFinder` is in sync.

   //The Code. Both calls are made from renderField(int)
   ```
   _edge_finder.pause();
   _edge_finder.accessLock().lock();
   ```

   Side note: There are several ways to decide when `EdgeFinder` should stop. Small changes to the code can make it stop right in the middle of finding an edge, or immediately after it has finished tracing out its current edge, or not until it decides somehow that it has made a pretty thorough search for edges.

2. Next, the `EdgeProcessor` also needs to be stopped at a safe place. The `EdgeProcessor` has an "Access Lock" called _access_lock which it locks when it has an edge to process. Once it has done all its processing, it releases the lock and goes to sleep.

   So, the main thread simply tries to get the lock. If it doesn't get it immediately, it goes to sleep and lets the `EdgeProcessor` finish all its work. Once the `EdgeProcessor` finishes, it wakes up any waiting threads, and puts itself to sleep.

   This, oddly enough, takes place inside `EdgeProcessor::draw()` , which is called by the main thread from `renderField(int)`.

   `EdgeProcessor::draw()`

```
_access_lock.lock(); //pause to wait for EdgeProcessor to finish
Draw all the edges...
Draw shading info...
_access_lock.unlock();  //allow EdgeProcessor to resume
```

3. Now we're almost done with **renderField(int)**. The model description and camera info are shared by many objects, including those that want to update them. However, neither of these are allowed to change while the EdgeFinder or EdgeProcessor are running, because the effects would just be bad. So, they are updated at this safe point, when the EdgeProcessor and EdgeFinder are stopped and everything has been rendered.

   Then, the boolean switch asking the EdgeFinder to stop is unset by the main thread, and the EdgeFinder thread is awakened, using these calls:

   ```
   _edge_finder.cont();  //unset 'stop' switch
   _edge_finder.accessLock().unlock();  //allow thread to continue.
   ```

   Nothing extra needs to be done to wake up EdgeProcessor. It will wake up automatically when it receives new edges.

4. Finally, the main thread puts itself to sleep for a while (maybe for several frames) in order to give the EdgeFinder and EdgeProcesor a chance to find as much as possible.

Next, here is the flow of control through the EdgeFinder thread. It is pretty well described above in EdgeFinder.run(). Basically, while it is awake it searches for edges, according to some heuristic if that is possible, or otherwise by shooting random rays.

It is put to sleep and awoken by the main thread, as described above. Every time it finds an edge, it inserts it in the EdgeProcessor class, and wakes up the EdgeProcessor thread.

The flow of control for the EdgeProcessor is also pretty straightforward. It is described above in EdgeProcessor.run(). The EdgeProcessor is never forced to sleep. It puts itself to sleep as soon as it processes all the available edges. So, during synchronization, the main thread never forces it to stop, but simply waits for it to put itself to sleep.