EUROGRAPHICS 2005 / M. Alexa and J. Marks
(Guest Editors)

Volume 24 (*2005*), *Number 3*

Pen-and-Ink Accelerated Implicit Blobtree Surfaces

Submission id: 403

Abstract

A system for fast rendering of Blobtree implicit surfaces in pen-and-ink styles is presented. Using the idea that small amounts of visual information can convey a detailed subject, we accelerate the rendering process by rendering only the surface's silhouette and interior strokes in principal directions of curvature and in the contour direction. Our system extends previous work that generates the surface's silhouette and tiny interior marks for simple implicit surfaces. We contribute methods to extract interior strokes from the surface suitable for animated primitives, a method to extract silhouette strokes for Blobtree surfaces, and methods to generate several stroke styles found in natural-science illustration.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Line and Curve Generation

1. Introduction

1.1. Goal

The goals of this research are (1) to create accurate natural-science style pen-and-ink images of Blobtree implicit surfaces and (2) to provide an accelerated method to visualize these surfaces which is compatible with animated surfaces.

1.1.1. What did we try to do?

We introduce a new system, the Non-Photorealistic Blobtree (NPRBT) which automatically renders static or animated Blobtree implicit surfaces with styles and techniques used by natural-science pen-and-ink artists. The system renders silhouette and interior strokes and provides support for transparency, various stroke styles and several different stroke placement techniques.

For silhouette extraction, the system extends a method presented by Bremmer and Hughes [1] using a Witkin Heckbert particle system[3] along with an modified extraction method which handles gradient discontinuities and high-frequency portions of the surface. To stylize these strokes, the NPRBT simulates various stroke styles using OpenGL primitives. For interior stroke extraction, the system extends methods presented by Elber[2] and Akleman[**?**]. Particles are passed to a selection pipeline which determines the particles that receive strokes. Then, the system creates stylized curved strokes along the surface.

1.1.2. Who would benefit?

This system benefits two groups of people. First of all, it is useful for modelers, animators and designers who require a program capable of interactive visualizations because it provides a fast interactive rendering of moderately complicated implicit Blobtree surfaces. Secondly, pen-and-ink renderings of implicit surfaces benefit any situation where a pen-and ink rendering of an object is required, such as in drafting, technical natural-science illustration or cartoon illustration.

1.2. Previous Work

There are three main works that address silhouette and interior stroke extraction and stylization for implicit surfaces[1, **?**, 2] in an NPR context.

Bremer and Hughes[1] use a hybrid ray-tracer/integration/OpenGL system to extract silhouettes and some interior strokes for pen-and-ink simulation. To do this, their system creates random rays from the scene's eye and collides them with the implicit surface. For successful intersections, a small interior mark is placed on the surface. To extract the silhouette, the system uses an integrator find the silhouette of the surface from this point. Finally, silhouette strokes are extracted using another integration method which estimates the silhouette direction. Finally, visible portions of the silhouettes are rendered in an ink-line style with varying width based on curvature.

Elber[2] presents a system which renders short strokes on

the interior of the surface for another pen-and-ink style. To accomplish this, his system distributes particles on the surface, evaluates shape measures and then extracts strokes directly from the particles. These strokes can be grouped and directed in various orientations to convincingly convey details of the surface. A variety of stroke types, such as scribbles, straight lines and jagged lines, can be applied to the particles and his system allows for transparency. Although this system doesn't specifically extract long silhouette strokes, particles can be grouped near the silhouette of a surface for a similar effect.

Akleman[**?**] extracts long flowing strokes along the interior of the implicit surface to create a painting style. His system uses particles similar to Elber's in that they adhere to and move across the surface. In this case, however, the positions of the particles are saved as the particles move across the surface and long strokes are created from these lists of points during rendering.

1.3. Approach

1.3.1. What approach did we try?

Our algorithm takes an input an implicit Blobtree model [Cite Papers] and uses particles as a base to create interior and silhouette strokes. The algorithm works completely in object space and is organized as follows:

1. Particles move towards a desired position based on user-input, for example to areas of high convex curvature.
2. Silhouette strokes are extracted
3. Silhouette strokes are stylized and hidden-line removal is performed.
4. Interior strokes are extracted
5. Interior strokes are stylized and hidden-line removal is performed.

Silhouette extraction is based on the method from Bremmer and Hughes[1]. Our version has been modified to use data from the particles to cope with high-frequency changes in the surface created by various primitives, warps and CSG operations. Furthermore our method requires less computation to operate and is thus faster. Interior stroke extraction is loosely based on Elber's[2] approach and attempts to create accurate, realistic strokes in the fastest time possible directly from the particles.

1.3.2. Under what circumstances do we think it should work well?

1.3.3. Why do we think it should work well under those circumstances?

2. Methodology

2.1. What pieces had to be implemented to execute my approach?

The NPRBT system separates stroke creation into two systems: the (1) the *Silhouette Stroke System* and (2) the *Interior Stroke System*. Furthermore, both of these systems rely on a third underlying system, the particle-based *Stroke Placement System* for data. For silhouettes, the *Stroke Placement System* provides start positions to trace out the silhouette and spatial information to help this extraction. For interior strokes, the particles in the Stroke Placement system are used to directly position and stylize interior strokes. The *Silhouette Stroke System* extracts the silhouettes, determines visibility and renders the silhouettes in natural-science pen-and-ink styles. The *Interior Stroke System* selects which particles from the Stroke Placement System will be used to create strokes, determines visibility and stylizes the strokes.

3. The Stroke Placement System

The NPRBT *Stroke Positioning System* uses a modified Witkin-Heckbert[3] particle system to distribute particles on an implicit surface. Witkin and Heckbert present a system where certain particles (floaters) space themselves uniformly on an implicit surface. Our system uses these floaters and adds several new repulsion methods, used to create interesting groupings of interior strokes (Section **??**) and new "anchor" particles used for Silhouette and CSG strokes.

3.1. Were there several possible implementations?

Another possible approach as a base for strokes is Bremmer and Hughes'[1] approach, where, for each frame rendered, ray intersections are used to create interior strokes and to extract the silhouette.

3.2. If there were several possibilities, what were the advantages/disadvantages of each?

The main drawbacks for the ray intersection method are (1) it does not produce frame-coherent strokes, (2) it is much more expensive than the particle system approach and (3) it is impossible to get a specific distribution of strokes with this method (such as grouping strokes in areas of convex curvature). Frame coherency is a problem because the rays cast for each frame rendered are completely random, and thus the strokes created from the intersection points move randomly from frame to frame. This approach is much more expensive than the particle system approach because ray intersections with implicit surfaces are very expensive, especially compared to the amount of computation required for each particle from frame to frame.

3.3. Which implementation(s) did we do? Why?

We tested both the ray intersection approach and the particle system approach. The particle system was chosen as the base for the NPRBT system because it requires much less computation and provides a perfect base to create coherent

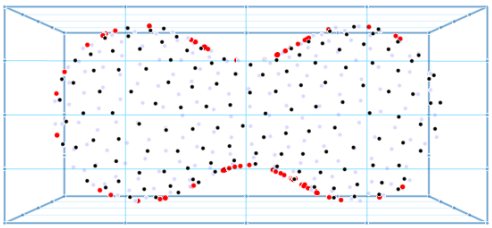

Figure 1: *The NPRBT Stroke Positioning System uses particles on the implicit surface as a base to create strokes. Black particles are on the front-facing side of the surface, light grey particles are on the rear-facing side of the surface and red particles are silhouette particles. A grid is used to spatially partition the particles and accelerate the speed of the repulsion equation. In this image, the volume of the grid is the light blue box surrounding the particles.*

silhouette and interior strokes for regular and animated surfaces. The particles maintain their relative position over successive frames of animation and therefore the strokes created are frame-coherent.

3.4. What did we implement? – Include detailed descriptions

The Stroke Placement System first performs a particle initialization preprocess to place a small number of particles on the surface. Then, for each frame rendered, two "forces" move the particles into one of several orientations. As this occurs, the system also saves information about the surface into an octtree grid and creates static *silhouette* particles and *CSG* particles.

3.4.1. Initialization Preprocess

During initialization, particles are created at random positions on the implicit surface. There are several methods to create particles on the surface. Our system [write current method here].

[TODO]

3.4.2. Runtime

Once particles are initialized on the surface, the system renders strokes from the particles (Section **??**) and distributes the particles on the surface. There are several reasons to render strokes from the particles as they distribute themselves. For animated surfaces, it ensures that the particles will provide frame-coherent strokes and remain in the proper distribution as the surface changes. Furthermore, since it can take a moment to properly place particles in a perfect configuration, this loop provides **feedback** as to the status of the system. Since the particles are always on the surface from initialization, this feedback provides a proper impression of the surface, although perhaps not in a uniform distribution.

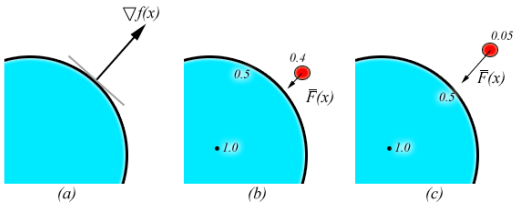

Figure 2: *This simple 2D example illustrates how $\bar{F}(x)$ keeps the particles on the surface using the gradient. (a) $\nabla f(x)$, the gradient, is a vector pointing directly away from the surface. (b,c) The value of the implicit function at the particle position can be used to guess the distance of the particle to the surface and to scale $\bar{F}(x)$, calculated in equation 2.*

Particle motion is governed based on two "*forces*": (1) $\bar{F}(x)$, a force that pulls particles towards the surface along the gradient and (2) $\bar{R}(x)$, a force which repulses particles apart. These are included in the following equation, which is executed for each particle in the system for each frame of animation:

$$\bar{\mathbf{x}} += S_{speed} * (S_{surface} * \bar{F}(\mathbf{x}) + S_{repel} * \bar{R}(\mathbf{x})) \quad (1)$$

where $\mathbf{x}$ is the particle's position and the S values are all user-controlled scalars. $\bar{F}(x)$ is calculated from:

$$\bar{F}(\mathbf{x}) = (f(\mathbf{x}) - iso) * \nabla f(\mathbf{x}) \quad (2)$$

where $f(\mathbf{x})$ is the implicit function, *iso* is the iso-surface value and $\nabla f(\mathbf{x})$ is the gradient. This equation will create a vector pointing directly towards the surface (unless the particle is exactly on the surface). Furthermore, note that $\bar{F}(x)$ is scaled by the difference between the field-value and the iso-value of the surface, an estimate of how far the particle is from the surface.

To space the particles on the surface, $\bar{R}(x)$ is used (Equations 3 to **??**). This force is calculated by assuming all neighboring particles in any given particle's vicinity contribute to repulsion based on their distance. This is accomplished by calculating the vector from each neighboring particle to the current particle, normalizing it and scaling it by distance with the same method used to calculate a magnetic repulsion force. If this is done uniformly, the particles will distribute themselves evenly. For a system with n particles where structure P holds these particles, this is calculated with the following summation:

$$\bar{M}(\mathbf{x}) = \sum_{k=1}^{n} \left(\frac{maxDist - \|\mathbf{x} - P_k.\mathbf{x}\|}{maxDist}\right) * \frac{\mathbf{x} - P_k.\mathbf{x}}{\|x - P_k.\mathbf{x}\|} * P_k.dis \quad (3)$$

where *maxDist* is the maximum distance of particles that are included in the summation and $P_k.dis$ is a scalar used to control particle distribution. If an even particle distribution is desired, this value is not used, and the repulsion will be equal from all particles. To place more particles in areas of high-curvature, the following scalar is used for P_k:

$$P_k.dis = 1.0 - P_k.curvature/maxCurvature \quad (4)$$

where *maxCurvature* is the maximum curvature of all particles. This formula will create values from 0.0, where particles have the maximum curvature, up to 1.0, where particles are at a position on the surface that has no curvature. Artists often vary application of strokes based on the distance of the surface to the eye (depth-cues). This can be simulated, alternately with:

$$P_k.dis = \frac{\|P_k.\mathbf{x} - eye\|}{maxLength} \quad (5)$$

where *maxLength* is the maximum depth into the scene. This formula will create values from 0.0 to 1.0 as particles increase in distance from the eye. Thus, as particles are deeper in the scene, they (and the strokes that are created from them) will space themselves out further.

Different particle distributions are desirable because the particles are used directly to create strokes (Section **??**) and artists often create groups of strokes with non-even distributions.

Unless all the particles included in equation 3 are co-planar, the vector $\bar{M}(x)$ will point off the surface. This can push particles off the surface in areas of high curvature. To prevent this, we adjust the vector $\bar{M}(x)$ to be orthogonal to the gradient:

$$\bar{R}(x) = \nabla f(x) \times (\bar{M}(x) \times \nabla f(x)) \quad (6)$$

3.4.3. Particle Creation and Destruction

To finds a stable particle layout quickly, particles are created and destroyed. When the repulsion equation 3 returns a value lower than the *creation-threshold* for a particle, there are very few other particles in the vicinity. In this case the particle is duplicated. Conversely, when equation 3 returns a very high value for a particle, there is a large number of particles in the region and the system deletes the particle. These operations decrease the amount of time that it takes for the particle system to become stable.

3.4.4. Silhouette and CSG Particles

The particle system also maintains a list of particles that are on the silhouette of the surface and on CSG discontinuities.

Silhouette particles are created whenever a regular particle crosses the silhouette. This can be done by checking the angle between the view direction from the eye to the particle and the gradient of the surface at the particle's position (Figure **??**a). When this angle between these is about 90 degrees, the particle is on the silhouette. This can be tested using the dot product operation, which returns 0 when the angle between two vectors is 90 degrees. Thus, the NPRBT system checks for when the result of the dot product operation switches sign. When this happens the particle's position is used to create a new silhouette particle. These *silhouette* particles are never altered or moved until the silhouette of the surface changes (when the surface moves during an animation or when the program user rotates or moves the surface). In this case, the system attempts to move the silhouette particles back onto the silhouette using the following:

$$\bar{\mathbf{x}} += S_{speed} * (S_{surface} * \bar{F}(\mathbf{x}) + S_{silhouette}\bar{SL}(\mathbf{x})) \quad (7)$$

$$\bar{SL}(x) = (\nabla f(x) \cdot (\mathbf{x} - eye)) * \nabla f(x) \times ((\mathbf{x} - eye) \times \nabla f(x)) \quad (8)$$

If the particle does not find the silhouette after 10 iterations, the silhouette particle is discarded. Of course, new silhouette particles are found for each view so this step is not necessarily required. In our implementation, our goal is as many silhouette particles as possible so that the chance of missing a silhouette is minimized. Our system usually finds many silhouette particles on a single silhouette for each viewing angle.

CSG particles are again created from regular particles. In this case, however, we are looking for the position of gradient discontinuities. To do this, the system compares the gradients of the surface at the particle's current and previous position. If the difference between the two gradients is larger than some threshold, we assert that the particle has crossed a CSG discontinuity (Figure **??**b). Then, the CSG particle is created directly between these two positions. These CSG particles are never altered or moved unless the surface immediately around the CSG join moves in which case the points will no longer be on the CSG operation and they are deleted.

3.5. What didn't we implement? Why not?

Your text here...

4. The Silhouette Stroke System

The NPRBT system creates long stroke chains which follow the silhouette (contour) of the implicit surface. To extract these silhouette chains, (Section 4.4.1) a version of Bremer and Hughes'[1] method is computationally simpler and it is extended to handle high-frequency areas and CSG discontinuities on the surface. Then, (Section 4.4.5) the silhouette

chains are stylized using one of several styles and ink application methods.

4.1. Were there several possible implementations?

Our approach to render silhouettes is to extract complete stroke chains from the surface. It is also possible to render silhouettes using individual strokes as is demonstrated by Elber[2].

4.2. If there were several possibilities, what were the advantages/disadvantages of each?

In terms of stylization and control, it is possible to create all sorts of effects with both methods to render strokes. However, with use of single silhouette chains, a neater result can be produced.

4.3. Which implementation(s) did we do? Why?

Both methods are implemented in our system, as demonstrated in Figure **??**, but we recommend using the Silhouette Stroke System instead of getting the interior stroke system to group strokes at the silhouette.

4.4. What did we implement? – Include detailed descriptions

The pipeline for the system is presented in Figure **??**. When the system finds a new silhouette particle that does not belong to a previously extracted silhouette stroke chain, the stroke extraction begins. Once this is complete, Hidden Line Removal is performed and shape measures used with rendering methods to stylize the strokes.

4.4.1. Silhouette Extraction

Blending implicit surfaces are well suited for silhouettes because, unlike polygonal meshes, a continuous path exists on the surface that the exact silhouette follows. Bremer and Hughes[1] take advantage of this by using the gradient of the surface to step along the curve of the silhouette. This approach does not work directly for Blobtree implicit surfaces because discontinuities exist at CSG boundaries on the surface. However, as we present, slight modifications to Bremmer and Hughes' method can handle these situations.

Observe that a curve c lies on the silhouette of an implicit surface S if the following hold[†]:

1. $c(t) \in S$ for every t
2. the tangent plane to the surface at $c(t)$ contains v, the view direction, for every t

[†] Bremer and Hughes[1]

In other words, to be a silhouette curve for the surface, it must lie on the surface (at value *iso*) and the direction from the eye to the point must be orthogonal to the gradient (the definition of a silhouette):

$$f(c(t)) = iso \tag{9}$$

$$\nabla f(c(t)) \cdot v = 0.0 \tag{10}$$

The complete silhouette extraction method, detailed in Figure 3, starts from a silhouette particle and follows a function based on the conditions in equations 9 and 10 along the silhouette. As each new point is calculated, it is stored in a chain data structure (section 4.4.3). Termination occurs when (1) the extraction meets the original starting point (meaning that a looping silhouette has been found) or (2) the stroke trace exceeds the maximum length (this is provided as a last minute out in an error case) (Figure 3). The first time the trace hits an existing stroke or the silhouette disappears due to local geometry, the system returns to the original starting point and traces the silhouette in the other direction. The second time this happens, the loop terminates. Note that the silhouette is traced, even if it is occluded by some closer part of the surface. This is done to ensure that the entire silhouette is traced and to facilitate transparency (section **??**).

The system presented here is modified slightly from Bremer and Hughes'[1] approach. They require a ray-surface intersection test, an integration loop to find a point on the silhouette and another integration loop to trace around the silhouette for each frame of animation. The NPRBT does not require the first two steps because the *Stroke Placement System* provides points on the silhouette (section 3.4.4). Furthermore, our silhouette extraction method is altered slightly, as is now detailed.

4.4.2. Estimating the Silhouette Direction

The method to calculate the direction along the silhouette requires three parts: an initial estimate of the silhouette direction, and two correction steps that adjust the direction. Without these correctors, the trace loses the silhouette. The initial estimate is based on the idea in formulas 9 and 10, that the silhouette should remain on the surface and follow the direction where the angle between the view direction and the gradient are ninety degrees (figure **??**):

$$D(x) = (\nabla f(x) \times (x - eye)) \tag{11}$$

where x is the current position. $D(x)$ approximates the direction of the tangent to the gradient at point x (Figure 4a).

As is illustrated in Figure 4a, if this direction is used alone to trace the stroke, the the surface and the silhouette will slowly be lost. To compensate for these problems, the initial direction is modified by adding two "correctors" (figure

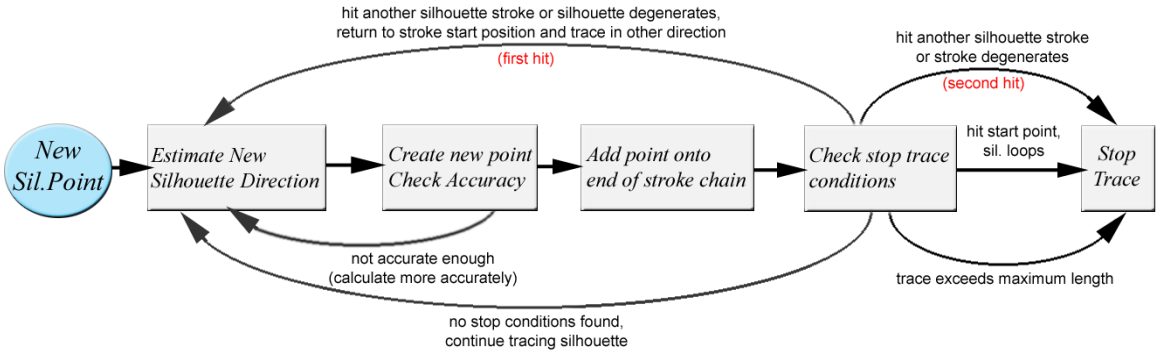

Figure 3: *A high-level view of silhouette extraction.*

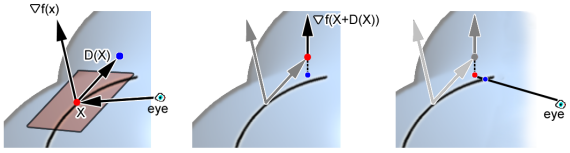

Figure 4: *Left: Calculation of the new point before the correctors (equation 11). Middle: Using the field value and new gradient $\nabla f(x+D(x))$ to keep the trace on the surface (equation 13). Right: Using the new view direction to keep the trace on the silhouette (equation 14).*

4b,c) to keep the trace following the silhouette. Before calculation of these correctors takes place, a new position is calculated from equation 11:

$$\dot{x} = x + k_{stepsize} * D(x) \tag{12}$$

where $k_{stepsize}$ is a scalar value specifying the magnitude of the step that the silhouette tracing algorithm takes. With this calculated, the first corrector adjusts this point closer to the surface:

$$D_{surface}(\dot{x}) = \nabla f(\dot{x}) * (0.5 - f(\dot{x})) \tag{13}$$

Note the similarity with this equation and equation 2 which keeps particles in the *Stroke Placement System* on the surface. The field value of the surface is used to scale the size of the step taken back towards the surface along the gradient. The gradient direction is used because it is the best quick estimate of the direction to the closest point on surface. As the trace progresses, this corrector will constantly pull the trace back onto the surface, although it will not put the point exactly on the surface. The second corrector keeps the stroke following the silhouette:

$$D_{silhouette}(\dot{x}) = (D(x) \times \nabla f(\dot{x})) * (view \cdot \nabla f(\dot{x})) \tag{14}$$

where $view = \dot{x} - eye$. As the silhouette trace moves off the silhouette, the magnitude of $view \cdot \nabla f(x + D(x))$ will increase to pull the stroke back towards silhouette. This is slightly modified from Bremmer and Hughes'[1] approach. In their method, a different direction, based on the Hessian is used to pull $\dot{x}$ back to the silhouette. Hessian evaluation requires 12 surface-evaluation tests. Our version only uses the gradient, requiring four surface-evaluation tests. The efficiency of an algorithm is dependant on the number of times that the surface-evaluation test is called.

Once $D(x)$, $D_{surface}(x)$ and $D_{silhouette}(x)$ have been calculated, the new chain position is calculated by their sum (the process of adding the vectors is illustrated in figure 4):

$$X_{new} = k_{stepsize} * D(X) + k_{surface} * D_{surface}(X) + k_{silhouette} * D_{silhouette}(X) \tag{15}$$

where $k_{stepsize}$, $k_{surface}$ and $k_{silhouette}$ are user-selectable scalars. A discussion of these is provided in section 4.4.4.

4.4.3. Chains

As the silhouette is extracted, points are stored in a "chain", a datastructure that stores points so that, during stroke rendering and stylization, each link (an element in the chain) is drawn joined to its left and right neighbors. The contents of the NPRBT's chain data structure is illustrated in figure **??**. Each link contains a *position*, a *plot_direction* vector (specifying the direction to widen the stroke, section 4.4.6), a *visible* boolean (specifying whether the link is visible or not) and a *plot_width* scalar (specifying the width of the stroke at the link). This width is interpolated between links during rendering.

4.4.4. Integration Step size and Dealing with High-Frequency Areas and CSG Discontinuities

4.4.5. Silhouette Stylization and Rendering

The *NPRBT Stroke Stylization System* renders strokes chains in a variety of styles in 3D in OpenGL. To complete this,

the system uses (4.4.6) a stylization module which simulates several styles typical to pen-and-ink rendering. This module uses information about (4.4.8) stroke visibility and (**??**) **shape measures** to further stylize strokes and simulate pen-and-ink rendering.

4.4.6. Stroke Volumes

In the NPRBT system, silhouette strokes vary in width and simulate various natural-science stroke styles. This is accomplished using stroke *ribbons*: a varying-width quad-strip which follows the stroke path. First, the method to calculate these quad-strips will be provided. Then, methods to use OpenGL primitives such as GL_LINE, GL_TRIANGLE and GL_POINT in these volumes to simulate different artistic styles will be presented.

To create the quad strip rendering-area, each links contributes two points: it's position (called a) and another point projected out from this position in the direction of the gradient, scaled by either curvature or depth (called a'). With *scale* as the scalar value to size the displacement (Section 4.4.9), the following is used to create a':

$$a' = a + \nabla f(a) * scale \tag{16}$$

Each quad is created from the points from a link (a, a') and it's neighbor (b, b') for each pair of links in the chain (Figure 5top-left).

4.4.7. Rendering ink to the Volumes

The NPRBT system simulates several natural-science rendering styles often used by artists simulate texture, to provide information about the object they are illustrating or to create a certain feel in the image. This is accomplished using a similar method to Sousa et al.[**?**] where OpenGL line, point and polygon primitives into the stroke volumes (a, a', b, b') . Our method is modified for silhouette strokes and more styles are presented (including transparent effects).

The styles that our system creates are ink-filled, serrated, variable-serrated, broken-serrated, double-line and stipple (Figure 6). The intensity of these strokes is constant (black). For transparency effects, our system provides a dotted-line style and can change the colour of occluded strokes to light grey or red.

The ink-filled style and the double-line (Figure 6a,f) are the simplest. The ink-filed style is made simply by drawing a black QUAD with points (a, a', b, b') (Figure 5). The double line style is created using two line-strips: one that follows the (a) points (the position values) and one that follows the (a') points (Formula 16) along the links.

The stipple and serrated edge styles (Figure 6b-e) require a slightly more involved rendering as the serrated edge styles

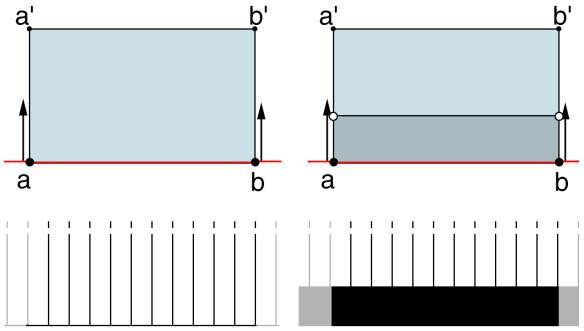

Figure 5: ***Top:*** *The (a, a', b, b') and the (a^*, a', b^*, b') volume.* ***Bottom:*** *Applying the quality-stipple style to the volumes.*

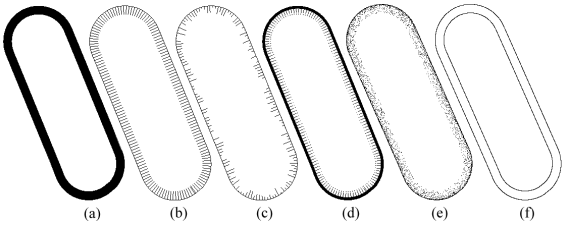

Figure 6: *The stroke styles simulated in this system. From left to right: ink-filled, serrated-edge, serrated-edge random, serrated-edge with ink-filled, stippling and double line.*

uses many lines parallel to the stroke direction and stippling uses many random dots in the stroke-volume. For these styles, an outline is always rendered. The fastest method in our system to draw an outline is to use a line-strip primitive following the (a) points in the chain (Figure 6b,c,e). However, artists often vary the width of the stroke using serrated styles, so another option is to use a render a quad into part of the edge of (a, a', b, b') area (Figure 6d). In our system, we calculate $a^* = \frac{2}{3}a + \frac{1}{3}a'$ and use that point to create the quad instead of a'. Then, the serrated edge lines or stipple dots are drawn into the area (a^*, a', b^*, b') (Figure 5top-right).

To create the non-random serrated lines (Figure 6b,c), the system interpolates two points along the line (a^*, b^*) or (a, b) if only line primitives are used at the silhouette. The user provides the resolution *res* as to how many serrated edges are desired per quad and the following algorithm is used:

SERRATED-STROKE(a, b, a', b', res)
1 $res \leftarrow res * (|ab|/l_{max})$
2 $\delta_a \leftarrow a$
3 $\delta_{a'} \leftarrow a'$
4 $\gamma_a \leftarrow (b-a)/res$
5 $\gamma_{a'} \leftarrow (b'-a')/res$
6 $step \leftarrow 0$
7 **for** $t \leftarrow 0$ *to* 1, $step += 1/res$
8 **do** $\delta_a += \gamma_a$

```
9        δ_{a'} += γ_{a'}
10       DrawLine(δ_a, δ_{a'})
```

The stipple (Figure 6) and the random-serrated style (Figure 6) are both created by introducing randomness into the algorithm, and interpolating between the *plot_direction* values for points (a) and (b) instead. With n^a as the plot direction from a link in the chain and n^b as the plot direction for its neighbor, the following algorithm is used to create the random-serrated style:

```
RANDOM-SERRATED(a, b, n^a, n^b, res)
1   res ← res * (|ab|/l_max)
2   δ_a ← a
3   γ_a ← (b − a)/res
4   δ_{n^a} ← n^a
5   γ_{n^a} ← (n^b − n^a)/res
6   step ← 0
7   for t ← 0 to 1, step += 1/res
8      do δ_a += γ_a
9         δ_{n^a} += γ_{n^a}
10        DrawLine(δ_a, δ_a + rand() * delta_{n^a})
```

where rand() returns a random value between 0 and 1. To create stipple marks, we replace line X with $DrawPoint(\delta_a + crand() * delta_{n^a})$ where crand returns $(1-rand)^2$. This creates a high density of points at the silhouette which falls off as one moves to the inside of the stroke.

Returning to the standard serrated-edge style, artists sometimes break the serrated stroke near its tip. We simulate this using the following algorithm:

```
SERRATED-STROKE(a, b, a', b', res)
1   res ← res * (|ab|/l_max)
2   δ_a ← a
3   δ_i ← 0.2 * a + 0.8 * a^prime
4   δ_j ← 0.1 * a + 0.9 * a^prima
5   δ_{a'} ← a'
6   γ_a ← (b − a)/res
7   γ_{a'} ← (b' − a')/res
8   step ← 0
9   for t ← 0 to 1, step += 1/res
10     do δ_a += γ_a
11        δ_i += 0.2 * γa + 0.8 * γa'
12        δ_j += 0.1 * γa + 0.9 * γa'
13        δ_{a'} += γ_{a'}
14        DrawLine(δ_a, δi)
15        DrawLine(δj, δ_{a'})
```

For non-looping chains, the beginning and the end of the stroke are scaled down in width.

4.4.8. Hidden line removal

The NPRBT system does not render surface geometry. Thus, there is nothing in the Z-Buffer to automatically occlude strokes blocked by a closer part of the surface. Thus, a Hidden Line Removal (**HLR**) method is required to remove these lines.

For silhouette HLR, the NPRBT system casts a ray from every fourth element in the silhouette chain to the eye and checks for collisions with the surface. Note that the system does not perform a complete surface collision because the problem is not to find the *exact* collision point, it is simply a question of if there is a collision or not. If a collision is detected, this part of the chain is occluded. The system repeats the intersection test for the previous three points in the chain to find the exact edge where visibility changes. From this point, links are flagged as *occluded* until the system finds the point where the stroke becomes visible again, done again by checking every fourth point. The system only checks every fourth link because collision tests for Blobtree surfaces are computationally expensive.

When part of a stroke is occluded, the system can removes that portion from the rendering pipeline, as is done for most of the images in this paper, or it can apply a transparent style to the stroke. Transparent styles (Figure **??**) include (1) a dotted line, created by rendering small quads into the system's stroke-volumes, or (2) a lighter intensity of stroke (grey) or (3) a red stroke.

4.4.9. Silhouette Stroke Width

Like Bremmer and Hughes' system[1], the NPRBT system varies stroke-width using the curvature at the points in the stroke or using the depth of the stroke. Use of these measures directly produces the inadequate results seen in Figure 7a; strokes almost disappear where the measure returns a low value and strokes become too thick in areas of high-curvature.

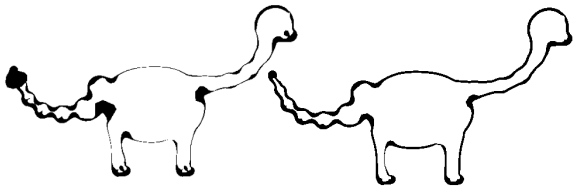

Figure 7: *Left: Using exact curvature values to vary the width of a stroke on the dinosaur model. Right: Bounding extremely low and high curvature values and mapping the remaining values to an ease-in/ease-out function.*

To process the values to be more suitable, the system applies an upper and lower-bound to all of the values and then scales them between user-specified min and max stroke width values. In the case of curvature, we observed that [fill in AUGH too late to type it now hehe]

the system uses an ease-in/ease out function, $f(t) = -2t^2 + 3t$, before mapping to the min and max values. effect

of applying an ease-in/ease-out function is that the curvature varies smoothly from the low-bound to the high-bound (figure 7right, section 4.4.6).

4.5. What didn't we implement? Why not?

Your text here...

5. The Interior Stroke System

The NPRBT system creates short strokes across the interior of the surface in several styles. These strokes are created in the position of certain particles from the *Stroke Positioning System*, selected based on shape-measures, and are directed in the contour direction and the principal directions of curvature. No integration-based extraction method, as used for silhouette strokes, is used to trace interior strokes so that strokes can be rendered and displayed quickly. Finally, the system can curve strokes based on local curvature information to improve the quality of the stroke.

5.1. Were there several possible implementations?

Your text here...

5.2. If there were several possibilities, what were the advantages/disadvantages of each?

Your text here...

5.3. Which implementation(s) did we do? Why?

Your text here...

5.4. What did we implement? – Include detailed descriptions

We implemented a (5.4.1) particle selection module (Figure 8a-d) which determines where to draw strokes and a (5.4.2) stroke stylization module (figure 8e) which points these strokes in certain directions, sets their length based on shape measures and curves the strokes to follow the surface.

5.4.1. Particle Selection

The NPRBT Interior Stroke System provides a pipeline to select the particles that will be used to create strokes (Figure 8) that is evaluated for each frame rendered to handle animated primitives and to display strokes as the particles settle on the surface after initialization. The system first removes occluded particles and then it removes certain remaining particles, based on shape-measures, from stroke inclusion. This simulates shading and focus styles commonly used by artists.

Occluded Particle Removal

To remove occluded particles, a two-step method is employed. The failsafe method to remove hidden particles is to preform a ray-intersection test, however this test is computationally expensive. Fortunately a simple test can remove many hidden particles before relying on the ray-intersection test. All particles that are back-facing are occluded and these can easily be identified by comparing their gradient to the view direction. If the angle between the two is less than 90 degrees, the particle is on the far side of the surface and thus is removed from the list of particles to render.

Once back-facing particles have been removed, the remaining particles must be checked with the ray-surface intersection to see if they are occluded by a closer surface. If a collision occurs, some part of the surface is between the eye and the particle and that particle is occluded and removed from rendering.

The NPRBT system uses a shortcut to cut down the number of occlusion tests. For each frame rendered to the screen, only one (random) particle in each grid cell (Section **??**) is tested for occlusion (this particle is henceforth called the *test-particle*). Once this is complete for all cells, the system determines visibility for the particles in any one cell by analyzing the test particles from the six cells immediately around the current cell and the current cell itself and takes one of three actions. (1) When all test-particles in the immediate cells are visible, all particles in the current cell are set to visible and no further tests are performed. (2) When all of the test-particles in the immediate cells are occluded, all particles in the current cell are considered occluded and removed from the rendering pipeline. (3) When the immediate test-particles are both occluded and visible, each particle in the current cell is tested for occlusion. The rationale for this is that these particles must be close to a boundary of an occluding surface and should each be tested to maintain accuracy.

Remaining Particle Selection

Artists commonly choose to create many strokes in certain regions of the surface and leave other regions void of strokes [references TODO]. These artists commonly use shape measures, such as curvature and slope-steepness, or lighting and depth information to position strokes as these help viewers perceive the surface. To bridge regions with strokes and regions without, artists blend the edges of regions-they lower the number of strokes on the edges of a high stroke density area to a sparse placement in a low stroke density area. The rate at which this changes varies depending on the artist, the subject and its application.

We present a simple method to accomplish this for implicit surfaces. The system removes particles from rendering in certain areas, renders particles in others, and blends the amount of particles rendered between these two areas (figure **??**). The system can use curvature, depth (distance of point from eye) and lighting (angle between a point-light

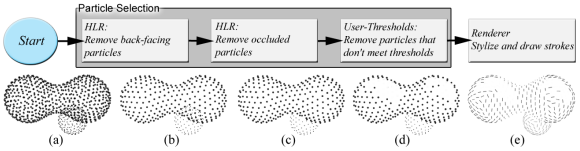

Figure 8: *Top: The interior stroke extraction system evaluates particles for visibility (HLR) and for certain shape measures to determine which particles get strokes applied to them. Bottom: (a) All particles in a sample scene, (b) removing back facing particles, (c) removing occluded particles, (d) removing particles based on a user-selected measure, (e) stylizing small marks in the principal direction of curvature on the surface.*

and the gradient) for this step. To accomplish this, an upper-threshold *ut* and a lower-threshold *lt* are used in conjunction with one of the five *measures*. Any particle whose *measure* is less than *lt* is removed from rendering and any particle whose measure is greater than *ut* is rendered. Particles in the areas between *ut* and *lt* are included and excluded so that the resulting strokes blend between the two areas (figure **??**).

The system uses the particle's index to create a pseudo-random number and then compares this value with its shape measure to determine whether the particle will be included. The particle's index is used to create a random number instead of a standard random number generator because the system redraws the particles for each step of the system's main loop and thus, completely random numbers would cause individual particles to randomly be included or excluded for rendering and create flashing particles. Using the particle's number solves this because the position of a particle *n* will be random (section 3.4.1). To determine this random value, the following formula is used:

$$et = \frac{n}{number\_of\_particles} * (ut - lt) + lt \qquad (17)$$

where *n* is the number of the particle. This equation calculates value *et* which is scaled to be between *lt* and *ut*.

If the selected shape measure is greater than *et* at particle *n*'s position, the particle will be included for conversion into a stroke. This makes it so that particle inclusion is scaled so that all particles are included where the shape measure is *ut* scaled linearly down to no particles being included where their shape measure evaluates to *lt*. For instance, say depth is the selected measure with $ut = 6$ and $lt = 3$ and the depth of a particle is found to be 4.65. If the particle's number is 10 and there are 500 particles in the system, then *et* will be 3.6 (equation 17). Since $4.65 > 3.6$, the particle would be included in rendering. However, if the shape measure is 5.0, the particle would be excluded from rendering because $4.65 <= 5.0$. The higher the shape measure is, the greater the chance that the particle will be converted to a stroke.

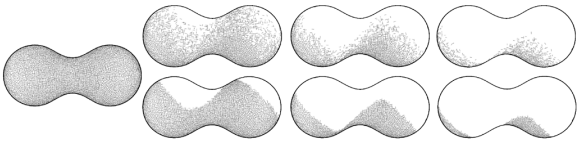

Figure 9: *This figure illustrates different effects of using various values for ut and lt with light direction* $(-1,-1,0)$. *Left: A peanut shape without* ut/lt *thresholding. Top-row: using a large difference between ut and lt creates a smooth blend between areas with strokes and those without. Bottom-row: using close ut and lt values creates a sharper transition between stroke-filled areas and those without.*

5.4.2. Interior Stroke Stylization

The NPRBT Interior Stroke Rendering System creates short directional strokes on the surface at the position of each of the particles selected with the processes described in the previous section. Note that this system does not rely on a functional integration as does the silhouette stroke extraction method to extract long stokes. Instead, the method here creates short strokes using the principal directions of curvature or the contour direction and can curve these strokes based on surface curvature.

Time is critical for particle rendering because it must be performed for each loop of the NPRBT system to display particle motion and to permit for animated primitives.

Stylizing Strokes

The NPRBT system provides a slightly more involved method to create short lines on the surface. The method works as follows: (1) a direction is selected for each rendering particle, (2) this direction is used to create end points for the stroke with intensity and length variation based on shape measures and the option of (2.1) smooth stroke creation based on curvature.

1. Stroke Directions

Lines can be created in three different directions with the NPRBT system: (1) the primary principal direction of curvature (the direction where the surface curves the most, figure 10a), (2) the secondary principal direction of curvature (the

direction where the surface curves the least, figure 10b) and (3) the contour direction (figure 10c). To extract the principal directions of curvature and associated scalar values of curvature, our system extracts the eigen-vectors and eigen-values from the Hessian of the surface as described in [**?**]. To extract the contour direction on the surface, we use:

$$contour(X) = \nabla f(X) \times \frac{(X - eye)}{\|(X - eye)\|} \quad (18)$$

2. Creating the Stroke

The system advances to where the stroke is ready to be rendered. Rendering first involves calculating where the endpoints of the stroke are and, for smoothed strokes, where the control points for Bezier curve generation of the stroke are and executing the Bezier method to create the stroke.

To create a basic line, point calculation involves projecting two points out from the particle position in the stroke direction as illustrated in figure 11a. These points p_1 and p_2 are created with:

$$p_1(X) = X + stroke\_direction(X) * scl \quad (19)$$

$$p_2(X) = X - stroke\_direction(X) * scl \quad (20)$$

where X is the particle's position, $stroke\_direction(X)$ is the stroke direction calculated for position X and scl is a scalar value which determines the length of the stroke. In the NPRBT system, scl can be set by the user, can be based on one of the scalar measures in section **??** or can be a user set-value multiplied by a measure.

The NPRBT system can curve the strokes slightly based on surface curvature to make the strokes adhere better to the surface and to improve the quality of the rendering. To do this, the system creates four *control-points*, c_1, c_2, c_3, c_4 based on the particle's position to be used with a Bezier curve method (figure 11right).

These points are created from p_1 and p_2 along the gradient direction at the particle's position. With a Bezier curve approach, the curve starts at the first control point c_1 and ends at the last control point c_4. For convex areas, c_1 and c_4 are projected negatively along the gradient into the surface with:

$$c_1 = p_1 - (\nabla f(x) * 0.75 * scl * curvature(x)) \quad (21)$$

$$c_4 = p_2 - (\nabla f(x) * 0.75 * scl * curvature(x)) \quad (22)$$

where scl is the distance between p_1 and p_2 and $curvature(x)$ gives the curvature from the Hessian matrix (section **??**). c_2 and c_3 are projected away from the surface along the gradient direction with:

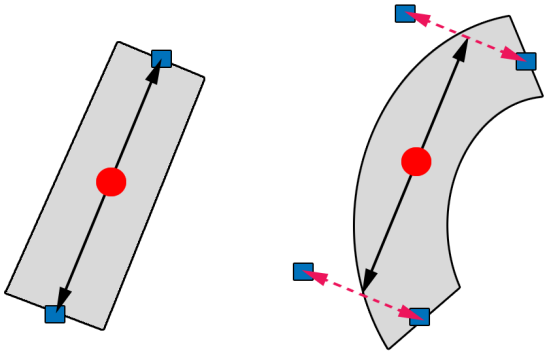

Figure 11: *Left: To create a stroke, the system simply creates two endpoints from a particle's position using one of several direction measures and uses them to create a line. Right: If smoothed strokes are desired, two extra points can be created from the endpoints using the gradient of the surface and all four points can be used as control points for a Bezier curve.*

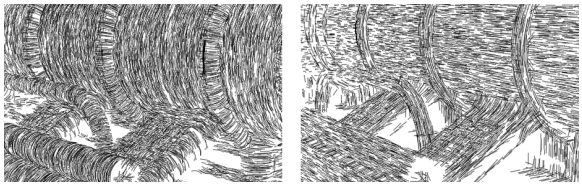

Figure 12: *The body of a train model with strokes applied in (left) the primary principal direction of curvature and (right) the secondary principal direction of curvature. Strokes are curved to follow the surface.*

$$c_2 = p_1 + (\nabla f(x) * 0.25 * scl * curvature(x)) \quad (23)$$

$$c_3 = p_2 + (\nabla f(x) * 0.25 * scl * curvature(x)) \quad (24)$$

Using these control points the stroke will pass through the particle position and will curve outwards with the surface (figure 12). For concave areas, c_1 and c_4 are switched with c_2 and c_3. This will make strokes curve inwards with the surface in concave areas (figure 13).

Note that this value will return positive values in convex areas and negative values in concave areas. Thus, it will automatically scale $p3$ and $p4$ inwards and outwards and will produce flat lines in areas of no curvature.

3. Rendering the stroke

Rendering interior strokes relies on OpenGL calls. The system first sets the width of the stroke (these can be based on the shape measures, section **??**). To do this, the *glLineWidth* functions are used. Then, strokes is rendered using the

GL_LINE_STRIP primitive with the points in the stroke (2 points for straight strokes, more points for curved strokes) (section **??**). A variety of effects can be achieved by varying the width and length of the stroke. Figure 10 shows use of very short, thin strokes. In figure 13, thick long strokes are used to provide a strong impression of the surface.

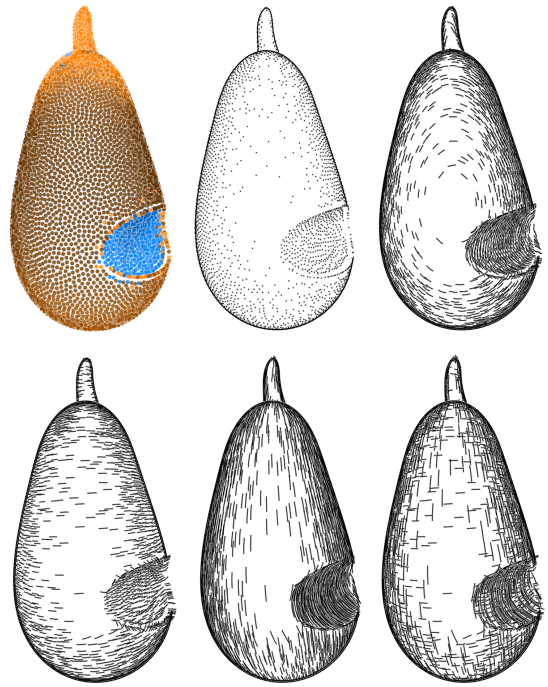

Figure 13: *Creating long, thick strokes onto a pear shape. Top-left: different shades of orange and blue denote different levels of positive and curvature respectively. Top-middle: the stippling style. Top-right: strokes in the contour direction, curved to follow the surface. Bottom-left: strokes in the primary principal direction of curvature. Bottom-middle: strokes in the secondary principal direction of curvature. Bottom-right: strokes in the primary and secondary directions of curvature. Note in the concave "bite" area that strokes are mapped in the opposite direction to follow curvature.*

5.5. What didn't we implement? Why not?

Your text here...

6. Results

6.1. How did we measure success?

Measures for success include visual quality of the images as a whole and the quality of individual strokes and how much speed increase our system gains over other methods.

6.2. What experiments did we execute?

We evaluate the NPRBT system qualitatively and quantitatively. Comments on the quality of the strokes created for several surfaces (Figures **??** to **??**) with test-viewers are provided. Our method does not guarantee that all will be extracted. We test the amount of silhouettes missed with various amounts of particles on the surface to demonstrate that with a minimum coverage, this is not a problem. To demonstrate the speed increase offered by our method, we compare (1) the run-times of Bremmer and Hughes' approach with our silhouette extraction method and (2) the time required to ray-trace and polygonize a surface compared to the time required with our approach for several complicated models (Figures **??** and **??**).

6.3. Visual Results with Timings

Here I will describe the results illustrated in the figures including comments from viewers with timings of the system versus timings from a polygonizer and with a ray-tracer.

6.4. Silhouette Extraction Measure

Our system does not guarantee that all silhouettes will be extracted from the surface. Instead, it relies on an adequate coverage by particles in the surface and accurate silhouette extraction to find all of the silhouettes. For three different models, the simple tooth model, the stalactite and the train model, we now provide the percentage of the complete silhouette that our system extracts, averaged over 256 tests with random viewing angles. To extract guaranteed complete silhouettes, a naive brute-force approach is employed.

6.5. What do my results indicate?

Your text here...

7. Discussion

7.1. Overall, is the approach we took promising?

Your text here...

7.2. What different approach or variant of this approach is better?

Your text here...

7.3. What follow-up work should be done next?

Work should be done comparing different methods to extract silhouette and interior strokes. For example, with a large enough set of particles on the silhouette, an accurate silhouette stroke could be generated by chaining the particles and using subdivision. This method would probably be simpler to implement and might prove to be more efficient or

more accurate. For interior strokes, more methods to extract continuous long strokes over the surface, such as Akleman used[**?**], should be explored for static and animated surfaces. Such strokes were not explored significantly in this work since they take too long to extract, and would slow down visualization for regular and animated surfaces. However, it might be possible to make a "fluid" stroke that maintains its direction over the moving animated surface in a similar method used by the particle system to keep particles on the surface.

7.4. What did we learn by doing this project?

Non-photorealistic rendering not only provides value with the inherit perceptual advantages of a non-photorealistic style, but it can also be used to speed up and simplify rendering implicit surfaces.

8. Conclusion

Your text here...

Acknowledgements

Your text here...

9. Appendices

Curvature

Curvature values can be calculated along with the principal directions of curvature with eigen-value and eigen-vector analysis of the Hessian matrix. These directions and values are useful because artists often use them to provide surface cues [TODO cite paper or two here]. The Hessian is a 3 by 3 matrix which describes the acceleration of the implicit surface. It is built as follows:

$$h(X) = \begin{pmatrix} f'_{xx}(x) & f'_{xy}(x) & f'_{xz}(x) \\ f'_{yx}(x) & f'_{yy}(x) & f'_{yz}(x) \\ f'_{zx}(x) & f'_{zy}(x) & f'_{zz}(x) \end{pmatrix} \quad (25)$$

The eigenvectors of this matrix will be the primary principal direction of curvature, the secondary principal direction of curvature and the gradient of the surface. The corresponding eigenvalues represent the amount of curvature in these directions. The curvature values can be used with *ut* and *lt* (section **??**) to choose which particles will be rendered as illustrated in figure 14. The principal directions of curvature can be used during stylization (section 5.4.2) to create stroke directions.

Calculation of the Hessian requires 3 gradient function evaluations which in turn each require 4 field function evaluations. Although an efficient algorithm is dependant on a low number of implicit function evaluations, curvature information is very useful for stylization.

coming soon

Figure 14: *Left: including particles in more convex areas, $ut = X$ and $lt = X$. Right: including particles in more concave areas, $ut = X$ and $lt = X$.*

References

[1] D. Bremer and J.F. Hughes. Rapid approximate silhouette rendering of implicit surfaces. pages 1–10, 1998. 1, 2, 4, 5, 6, 8

[2] G. Elber. Line art illustrations of parametrix and implicit forms. In *IEEE Transactions on Visualization and Computer Graphics, Vol. 4, No. 1*, pages 71–81, 1998. 1, 2, 5

[3] A.P. Witkin and P.S. Hecknert. Using particles to sample and control implicit surfaces. In *Proceedings Siggraph'94*, pages pp. 269–277, 1994. 1, 2

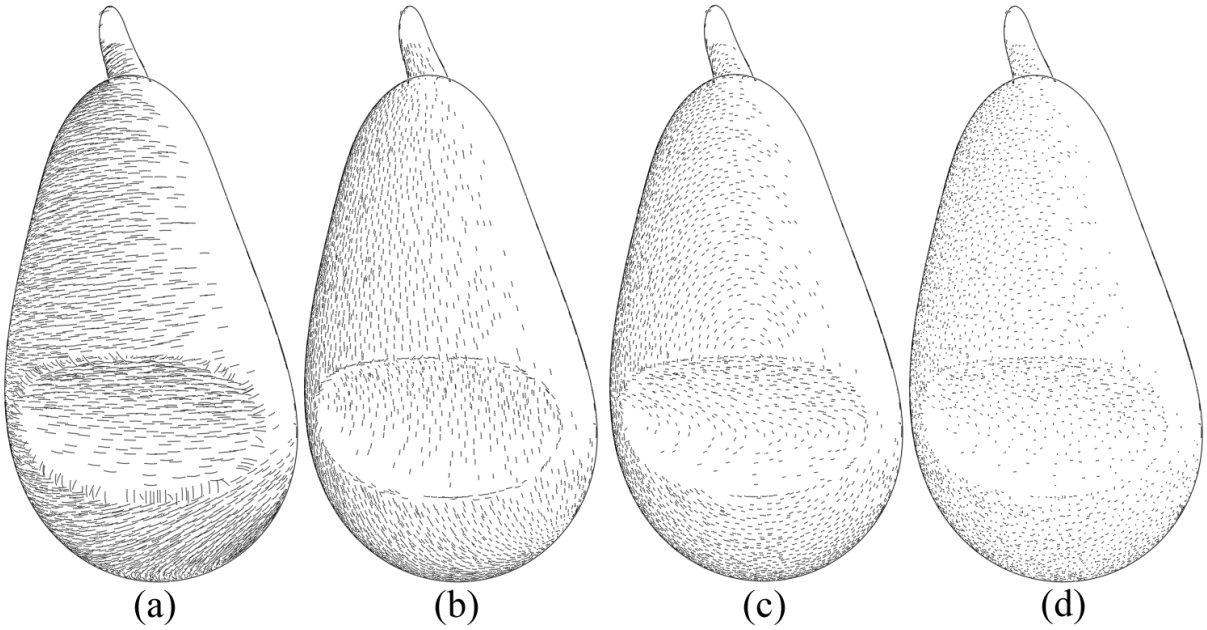

Figure 10: *Short, thin strokes on a pear shape in in the (a) primary principal direction of curvature, (b) the secondary principal direction of curvature, (c) the contour direction, and (d) all three directions. Note in (d) that when extremely short stroke lengths are used, a stippling effect is achieved.*