Technical Report No. 12/2004

Creating High Quality Hatching Illustrations

Johannes Zander Tobias Isenberg Stefan Schlechtweg Thomas Strothotte
jzander@cs.uni-magdeburg.de, {isenberg|stefans|tstr}@isg.cs.uni-magdeburg.de

Department of Simulation and Graphics
Otto-von-Guericke University of Magdeburg, Germany

Abstract

Hatching lines are often used in line illustrations to convey tone and texture of a surface. In this report an approach for rendering hatched line drawings for polygonal meshes is presented. The proposed method allows the generation of hatching lines for interactive on-screen display as well as for vector-based printer output. We use local curvature information to compute streamlines on the surface of the model. A new algorithm ensures here an even distribution of the streamlines such that the visual quality of the final rendition is enhanced. These streamlines form the basis for the calculation of hatching lines. Line shading methods in general are presented that adapt the drawing to lighting conditions. To achieve interactive frame-rates the shading of the hatching lines is calculated at run-time using a virtual machine. For printer output a vector oriented format is generated.

Table of Contents

1. Introduction

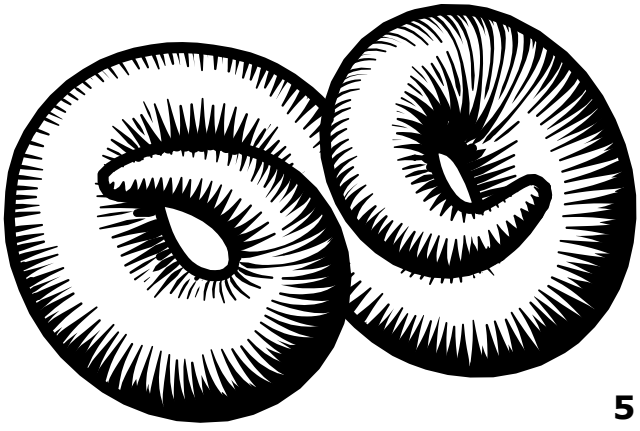
For a long time, illustrations have been an essential part of various types of text. In the early days of printing, illustrations were typically created using the techniques of woodcuts and engraving yielding so-called hatched images. The reason for this is that the printing process is limited to either put ink of a certain color on a specific point of the paper or leaving it empty. The printing process nowadays still has the same limitation. However, halftoning techniques have been developed that allow to simulate shades of a color. Therefore, hatched images are not as widely used anymore. On the other hand, hatched images have one very important advantage over halftoned: hatching lines can be used not only to convey tone but also to depict the texture of a surface.

In recent years, a variety of techniques have been developed to digitally create hatched images from 3D models. However, most of these methods were either not interactive or did not pay special attention to the quality issues that are important in the context of reproduction in print. In this report we present a number of methods that allow to generate hatching lines from polygonal meshes and to render them in high quality. We provide

means that allow either interactive rates for on-screen display or high quality vector output for the reproduction in print. Our approach is based on approximated local curvature information that is integrated to form streamlines on the surface of the mesh. Here, we use a new algorithm that provides an even distribution of these lines. A special processing of these streamlines ensures high quality line rendering for both intended output media later on. While the streamlines are generated in a preprocessing stage, the hatching lines are rendered either for vector-based printer output or on-screen display, the latter allowing for interaction in terms of changing the view parameters or manipulating the entire line shading model at run-time using a virtual machine. This allows a much better control for the designer over the resulting illustration. We also show how to combine several layers of hatching to create cross-hatched images that even can be colorized.

The remainder of the report is organized as follows. To understand the principles used in traditional hatching we examine hand-made hatched illustrations created by artists in Section 2. We extract properties and requirements that will later be used in the design of our algorithms. In Section 3 we review previous work that deals with the rendering of hatched images. Then, Section 4 discusses how to obtain a direction field on the surface of the 3D model and how to optimize it to remove certain artifacts. This direction field is used in Section 5 to derive the streamlines that will later become hatching lines. For this stage we give details for both conceptual and implementation issues. This is followed by a discussion of visualization issues of the streamlines in terms of occlusion and shading in Section 6. Section 7 presents methods for rendering high quality vector lines which is necessary, in particular, for the reproduction in print. We discuss the representation of the line shape including turns, the control of line width and the reproduction of line density using line stippling. Also, we present the two different output modules and illustrate the two techniques using a number of examples. Section 8 then addresses implementation issues in the context of the used framework – OpenNPAR. Section 9 shows more examples created with our system and discusses, in particular, the shading at run-time using an example session with the program. Finally, we conclude our discussion in Section 10 and name a few directions for further research.

The techniques presented in this work were also presented at Eurographics 2004 and published in (Zander et al., 2004).



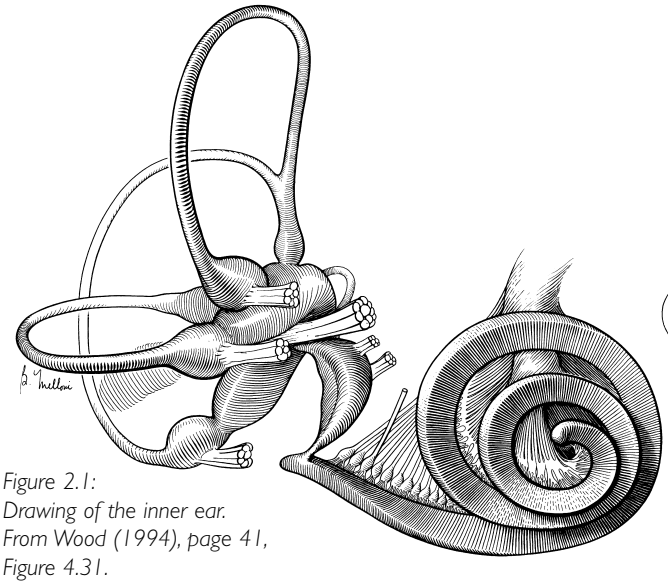


Figure 2.1:
Drawing of the inner ear.
From Wood (1994), page 41, Figure 4.31.

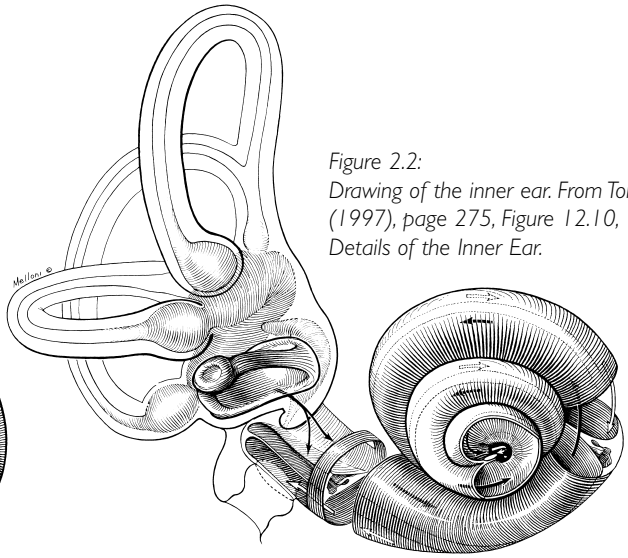
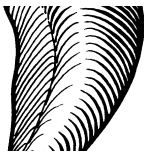
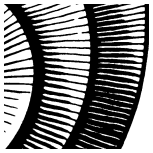


Figure 2.2:
Drawing of the inner ear. From Tortora (1997), page 275, Figure 12.10, Details of the Inner Ear.

2. Illustration Techniques

In order to emulate the process of creating illustrations with the computer it is essential to first study real hand-made illustrations. In particular, technical and stylistic features should be analysed in this step. In the following we will look at three different visualizations of the human inner ear (see Figures 2.1, 2.2, and 2.3) as well as a schematic illustration of the cerebellum (see Figure 2.4) to show the differences of depiction of one and the same object.





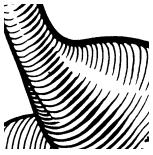


Figure 2.1 is a very detailed pen-and-ink drawing. The silhouette is emphasized in order to visualize the main form of the object. However, with growing distance from the viewer the lines become thinner and even start to show line haloes to clearly distinguish between separate structures. For better visualizing plasticity and form the artist used a uniform hatching for most of the surface. As described in Hodges (1998), the lines are drawn along ellipses that follow the object's structure. However, the lines do not continue across sharp bends of the surface but end there. This way the discontinuity of the surface is emphasized. Shadows are depicted by varying the line thickness. In very bright areas lines are completely omitted which is visible, in particular, at highlights.

The drawing shown in Figure 2.2 was created by the same author as the one in Figure 2.1. Therefore, there are quite some similarities between the two images. However, in the second drawing more attention was paid to formerly hidden structures. These are now shown by employing transparency and cut-away views (for techniques to create those types of images automatically see, for example, Diepstraten et al. (2003)). In addition, internal processes were visualized using arrows. For example, black and white arrows run along the sections of the cochlea. Other parts that are not essential for the communicative goals were omitted, for example, the onsets of the nerves.

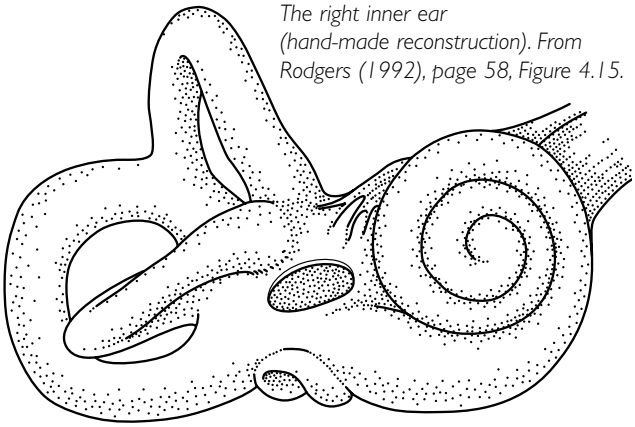


Figure 2.3:
The right inner ear (hand-made reconstruction). From Rodgers (1992), page 58, Figure 4.15.

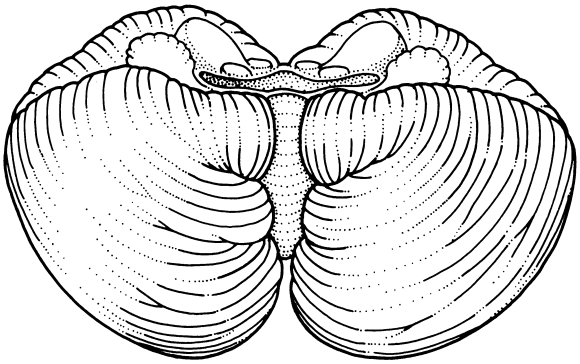
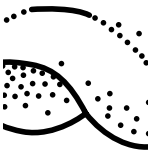
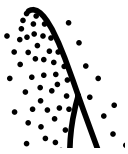


Figure 2.4:
The cerebellum. From Rodgers (1992), page 324, Figure 21.9b.

The artwork shown in Figure 2.3 differs significantly from the previous two in that considerably less emphasis was placed on the depiction of details. Instead, the artist used special care to visualize the overall form of the object. Lines are only used to represent the silhouette and have a constant line width. In some positions they are replaced by sets of dots, in particular, in places where silhouette lines vanish or to indicate bending surfaces. A second important difference is the type of shading that was employed. In this example, shading is achieved only by using dots.





The drawing in Figure 2.4 shows the schematic view of a human cerebellum. Similarly to the illustration in Figure 2.3 shading is not achieved by varying the line width. Instead, lines stop at some point and are replaced by a series of variably spaced dots in order to vary the perceived darkness of the lines. This occurs, in particular, at lines that represent strong bends of the surface which leads to a characteristic texturing. On the other hand, this technique is also used to simulate shading of the surface in general. Specifically, one can observe that the dots of the stippling only follow almost parallel lines and are not distributed in all directions.

3. Related Work

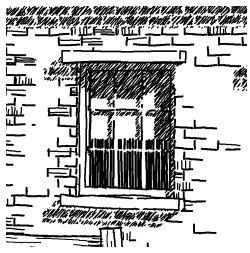
The domain of non-photorealistic rendering has been developing over the period of approximately the last 20 years. Therefore, there is a number of publications that are similar in their goals and methods to this report. They will be reviewed in the following.

Salisbury et al. (1994)

Salisbury et al. (1994) describe the creation of stroke textures that store individual strokes in order to achieve a specific structure. In addition, it is possible to assign priorities to strokes so that they can be grouped such that each group represents a given tone. This way strokes with higher priority are favored over those with lower priority. For rendering scenes the user selects a specific stroke texture and then starts drawing on the canvas. During this drawing process the system selects strokes from the stroke texture until the user-specified gray value has been reached.

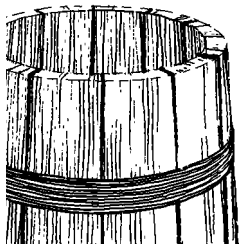
Winkenbach and Salesin (1994)

The technique from Salisbury et al. (1994) can also be used to visualize 3D objects with stroke textures. In addition to the tone value, in this approach it is also possible to control the level of detail shown for certain parts of the scene by user-placed detail segments. This allows to easily simulate the omission of detail in some regions of illustrations as used by real artists as well. The major advantage of using stroke textures in 3D rendering is that the density of strokes automatically adapts to the desired output resolution while maintaining the desired perceived local tone.



Winkenbach and Salesin (1996)

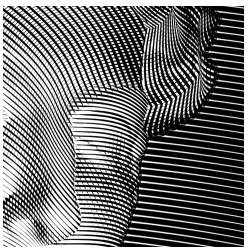
Instead of using polygonal models, the rendering system from Winkenbach and Salesin (1994) was extended to work with parametric surfaces as well. The surface parameterization may now be used to specify the orientation of strokes. In order to achieve an even distribution of strokes, the authors describe a metric that allows to measure the distance of the projected strokes in image-space. This is used, in turn, to adapt the line thickness and, thus, to control the perceived brightness of the texture.



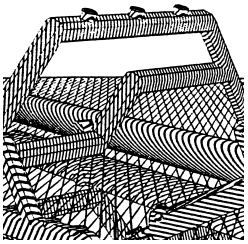
Pnueli and Bruckstein (1994)

In their system **Digi*Dürer***, a grayscale image is segmented into level curves using potential fields so that the curves are equivalent to a rasterization of the image and also follow the image's content. At the same time, it is also possible to convert the curves into dot sequences to simulate techniques used by artists to create half-toned images.

Ostromoukhov (1999)



Regions from a grayscale image are manually segmented and covered with Bézier patches with their parametrization directions following the image's main features. Specific dither screens are assigned to each patch and warped according to the form of the patch. Finally, all dither screens are combined using several combination rules to form a dither screen for the whole image. Applied to the image this allows to generate a black-and-white version of it that simulates the technique of wood-cuts.

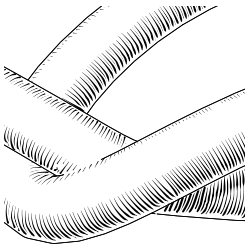


Leister (1994)

This approach is based on modifying a raytracer. In particular, the color computation stage was modified to generate hatched images. Instead of simulating a physical illumination processes, a procedural texture is evaluated to generate hatching patterns. These are discretized in a final step according to illumination data and composed with object silhouettes.

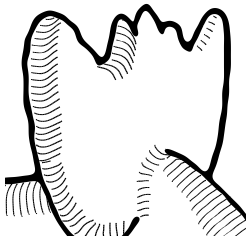
Deussen et al. (1999)

To create hatching lines, the authors first compute an internal skeleton of a given triangular mesh. Following the path of this skeleton the object's surface is cut with planes perpendicular to the skeleton. The line width of the resulting visible intersection curves are post-processed according to shading information and then rendered. However, this approach can only be applied to non-branching structures which severely limits the possible application areas.



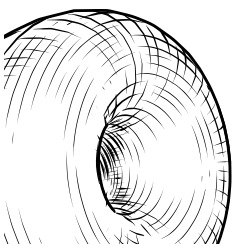
Rössl and Kobbelt (2000)

All nodes of the model's mesh are processed using a discrete curvature analysis to approximate the principal curvatures. They are then linearly interpolated over the surface of the model and projected according to the virtual camera to generate a G-buffer. Now, parts of the image can be covered with curves following the principal curvature directions. In order to suppress turbulences and noise, only a few very significant curves are chosen. Curves in between these are determined using interpolation to generate the final rendition.



Hertzmann and Zorin (2000)

This technique also uses a discrete curvature analysis. In contrast to the previous method, however, the authors determine a vector field of principal curvature directions to be used for cross-hatching. To guarantee homogeneous directions as much as possible, the vector field is first enhanced using a global optimization strategy. After a projection into image-space, streamlines are traced along both projected curvature directions to generate cross-hatching.



4. Direction Vector Fields

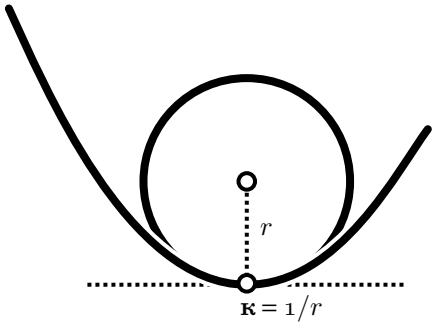


Figure 4.1: Definition of curvature κ.

Since lines on the model's surface should not follow arbitrary directions, we first need a technique to specify line directions for the entire surface. We use a tensor field for this purpose which we will call direction field in the following. It will be used to store a unit vector in each vertex' tangential plane which specifies the preferred hatching direction in this point.

There are many possibilities for specifying the directions in such a field. For example, the direction of an internal skeleton can be used as a basis (as done by Deussen et al, 1999). However, we want to focus on curvature analysis in our approach. Local curvature is well suited for our purpose because it can be derived directly from the object's geometry and since artists also seem to follow principal curvature direction in hatched images. In addition, there are a number of previous techniques that successfully base hatching line generation on local curvature (see, for example, Interrante (1997), Girshick et al. (2000), and Hertzmann and Zorin (2000)). These works are based on psychological studies that suggest that curves following the principal directions are well suited to visualize the three-dimensional structure of objects to a viewer even if the lines are projected onto a plane.

4.1 Curvature Analysis

The curvature of a point on a curve is defined by the circle that meets the curve tangentially in that point and that has locally the same first and second derivative (see Figure 4.1). The reciprocal value of the radius of the circle equals the curvature κ of the point on the curve. The sign of the curvature denotes whether the curve turns to the right (negative curvature) or to the left (positive curvature).

To extend this concept to surfaces, we have to look at a set of curves through one point on the surface. These curves result from intersections of the surface with all planes that contain both the point on the surface and its normal vector. Essentially, this is equivalent to a plane containing the point's normal vector which is rotated around it. Curvature values can now be determined for each of the resulting intersection curves as described above. However, two of the intersection curves deserve further attention: those with the maximum and minimum curvature value – the so-called principal curvatures ($\kappa 1$ and $\kappa 2$, respectively). The directions characterized by the corresponding intersection curves are called principal directions and lie orthogonal to each other. They also both lie in the point's tangential plane and, thus, form an orthonormal basis together with the point's normal vector. The product of $\kappa 1$ and $\kappa 2$ is called Gaussian curvature and

the mean value of $\kappa 1$ and $\kappa 2$ is called average curvature.

However, when trying to apply this concept to polygons it quickly becomes apparent that already the piecewise linear nature of polygonal strokes (in 2D) leads to problems when trying to determine the radius of the tangential circles. The radius would be zero at nodes and infinitely long everywhere else. Therefore, in practice polygonal strokes are approximated by smooth curves which, in turn, are analyzed to determine the curvature. For 3D surfaces a similar approach is taken. For example, a quadric is defined through a node and its neighbors. In our approach we apply a similar method which that is based on approximating the mesh locally by fitting a second degree Taylor polynomial to a node and its direct neighbors (see Rössl et al., 1999). This technique has proven to be very robust in practice.

Once having approximated the principal curvatures of a vertex it is also necessary to use this information in order to determine which direction a hatching line should follow at this point. In hand-made illustrations the hatching lines often run cylindrically around elongated structures (see, for example, Figures 2.1 and 2.2). This has the advantage that, in particular, the distortion of the circles caused by their projection indicates the orientation of the object's basic structure (see Hodges, 1989, page 99). Therefore, to achieve a similar effect, a simple heuristic suffices to determine the hatching direction.

The first or second principal direction is selected depending on the local form of the surface. Because we are interested in the direction of maximum absolute curvature, it suffices to evaluate the sign of the average curvature. Depending on the individual absolute values of the principal curvatures, the one with the maximum absolute value determines the sign of the average curvature. In case it is positive we choose the first principal direction (i.e., that of $\kappa 1$) and otherwise the second principal direction (i.e., that of $\kappa 2$) as illustrated in Figure 4.2.

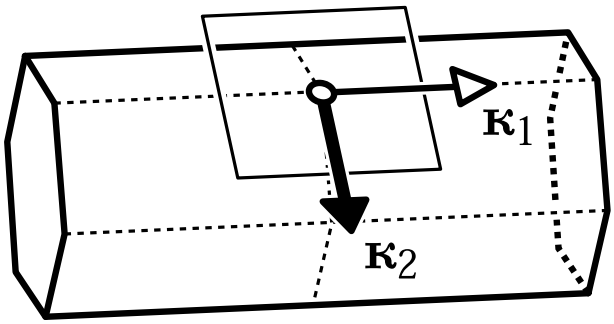


Figure 4.2: Selection of hatching direction for a vertex on the surface of a mesh based on the maximum absolute values of $\kappa 1$ *and* $\kappa 2$.

4.2 Optimization

Unfortunately, there are some fundamental problems with determining the hatching direction based on discrete curvature analysis. On the one hand, the curvature is not defined for every vertex of a mesh. For example, on planar surface or on a sphere it is impossible to determine the first and second principal directions. On the other hand, the method is very sensitive to high frequencies of the surface geometry, e.g., due to noise or fine details. These result in inhomogeneous direction fields. Trying to derive hatched illustrations from such inhomogeneous fields would yield very unnatural and rather disturbing images that do not support the recognition of form.

In order to overcome problems with high frequencies filtering can be applied. For example, the model can be smoothed prior to determining the direction field or details of the mesh can be removed using a mesh simplification algorithm (see, for example, Praun et al., 2001). This, however, only removes parts of the problems. There will remain points where the principal directions are not well defined. The problem may even get worse because mesh smoothing operations tend to increase the percentage of planar and round parts of the model. Then, even less nodes are assigned a well defined first and second principal direction.

Therefore, we put more emphasis on methods that first compute the direction field and, afterwards, try to homogenize the field as well as complete it by adding directions where they were previously not well defined. In order to do this, first a criterion is necessary that can be used to evaluate the field and determine those inhomogeneous regions. This is achieved by using an energy term that indicates how severe the differences are between the individual direction vectors. The higher its value the higher is the inhomogeneity. To compute such an energy term we examine two different versions and discuss them in the following.

4.3 Tensions Between Neighbors

The first approach determines the tension between adjacent nodes. For every pair of two nodes the respective situation is translated into a planar problem. For that purpose we project the connection vector between both nodes into each respective tangential plane of both involved nodes. The projection is used to align the local coordinate systems of both tangential planes and to form a uniform shared coordinate system. Now we are able to also find the transformed direction vectors of the two vertices (which were already located in the respective tangential plane) in this joint coordinate system and we can derive the difference between the two vectors. Attention has to be paid to the fact that the tension should be biggest for orthogonal directions while it should be minimal when the two direction vectors are linearly dependent. In order to model this we take the angle between both vectors and apply the following function to it:

$$\text{tension} = 1 - \cos(2\theta) \qquad \textit{Equation 4.1}$$

In contrast to directly working with angles this has the advantage that we may obtain a continuous derivative which will later be of importance.

Performing the step described above for every edge in the model and summing up the results for the individual nodes yields the complete tension for each node.

4.4 Tensions in Local Neighborhoods

This second approach is equivalent to the previous one in most aspects except that now we do not only consider adjacent edges but all edges in a well defined local environment of the node. The size of this environment is given by a sphere with a user-defined radius. To find all edges in the environment we first collect all faces into a set that are adjacent to the starting node. Then, as long as there are faces adjacent to the set that are still located at least partially in the sphere we add them to the set. Finally, all edges that bound the faces in the set are examined and their tension is determined. The tension of each edge is weighted according to how much of the edge lies inside the sphere (see Figure 4.3). The final energy term for the local environment is now obtained by summing up the weighted individual tensions.

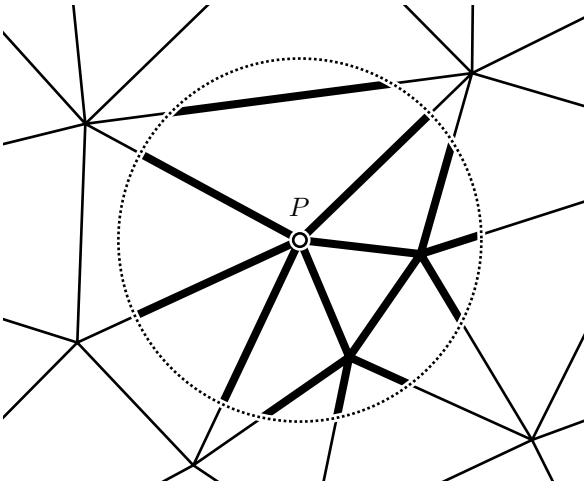


Figure 4.3:
All edges in the local environment of point P. In order to weight the contribution of each edge the percentage of the edge inside the environment is considered.

4.5 Minimization

After having defined a criterion for homogeneity, a user-defined threshold is used to determine all nodes that do not meet this criterion. Their degree of homogeneity has to be improved (see an example of a non-optimized direction field in Figure 4.4). We accomplish this by minimizing the nodes' energy term. By doing so we reduce the inhomogeneity of the entire field which was the initial goal. The most simple approach is to consider each node individually and alter its direction vector depending on the direction vectors in the local neighborhood so that the energy term is minimized. A simple method for this is, for example, the local relaxation of the direction field. For every node we determine by how much the direction vector deviates from the vectors in the neighborhood. The vector in question is then rotated back by the average difference so that the tension to its neighbors is reduced. By iterating this process several times the direction field can be improved to a certain degree. However, there are situations where this technique has limitations. Then, remaining tensions cannot be further removed since there are zones that remain in a stable but not energy-minimized situation. These problems can only be solved by rotating several direction vectors at once which is not possible due to the local character of the algorithm (see Figure 4.5).

In order to avoid such problems a global optimization technique (see Hertzmann and Zorin, 2000) is preferrable. A well-known and freely usable algorithm for global optimization of systems with a very high number of variables is the L-BFGS-B technique (see Zhu and Byrd, 1997). An implementation thereof is

available as a complete package. However, since this is implemented in FORTRAN it had to be ported to C++ first to be usable for our application. We employed [F2C] to do this with relatively little effort. L-BFGS-B is a gradient descent technique, and therefore we need access not only to the global energy term and to the variables to be optimized but also to their derivatives. The technique tries to optimize the angles by which the direction vectors have to be rotated so that these offset angles were used as the input variables. In order to obtain both the global energy term and the gradient vector of the variables we again use the homogeneity criterion. Due to computational complexity considerations we only use the simple version, i.e., the criterion of tensions between neighbors. Using this and the angle offsets we are now able to determine the degree of tension at the individual vertices. The sum of all tensions yields the global energy term. The respective components of the gradient vector can be determined using the derivative of the previously given cosine term (see Equation 4.1). The optimization now tries to find a constellation of angle offsets in which the global energy term is minimal. Rotating all direction vectors by exactly this computed amount will results in a very homogeneous direction field (see Figure 4.6).

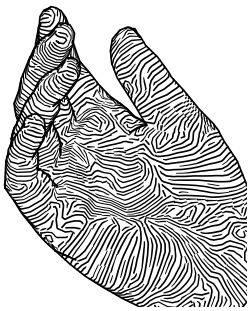


Figure 4.4: Direction field without any optimization.

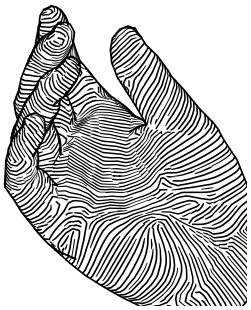


Figure 4.5: Direction field with only local optimization.

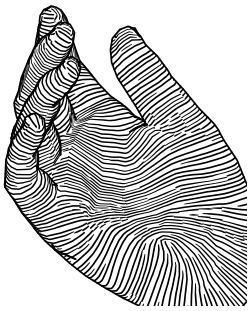


Figure 4.6: Direction field with global optimization.

5. Streamlines

After we have obtained a direction field as homogeneous as possible, we now have to create lines based upon the given direction information that follow the direction vectors on the surface of the model. Since this process is rather time-consuming and we want to allow the interactive exploration of models we do not place the streamlines in image-space but let them live in object-space. Although this means that some artifacts arise during projection that have to be dealt with separately this has the advantage that the line representation is independent from the viewing direction and has to be recomputed only if the model itself changes.

During the streamline computation it has to be ensured that the entire surface is covered so that no empty spots occur. In addition, all lines should have approximately equal distances to each other. Since it is not feasible to require exactly the same distance everywhere we only define minimal and maximal distances.

As foundation for our line placement technique we use an algorithm previously suggested for the visualization of 2D vector fields (see Jobard and Lefer, 1997) and adapt it to 3D. The algorithm starts by integrating streamlines originating from an initial starting vertex. During this process, new potential starting vertices are determined and stored in a queue for later use. While being constructed, a streamline grows simultaneous at both of its ends and continues to grow until a termination criterion prevents further growth. To ascertain that the entire surface is covered with streamlines we use the centers of gravity of one of the the mesh's triangles as a new starting vertex if the queue of starting vertices should be empty at one point in time. The algorithm terminates as soon as the queue of possible starting vertices is empty and the centers of gravity of all triangles have been tested. In order to find new starting vertices while a line is integrated we repeatedly start two queries orthogonal to the line. These try to move away from the line far enough so that they reach the maximal allowed distance from the line. If there is no other line within the minimal line distance from this new point it is considered to be a potential starting vertex and we add it to the starting vertex queue.

5.1 Integration

For the integration itself we use a fourth order Runge-Kutta ODE solver (see Press, 1992). We also experimented with a simple Euler integrator. However, this did not produce satisfying results. By using a better integrator we are able to keep the step size larger without having to compromise the stability of the integration process. This not only results in a significantly reduced processing time for the integration but also reduces the number of resulting line segments. Ultimately, this also increases the frame-rate when rendering the resulting line drawing and allows interactive processing.

Since the direction vectors are only defined for the vertices of the polygonal model it is necessary to interpolate them for in-between points. For this purpose we first determine the barycentric coordinates for the point based

on its position inside the surrounding triangle. These, in turn, are used to compute a direction vector for the point using spherical linear interpolation of the direction vectors of the vertices of the triangle. Since it is important that all direction vectors point in approximately the same direction potentially we have to turn the direction vectors of the triangle's vertices by 180 degrees before doing the interpolation. As a reference for this step we use the last determined orientation. By computing its dot product with the direction vectors of the new triangle we are able to determine whether the new directions differ by more than 90 degrees or not. If this is the case we negate them. In case there are no previous orientations (which is only the case for starting vertices of streamlines) we use the direction vector of the closest triangle vertex.

In addition to determining the direction vector for a given point it is also necessary for the Runge-Kutta integration to obtain a modified direction vector that will later be used for the actual integration step. It is sampled at different points that lie in the direction extracted from the direction field and have certain distances. We follow the direction vector from the previous point and run along the surface until we have reached the given distance. In case we hit a triangle edge during this process we adapt the direction vector by rotating it around the triangle edge until it lies in the plane defined by the new triangle. When we have reached the desired distance from the original point we determine a sampled direction vector by interpolation as before. Now we have to back-transform this sample into the original local coordinate system. This way we end up with several sampled direction vectors that are then weighted according to the Runge-Kutta scheme. This allows us to determine the effective direction vector that we can now use to follow the streamline.

This completes the necessary steps required for implementing the ODE solver. Based on weighting several direction vectors and a given step size we determine a new direction vector. This is used to run along the surface of the model and to find a new vertex for the streamline. In case triangle edges are crossed new vertices are added there as well in order to make sure that the streamline does not leave the surface of the model.

5.2 Termination Criteria

As mentioned before a streamline is continued until a termination criterion has been met. In our implementation we provide several of these for the user to turn on and off in order to achieve different effects. However, in some rare cases it may happen that streamlines have to be stopped because the integration cannot be continued. These cases occur because the program uses floating point arithmetic which may result in floating point underflows in certain situations. Such cases are also considered to be termination criteria.

During the integration of a streamline it is recorded how much a line is bent, i.e., how much it deviates from a straight path due to the direction field. The user is able to specify a threshold for this value which is then used to determine when a streamline should no longer be followed. This is necessary to avoid too sharp bends of streamlines. In particular,

this comes in handy when working with less homogeneous direction fields or in the neighborhood of singularities.

To visualize sharp bends of the surface we only pass the border of triangles if the angle between their normals is less than a certain user-defined threshold. This results in a visual discontinuity of the flow of streamlines along sharp edges of the model. This simulates a technique which is often used by artists in hand-made illustrations (see, for example, Figures 2.1 and 2.2).

Very long streamlines often tend to emerge as separate structures which is a not desired effect. In order to prevent this it is possible to specify a maximum length for a streamline. If this is exceeded during the integration only that part of the specific step is finished that is required to exactly reach the maximum length.

The most important termination criterion, however, is the distance to other streamlines in the neighborhood. Since it is not desired that streamlines cross each other we terminate the integration of a new streamline as soon as its distance to other streamlines is less than the minimum line distance.

In addition, we test each finished streamline whether it has reached a minimum length. If this is not the case we discard the entire streamline. This prevents the creation of very short line fragments and results in a more quiet appearance.

5.3 Distance Between Streamlines

As already mentioned it is important to determine the distance of one streamline to other streamlines on the surface in order to avoid line collisions and to discard starting vertices for new streamlines. There are several ways to accomplish this which we will discuss in the following.

Our first approach was to use 3D grid hashing. For this purpose each segment is represented by a series of points that are stored in a 3D grid. The storage space for such a 3D matrix may be very high although most of the cells are empty. Hence, it is advantageous to use a so-called sparse matrix. This sparse matrix was implemented by deriving a hash value from the 3D grid coordinates which, in turn, was used to access an array of the respective grid cells. The size of each cell is bounded by the minimum line distance. In order to test the environment of a given point it is first determined to which 3D grid cell it belongs. Then, all points in this cell and its eight neighboring cells are determined and the minimum distance to them is computed. This approach is fairly fast and the error introduced by the necessary discretization can be neglected in practice. However, the region of influence of a line is not bounded by the object's surface. This can be noticed, in particular, on very thin structures where the geometry is thinner than the minimum line distance. This occurs, for example, for models of flowers where the flower petals are usually very thin. In such cases the generation of lines on one side of the petal is prevented by the lines on the other side.

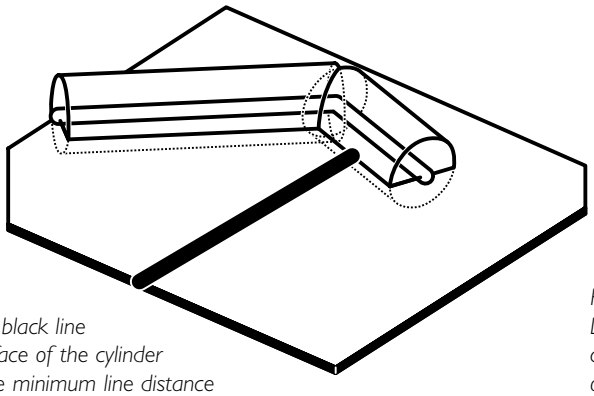


Figure 5.1:
The integration of the black line terminates at the surface of the cylinder which ensures that the minimum line distance is not violated.

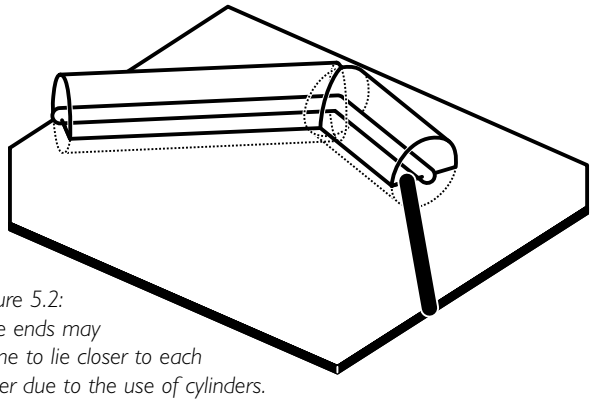


Figure 5.2:
Line ends may come to lie closer to each other due to the use of cylinders.

Instead, we now represent streamlines by a sequence of segments that are, in turn, represented by cylinders having the radius of the minimum line distance (see Figure 5.1). They are stored together with the triangles on which their respective line segments are located. In addition, they are also stored in the triangles that are at least partially inside the cylinders. In order to accomplish this we start from the original triangle and test its edges if they intersect or are contained in the cylinder. We continue to recursively test all neighbors provided that the angle between each two neighboring faces' normals is sufficiently small. This results in cylinders that only have influence on their local environments. Provided that the triangles do not get to large the access of the contained cylinders is sufficiently fast. In order to test the environment of one point it is sufficient to check all cylinders assigned to the respective triangle. As long as the size of the triangles is smaller than the minimum line distance one cylinder at most may be contained in each triangle and has to be tested.

An additional advantage of this approach is that lines can come much closer to each other than with techniques that test the actual distance between a point and a segment (see Figure 5.2). Otherwise there would be a gap of at least the minimum line distance between two not connected streamlines (see Figure 5.3). One aspect to notice, however, is that at the connection between two line segments gaps occur between the respective cylinders. These gaps should not be filled with additional lines. In order to prevent this we add additional cylinders that will close this gap but that do not represent a specific line segment of a streamline (see Figure 5.4).

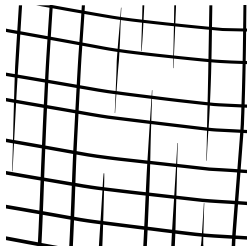


Figure 5.3:
Detail from a flow field generated by Hertzmann and Zorin (2000) showing obvious gaps between the line ends.

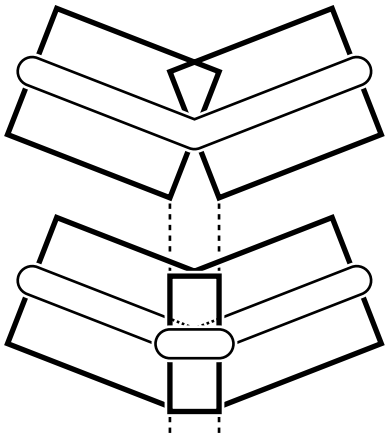


Figure 5.4:
The cylinders of a two-segment streamline exhibit gaps at bends of the streamline. These have to be covered by an additional cylinder to prevent lines entering this gap.

5.4 Implementation Details

For the previously described algorithms it is necessary to work intensively on the surface of the models. Therefore, information about the local neighborhood of triangles is very important. Since it is sufficient for our purposes to work with two-manifolds we decided to work with the half edge data structure (see Mäntylä, 1988). This data structure represents each edge of the mesh by two directed edges that each belong to one of the two adjacent faces and that carry information about their neighborhood (see Figure 5.5). With this fairly simple data structure it is possible to answer the most important queries about neighboring triangles in very little time:

```
struct HalfEdge {
   Vertex*   vert;
   HalfEdge* next;
   HalfEdge* loop;
   HalfEdge* pair;
   Triangle* tri;
};
```

A frequently used operation is to flip a vector around an edge in order to transform it from one triangle to the next. A naïve implementation would simply rotate the vector by the angle between the two adjacent faces' normals. However, this is quite complex to compute and there is a much simpler geometric way to achieve the same effect. By projecting the problem into a plane the two planes given by the two triangles become straight lines with the given vector being part of the one line and the wanted vector being part of the other line. Because the rotation does not alter the length of the vector the original and the wanted vector together span an isosceles triangle. Splitting this triangle at the bisector results in two right triangles whose hypotenuses represent the given and the wanted vectors. The shared part along the bisector is equivalent to the projection of one of the vectors onto the bisector. Therefore, based on the given vector and this projection it is easily possible to derive the wanted vector. This can now directly be applied to the original case (see Figure 5.6).

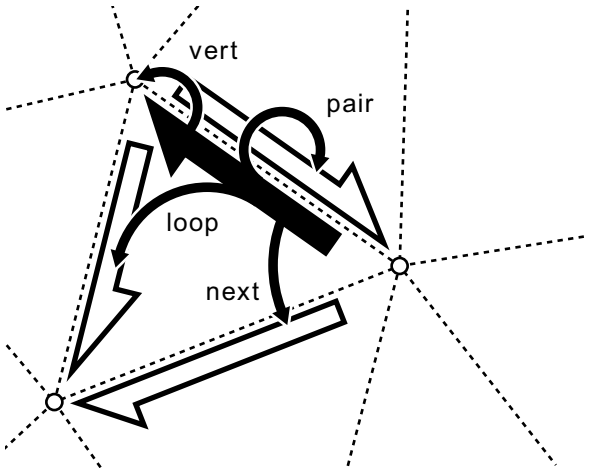


Figure 5.5:
Visualization of a half edge along with its direct neighbors.

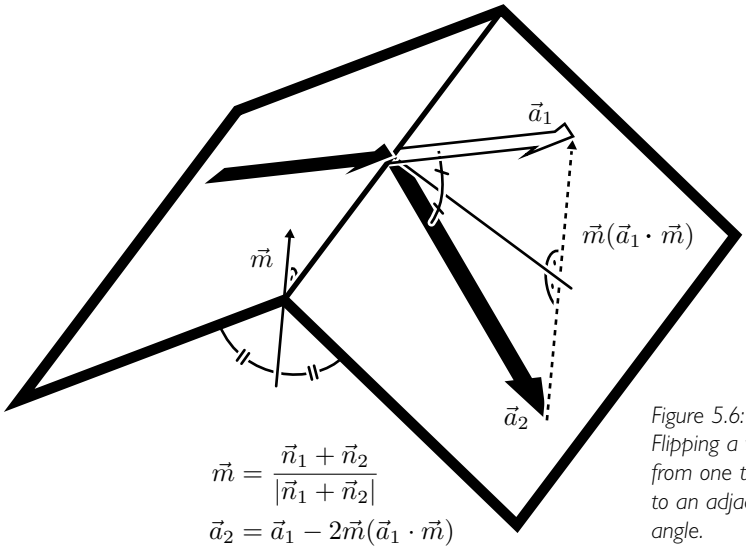


Figure 5.6:
Flipping a vector from one triangle to an adjacent triangle.

6. Visualization

After the pre-processing has been completed the generated data will now be used to generate the final hatched rendition. The steps necessary for this task will be discussed in the following.

6.1 Hidden Line Removal

We will only render lines into our final image and do not use occluding surfaces as in traditional rendering. Therefore, we will have to remove those parts of the surface that are occluded by parts of the model that lie in front of them. In the system our implementation is based upon (OpenNPAR, see Halper et al., 2003) there is already a module for removing hidden parts of lines (see Isenberg et al., 2002). This module, however, was originally intended for the hidden line removal of silhouette lines. This algorithm removes occluded parts of a stroke based on the completed z-buffer rendering of the 3D model. The stroke segments are rasterized and compared with the z-buffer data. In case there is a line pixel or a pixel in its 8-neightborhood that is closer to the viewer than the pixels recorded in the z-buffer then it is visible. The invisible parts of the stroke are detected using this method and eliminated from it. Since the strokes generated by our method do also lie on the surface of the model this technique works as well. The only needed modification was to treat some additional data similar to the strokes' geometry so that this was adapted accordingly.

In addition, some of this supplemental data was used to perform backface culling prior to the hidden line removal so that only about half of the original stroke data had to be processed by the hidden line module. The resulting gain of rendering speed was very noticeable. An additional advantage of the backface culling step is that streamlines that intersect with silhouette strokes were previously not clipped correctly due to the only pixel accuracy of the z-buffer technique. This caused a small part of the back-facing portion of the streamline to be rendered anyway (see Figure 6.1). This problem was also solved by first applying backface culling and performing the hidden line removal afterwards.

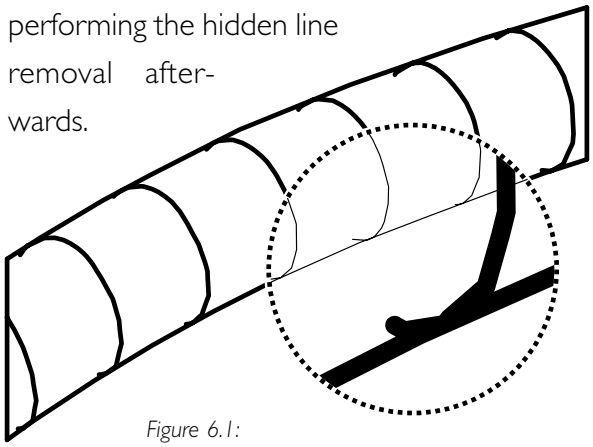


Figure 6.1:
Due to tolerances in the z-buffer based hidden line removal technique the streamlines do not get clipped exactly at the change of visibility. This can be corrected by first applying a backface culling step.

6.2 Line Shading

Before we can apply shading to the streamlines we have to make sure that the perceived gray value of the unprocessed streamlines is homogeneous for the whole model. After this has been done we are able to apply shading to the lines by simply modulating the lines' thickness which is proportional to their perceived gray value. Therefore, the shading model is entirely local and can be applied on a per-vertex basis.

One factor that causes inhomogeneities it the projection of lines from object-space into image-space. Depending on the orientation of the surface and the direction of approximately parallel lines on it the distance between the projected lines is much smaller than it was originally specified in object-space. If now the line thickness is not affected by the projection then only the distances change. Therefore, a changed proportion between space covered by lines and background results depending on the orientation of the surface. This directly affects the perceived gray value as shown in Figure 6.2.

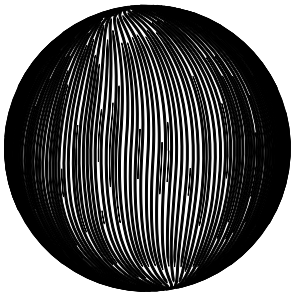


Figure 6.2:
Differences in the perceived gray value due to placing the lines in object-space.

In order to account for this we employ a heuristic to approximate the distances of lines after their projection into image-space. For this purpose we construct a unit vector perpendicular to the average direction of the streamlines s lying in the tangential plane of the considered vertex given by the vertex normal n. The length of its projection onto the viewing plane is proportional to the actually rendered line distances (see Equation 6.1).

$$w' = w \left|\left| \frac{s \times n}{||s \times n||} T \right|\right| \qquad \textit{Equation 6.1}$$

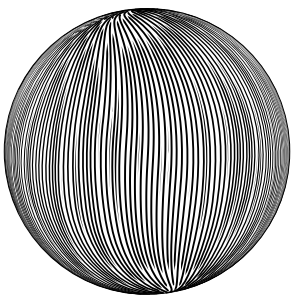


Figure 6.3:
The same set of streamlines as in Figure 6.2. To control the perceived gray value a correction factor has been applied.

Modulating the line thickness using this value increases the homogeneity of the perceived gray value significantly (see Figure 6.3). For further improvement of the quality, however, we would have to use additional information. For example, one possibility would be to directly compute and store two vectors in each vertex that denote the distance to the two neighboring lines.

Inhomogeneities are also caused by lines having not exactly the same distance to each other because the distance may vary between the minimum and maximum distance. To account for this, we apply a preprocessing step that examines all stroke vertices in object-space and computes a correction factor after all lines have been placed. This factor tapers lines that come closer to each other by computing a quotient of the distance to the nearest segment in the neighborhood of the vertex and the difference between maximum and minimum line distance. Similar to the approach of using cylinders for computing the distance criterion when integrating the streamlines we here compute the distance not omni-directionally (as shown in Figure 6.4) but parallel to the direction of the streamline (as can be seen in Figure 6.5). To do so, we determine the average direction of the streamline at a vertex and construct a plane that has this direction vector as normal vector. We now compute the intersections of near-by streamlines with this plane and derive the shortest distance to the vertex. The advantage of this method is that now lines can come much closer to each other without having their line thickness reduced too early. Thus, we avoid the occurrence of visible gaps.

Figure 6.4:
Tapering based on an omni-directional distance metric.

Figure 6.5:
Tapering based on a distance metric that considers distances parallel to the direction of the streamlines only.

The shading model itself consists of two expressions – one for changing the line thickness and the other one to control line density. The latter allows to change the line appearance from a solid line into a stippled style (see Figure 6.6). These two expressions are evaluated for each vertex and are used by the line rendering module described below to control the actual appearance of the lines.

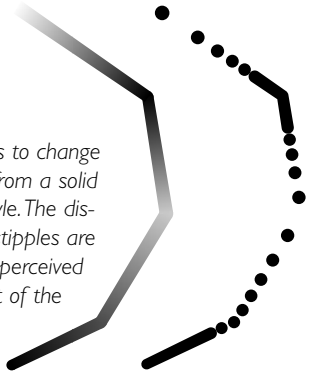


Figure 6.6:
The line density allows to change the line appearance from a solid line into a stippled style. The distances between two stipples are chosen such that the perceived gray value equals that of the desired line density.

6.3 Virtual Machine

To evaluate the shading expressions we implemented a specialized virtual machine. This has the advantage that the expressions do not have to be specified at compilation-time but can be altered at run-time. This makes it much easier to develop a feeling for the result of different expressions and to interactively create new expressions for a set of effects.

The expression itself consists of a defined set of identifiers and operations. These are translated at run-time into a specific byte-code along with changing it from infix notation into postfix notation (i.e., inverse polish notation). Thus, all brackets are removed and the precedence of operators becomes irrelevant (see Figure 6.7).

Figure 6.7:
An example expression on its way from infix notation to byte code.

infix notation: "1+3*(light-rim)"
postfix notation: 1 3 LIGHT RIM - * +
byte code: 02 02 05 06 10 11 09

When such a byte-code expression has to be evaluated for a vertex, first the values for each of the identifiers in the expression are determined. For example, if the identifier "LIGHT" is used in the expression the virtual machine determines the diffuse component of the Phong illumination model at the vertex. Then, the byte-code is processed by a stack-based interpreter that exchanges all identifiers by their previously computed values.

In addition to the usual operations +, -, *, /, and ^ we offer the following identifiers for the user to work with (inspired by the non-photo-realistic shading model by Hamel (2000)):

`LIGHT`
The value of the diffuse component of the Phong illumination model. The necessary normal at the streamline vertex is computed by interpolation as used in Phong shading. The light is set to come from a constant light source left from and above the model as usually used by illustrators (Hodges, 1989, page 71).

`RIM`
Similar to the LIGHT identifier only that the light source is placed between viewer and object in order to simulate rim shadows.

`DEPTH`
The distance between viewing plane and depicted object in order to be able to implement depth cueing.

`CURVATURE`
The average curvature for using it in curvature lighting.

7. Vector Output of Lines

Lines play a central role in this work so we will now focus on their visualization. Related approaches can be found, for example, in Vehmeier (2002).

7.1 OpenGL

When using OpenGL for line rendering the obvious choice is to work with the `glLineWidth` parameter and render lines as usual using the line primitive. The disadvantage of this approach, however, is that OpenGL rather aims for fast rendering than for quality in line depiction. This can easily be observed in the fact that lines are modeled as sheared rectangles. This means that depending on the angle of the line its thickness varies. Only for horizontal and vertical lines the correct line thickness is used. The more the line is rotated to a diagonal orientation the bigger gets the introduced error. Diagonal lines ultimately only have approximately 70% of the specified line width (see Figure 7.1).

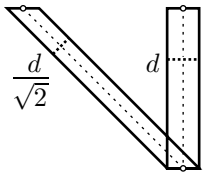


Figure 7.1: OpenGL lines vary their effective line width depending on their orientation.

A related problem occurs at the joints of two adjacent line segments. If the two connected line segments lie in different orientation quadrants the joint will be visible as a small quadratic gap in the stroke which creates a very unpleasant appearance (see Figure 7.2).

Figure 7.2: OpenGL artifacts at the joints between two connected line segments.

The same effect can be observed at line ends. If the last segment in a stroke is oriented approximately diagonally its end is rather pointed and it is dull if the segment is oriented horizontally or vertically.

These artifacts are visible, in particular, in animations if the gap in the joint can change from one frame to the next. Also, the change of line width due to a transformation applied to the line becomes very disturbing.

7.2 Quadrangles

As an alternative to the simple line drawing technique it is also possible to represent stroke segments as connected sets of textured quadrangles along the line's path. This has the advantage that we are able to profit from hardware acceleration when displaying these textured polygons. This enables us to apply many different effects to the line drawings.

Figure 7.3: Although the diameter of the line at the joint is the same as at the ends the line does not have the same width there.

A simple version computes for each joint between two adjacent line segments the respective bisector and places two vertices on it having the distance corresponding to the desired line thickness. These vertices are then rendered as part of a `GL_TRIANGLE_STRIP`. Unfortunately, problems similar to those occurring with OpenGL lines arise. Sharp bends are characterized by a reduced line thickness (see Figure 7.3). However, this effect can easily be accounted for by looking at the geometry of a correct miter. The bisector of the joint is equiv-

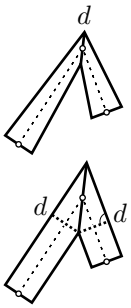


Figure 7.4: Right triangles at a correct miter.

alent to the hypotenuse of a right triangle (see Figure 7.4). The adjacent leg of the triangle now is shorter than the hypotenuse with the factor being the cosine of the angle between the two. Therefore, it is sufficient to divide the original line diameter at each joint vertex by this value in order to achieve a homogeneous line with for a stroke.

7.3 Miter Limit

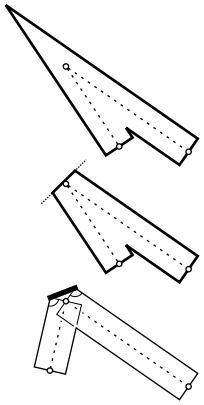


Figure 7.5: When exceeding the miter threshold the tip is cut off.

Unfortunately, even this approach is still not free of problems. The larger the considered angel gets (i.e., the sharper the bent gets) the smaller will be the compute cosine value. Ultimately this would end in an infinitely long tip. Therefore, we have to cut off the tip at a certain threshold (the miter threshold). For this purpose we connect the outer vertices of the two quadrangles representing the line segments. This means that we construct two vectors perpendicular to the two direction vectors at each point. These new vectors point to the outside and have half the length of the desired line width. Finally, the resulting points are connected as shown in Figure 7.5.

7.4 Rounded Lines

A different type of lines is commonly used, in particular, in the classic technical illustration – the rounded line. For this type both the line ends as well as the segment joints are rounded.

This is achieved similarly to the procedure described in the previous section only that we now connect the outer vertices of the two quadrangles with arcs and add a semicircle to each stroke end (see Figure 7.6). This construction is an easy task for lines with constant width. However, it is desirable to be able to use radii at different vertices of a stroke in order to introduce variations to the line width along a line. The construction of the contour line of a segment now is a bit more complicated than in the trivial case because now the contour does not have to be parallel to the segment direction anymore. In order to achieve seamless transitions between contour lines and arcs the lines have to run tangential to the circles. The cosine of the angle to the segment direction equals the ratio between the difference of radii and the distance between the two circle centers (see Figure 7.7).

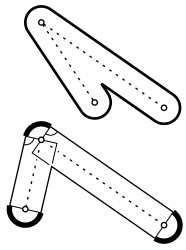
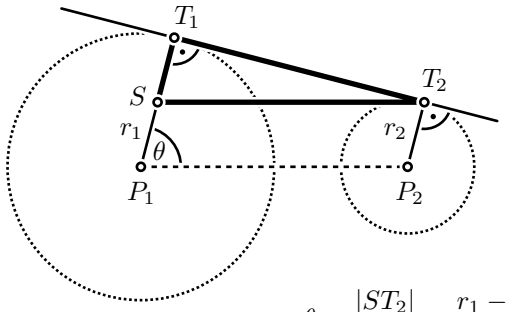


Figure 7.6: Arcs located on bends and semicircles at the stroke ends.

Figure 7.7: Computation of the angle between the line to the connection point of contour line and arc where they meet tangentially and the segment's center line.



$$\cos\theta = \frac{|ST_2|}{|ST_1|} = \frac{r_1 - r_2}{|P_1P_2|}$$

7.5 Implementation Details

First we will discuss the simple case of constant line width. In order to keep the algorithm simple we have to apply some preprocessing and remove segments with zero length. In addition, we collect the data that will later be

used for rendering the lines. This includes, in particular, connection vectors and for each of them an additional perpendicular vector with unit length.

In principle there are tree main cases each of which can be again split into two sub-cases. The main cases are right turns, left turns, and collinear segments. It is easy to distinguish these with an orientation test. For the two cases of left turns and right turns one has to check whether the intersection of the two inner outlines is valid, i.e., if it is inside the two outer outlines. In case of collinear segments it is possible that there is no turn altogether because we deal with a straight segment subdivided into two which can be skipped. However, it is also possible that we encountered a U-turn by 180°.

The rendering routine itself treats the vertices of the stroke sequentially, classifies each turn, and converts it into a triangle strip representation. This results in a triangulation for the lines once all lines have been processed. Since one vertex of a triangle strip may be accessed more than once it is advantageous to use vertex arrays in order to avoid gaps in the rendition. This way the OpenGL driver can discard degenerated triangles during the triangle setup. These are necessary for being able to represent the corners of turns as triangle fans although triangle strips are used for the entire rendering. This situation occurs when the triangles on the inside of a turn are squished together tightly so that all fall onto the same vertex (see Figure 7.8).

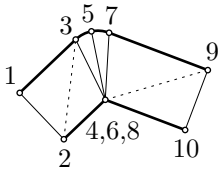


Figure 7.8: Triangle strips with degenerated triangles allow to simulate triangle fans.

However, if there is no single vertex that can represent all inner turn vertices (i.e., in case the inner intersection does not lie on both inner outlines or if we encountered a 180 degrees U-turn) we have to stop and restart the triangulation at this point. Also, in this case we add degenerated triangles in order to avoid the costs of starting and drawing a new vertex array (see Figure 7.9).

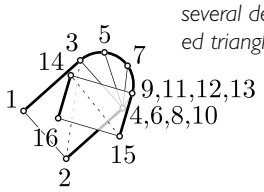


Figure 7.9: Starting over in a triangle strip using several degenerated triangles.

For the more complicated case with variable line thickness the algorithmic procedure is similar to the previously described simple version. However, due to the changed conditions there arise more cases (see Figure 7.10) that cannot easily be distinguished by an orientation test. Therefore we chose to work with the angles of the four tangential seam points. Each individual angle is tested whether it lies inside the range of the respective pair of angles of the opposite segment. This results in four comparison operations that all can be combined in a four bit expression. This is, in turn, used as an index to a lookup table where the respective case can be found. Experiments have proven that this implementation is not only fast but also very robust in practice.

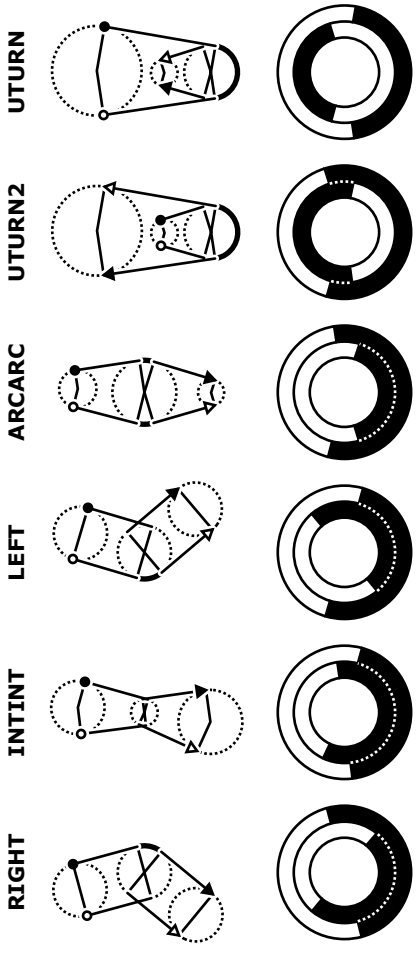


Figure 7.10: The six cases of turn classification and the respective overlapping angle ranges.

7.6 Negative Line Widths

Because we are restricted to parameterize the line widths only at individual vertices, line ends within a stroke can also only occur at vertices. This only happens if the line width is zero for such a vertex. To remove this restriction we introduce the possibility to specify negative line widths. This results in splitting segments for which their two end vertices have different signs of the line width. These segments are split exactly where the two lines tangential to the two radii would intersect (see Figure 7.11). This way we are able to place line ends independently from the specific stroke vertices.

Figure 7.11: Effects produced by successively reducing the line width of one of the vertices.

7.7 Line Stippling

When looking at hand-made line drawings one can observe that in addition to varying the line widths artists tend to use another style element as well – line stippling. This is typically employed when working with pure black ink to simulate varying line density without changing the line width. Either this is done two-dimensionally to depict surface shading and surface details or it is restricted to lines, for example, to end them in a more soft way. Therefore we implemented point stippling into our line renderer. This makes it possible to vary the density of lines by splitting them into a series of single dots (see Figure 6.6).

7.8 Point Distances

To compute the distance between stipple dots we examine the inverse case first – the perceived brightness of two neighboring dots. This can be approximated by the ratio of black area to white background in the quadrangle formed by the two dots. Without loss of generality, this approximation can be performed with two dots of unit size. However, two cases must be examined separately – either the two dots are separated by at least two radii and, therefore, at most touch each other tangentially (see Figure 7.12) or they are closer than this and overlap (see Figure 7.13) in which case we have to avoid to count the overlapped area twice. For both of these cases it is easy to derive respective formulas that take a given distance d and yield the approximated intensity $I(d)$:

$$f(b) = \int_0^b \sqrt{1-x^2}dx = b\sqrt{1-b^2} + \arcsin b$$

$$I(d) = \begin{cases} \text{für } d \geq 2r & \frac{\pi r^2}{2dr} = \frac{\pi r}{2d} \\ \text{sonst} & f(\frac{d}{2r})\frac{2r^2}{2dr} = f(\frac{d}{2r})\frac{r}{d} \end{cases}$$

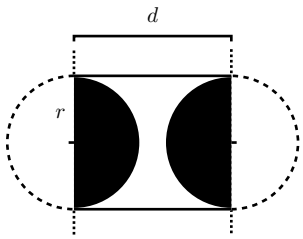


Figure 7.12: Approximation of the perceived brightness between two dots of the same size with distance d >= 2r.

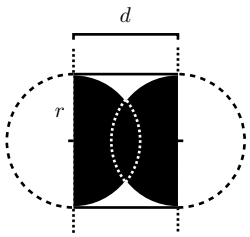


Figure 7.13: Approximation of the perceived brightness between two dots of the same size with distance d < 2r.

Originally, we were interested in the inverse case. Although it is easy to find an inverse function for the first case this is not as easy anymore for the second case. Since we are only interested in speed and approximate exactness it is sufficient, an approximate solution using a lookup table can be employed. We have to pay attention to the fact that the values in the lookup table should have approximately equal distances. For this purpose we use an adaptive grid to sample the function starting from its minimum. The adaptive step size is changed such that at least one function value lies between the horizontal grid lines. The exact intersection point is determined by linear interpolation of one value above and one below the horizontal line (see Figure 7.14).

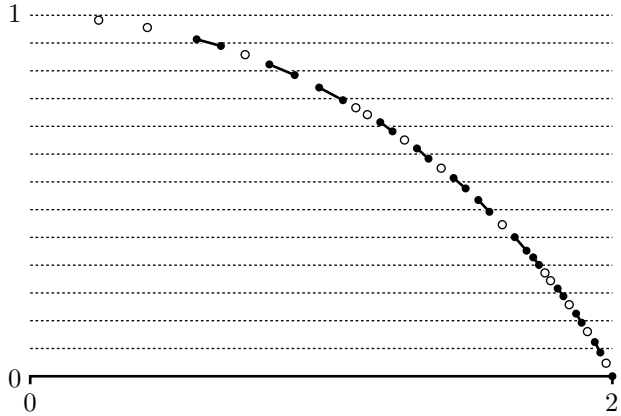


Figure 7.14:
Sampling the function of which we want to compute a lookup table for the inverse function. Segments are computed that intersect the grid lines to derive the lookup values.

In order to use this distance data for placing dots along a stroke we proceed as follows. First, we subdivide a stroke into several parts if it contains vertices that have minimum density or less because this would otherwise result in infinitely long distances. Then we start with a vertex of maximum density and begin with distributing dots on the lines path. For every point on the segment we first linearly interpolate the desired density and use that to estimate the distance to the next dot. However, this is only a first estimate for the distance because the goal is to approximate the distance between two dots and not to approximate the distance at a single point. Therefore, we use this first value to obtain an average density for the section it specifies. This is used in turn to determine the final distance between the current and the next dot. This, again, is only an approximation of the density but it has proven in practice to be a good compromise between exactness, speed, and a balanced dot distribution.

7.9 Minimum Dot Distances

In most cases it is more important to have an esthetically pleasing transition from a regular line to the stippled style than to have a mathematically perfect approximation of the line density for all positions. In hand-made drawings one can observe that the dots often have a certain minimum distance to the regular lines. Therefore, we allow the user to specify an equivalent minimum dot distance (see Figure 7.15). To achieve this we subdivide the line segments exactly at those positions where the density equals that of the desired distance. The part where the distance would have been too small is rendered as a regular line and the other one is stippled.

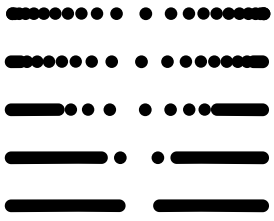


Figure 7.15:
Different minimum distances in the range of [0, 4].

There is a problem at the line ends that arises due to the sequential placement of stipple dots along the line. While the algorithm is able to use the correct distance when starting to place stipples it runs into problems when placing the last stipple. The stippling ends as soon as a stipple dot would be placed closer to a line end than allowed which usually results in an unpleasant gap. Therefore, we place the dots in a two-step process. In the first stage only their relative positions are computed. Once this is finished for an entire stipple section these positions are stretched such that the minimum distance is also maintained for the end of the section (see Figure 7.16).

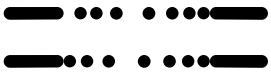


Figure 7.16: Dot placement without and with balancing.

7.10 Line Output

We implemented two different output modules that each have different goals. On the one side we provide a WYSIWYG preview. This allows the user to interactively work with the model and change both the view of the model and the shading formula in order to find optimum values for all of the parameters. We realize this using an optimized line renderer that converts the lines into triangle strips and the points into triangle fans as discussed above. This means that arcs are discretized using a user-defined resolution and the lines are slightly simplified. This allows to adapt the quality according to the users wishes.

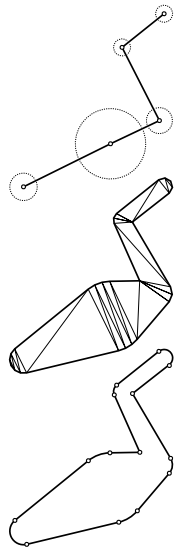


Figure 7.17: A line with indicated variable line thickness and its rendition with triangle strips and vector contours.

For the rendering of monochrome lines on a raster screen with limited resolution it is very important to use measures to fight aliasing problems. Current graphics boards offer full-scene anti-aliasing for this purpose. However, it was not possible to use this technique with our implementation due to a conflict with the used hidden line removal module. When trying to display a model with average complexity that produced about 900 streamlines which in turn yielded about 12,700 single segments this resulted in drastic rendering speed drops. Rendering of the model without hidden line removal allowed 20 fps while rendering with hidden line removal only resulted in not acceptable 2 fps (on a Dual Athlon 1600+ and GeForce2 MX400). Unfortunately, this problem could not be traced down so that the quality of the preview was reduced.

In addition to the WYSIWYG preview we also provide a second output module. This allows to export the graphic as vector data into a PDF file. In this case the arcs are not discretized but remain in their native form (see Figure 1.17). This allows not only to view them in full quality on the screen but can also be further processed which is advantageous, in particular, in the print domain.

Using the PDF format directly allows many possibilities for further processing. For example, several layers of hatching can be produced by turning the entire direction field by a certain angle and adapting the shading model (see Figures 7.18 and 7.19). This can be taken even further by assembling entire collages and colorizing areas and lines (see Figure 7.20). This way the limitation of black-and-white drawings is overcome. An additional advantage is that the limitation to the CMYK color scheme is overcome and spot colors can be used instead. This way colored images do not have to be rasterized in the printing process which significantly benefits the quality of line drawings.

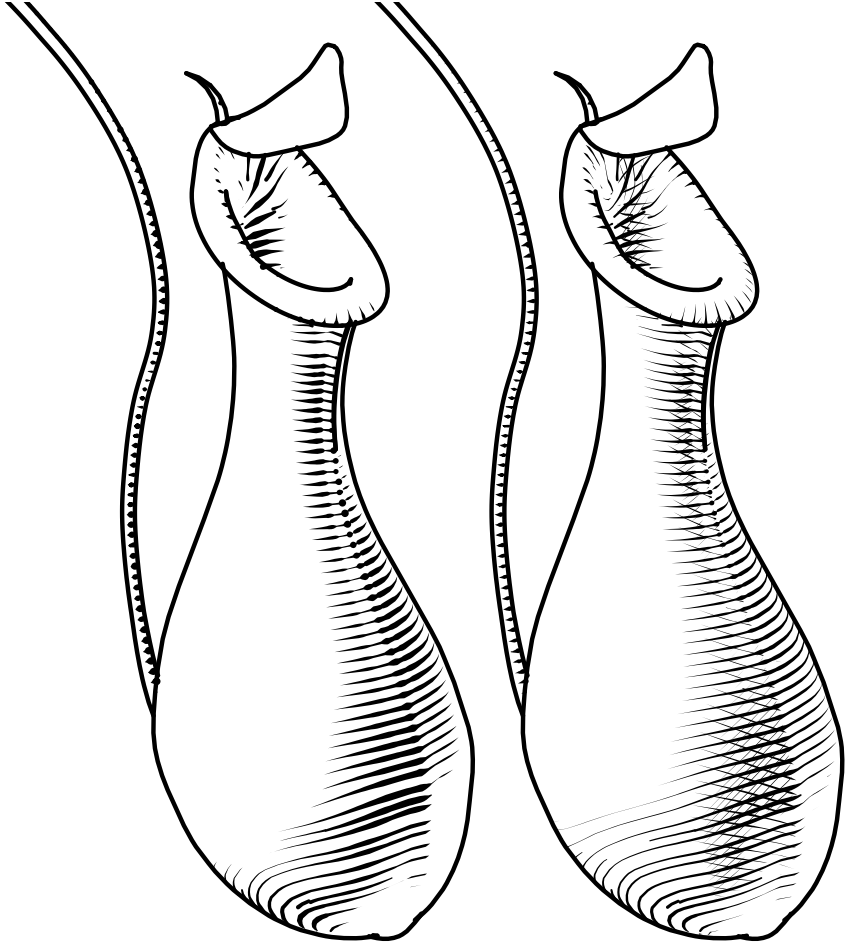


Figure 7.18: Pitcher plant (Nepenthes alata), once with a single layer of hatching and with three layers of hatching where each direction field has been turned slightly.

Figure 7.19a: Hodges (1989), Figure 115, hatching in a hand-made illustration.

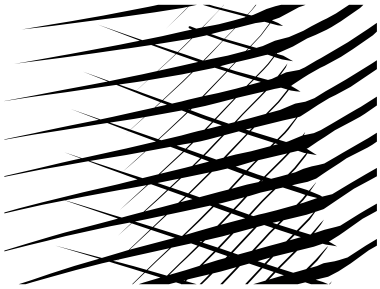


Figure 7.19b: Detail from Figure 7.18.

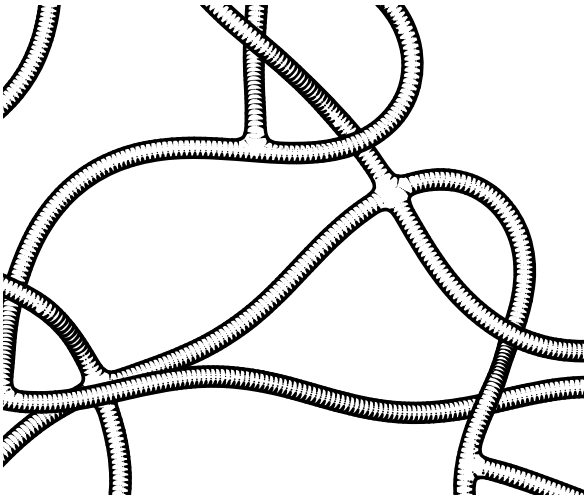


Figure 7.20a: Procedural geometry rendered with line shading.

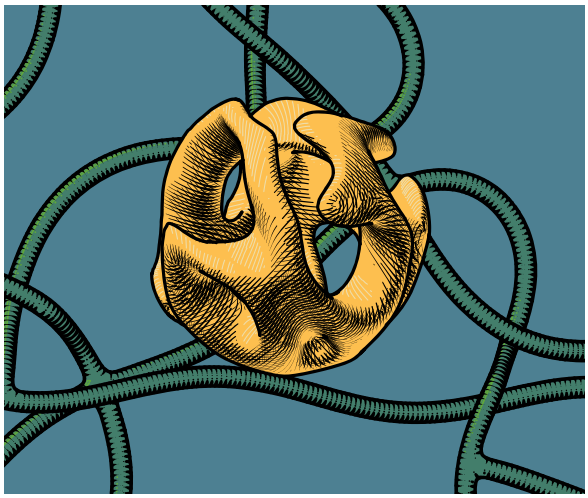


Figure 7.20b: Colorized composition using five colors and black.

8. OpenNPAR

OpenNPAR is a NPR framework that is being developed at the Department of Simulation and Graphics of the Otto-von-Guericke University of Magdeburg, Germany (Halper et al., 2003). OpenNPAR is based on the Open Inventor framework and uses its scene graph structure to manage the communication of different scene graph nodes. Each node implements a certain part of the entire process and the complete program is created by connecting several of the nodes into a pipeline.

In the following we will discuss how the algorithms described above have been integrated into OpenNPAR by giving a slightly simplified sequence of nodes in the OpenNPAR pipeline and describe roughly the information flow between them.

SoGenerate-WingedEdge

This node converts, if necessary, the raw data of the model into a Winged Edge data structure.

SoModify-WingedEdge

If desired, this node applies operations such as mesh smoothing or mesh subdivision to the previously created Winged Edge data structure.

SoGenerate-HeModelHQ

This node translates the Winged Edge data structure into a Half Edge data structure for further processing. If a polygon consists of more than three edges it is triangulated to produce a triangle mesh.

SoGenerate-CurvatureHQ

Using the Half Edge data structure as the basis this node computes curvature information for each vertex of the mesh. This includes the first and second principal curvature as well as their respective principal directions.

SoGenerate-FlowFieldHQ

Based on the curvature information a suitable direction vector is chosen for each vertex and stored in a flow field. If desired , this field can be optimized by the user.

SoWingedEdge

This node renders the Winged Edge data structure. However, in the default setting only a z-buffer representation is generated for the model.

SoGenerate-Silhouette

Using the Winged Edge data structure silhouette edges are extracted. They are stored in the Coordinates element and the LineCoordinateIndices element and will be called strokes in the following.

SoLineHidden-LineRemover

This node removes the occluded parts of the silhouette strokes.

SoLineThickener

Now, thickness information is generated for each stroke to specify the line widths. However, silhouette lines are assigned constant line width everywhere.

SoLineMergerHQ

This node stores all stroke date generated up to this point and removes it from the stroke pipeline so that the following data can be processed independently.

SoStreamlinerHQ

Using the previously generated Half Edge data structure the also previously generated flow field is integrated. The resulting streamlines are stored as strokes. In addition, information is collected that describes which vertices from the streamlines belong to which triangles (LineTriangle element). Also, for each vertex of the streamlines a tapering factor is stored (LineTaper element).

SoLineHidden-LineRemover

As done with the silhouette strokes, also the streamlines are processed to remove the occluded subset. Since we previously recorded the triangles that are associated with each streamline segment we can remove all backfacing stroke segments prior to hidden line removal. When removing streamline segments from the strokes the corresponding data in the LineTriangle element and LineTaper element have to be removed as well.

SoLine-ThickenerHQ

Similar to the regular LineThickener node this node also assigns line widths to the streamlines. By accessing the associated triangles of each streamline segment we are able to interpolate the normals for each streamline vertex which enables us to implement shading.

SoLine-DensifierHQ

This module is identical to the LineThickenerHQ except that it modifies the LineDensity element instead of the line thickness. This makes the modulation of line density as shown in Figure 6.6. possible.

SoLineMergerHQ

The previously stored silhouette strokes are now restored and merged with the hatching strokes to make both available in the pipeline.

SoLineSettingsHQ

Since there are more than one line renderers that all work with the same parameters these are stored by this node in form of a LineSetting element.

SoLineOutput-TgaHQ

This node is able to generate a pixel image screenshot in form of a TGA image with full anti-aliasing. It makes use of the LibArt library.

SoLineOutput-PdfHQ

Similar to the previous node, this node generates a vector image screenshot using the ClibPDF library and stores it in a PDF file.

SoLineShapeHQ

This node implements an interactive stroke renderer that triangulates the strokes and renders them onto the screen using OpenGL vertex arrays.

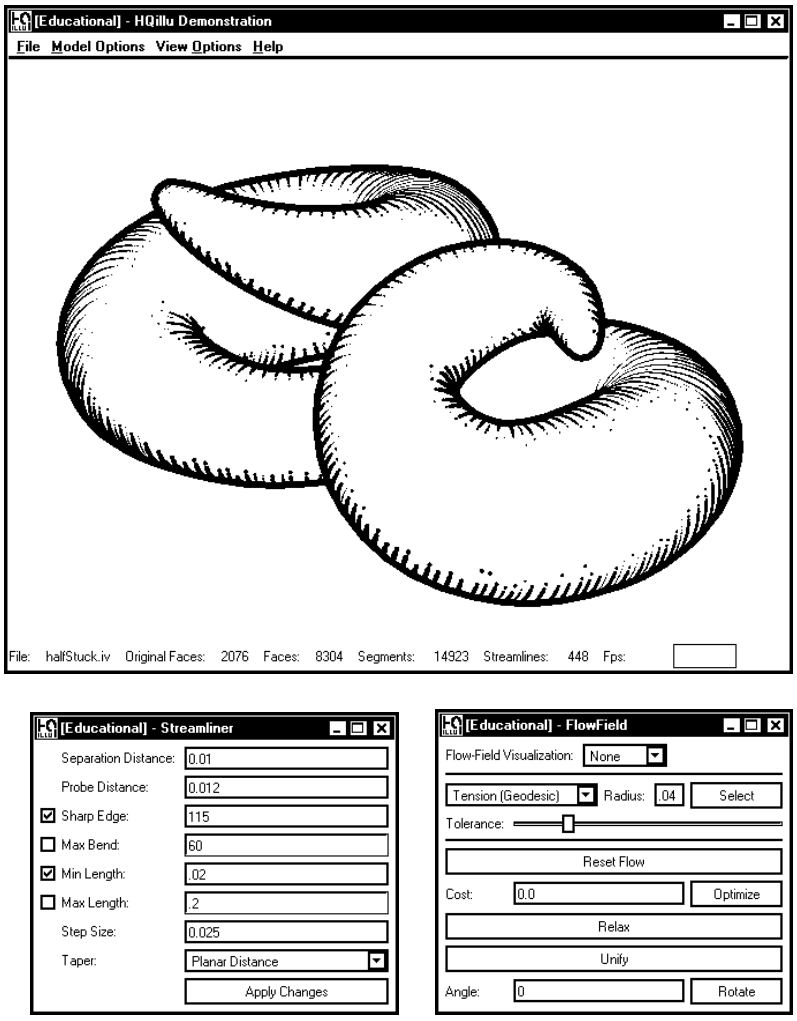
frame caption

newly developed node
modified node
available node

9. Examples

The graphical user interface was implemented in Qt based on an OpenNPAR example application (see Figure 9.1).

The user is able to manipulate most of the OpenNPAR nodes using several non-modal dialog windows. These can be used to change the parameters of the nodes while watching the resulting effects in the main window. In case operations are invoked that take some time to finish we show the progression of these operations using a progress bar at the bottom of the main window. Some status data about the scene and the streamlines are displayed there as well.



Using an entry in the file menu the user is able to invoke a PDF screenshot and store it into a file (see Figure 9.2). The dimensions of the image are chosen such that the aspect ratio and placement of the graphic in the window is preserved. However, at the same time we make sure that the format fits onto a DIN-A4 page as good as possible so that the length of at least one of the graphic's sides equals the format of a DIN-A4 side.

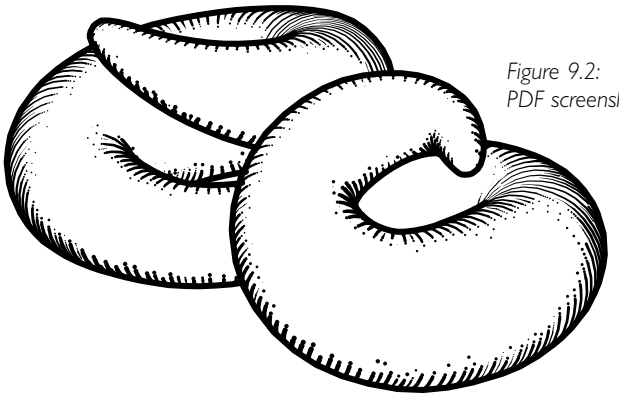


Figure 9.2: PDF screenshot.

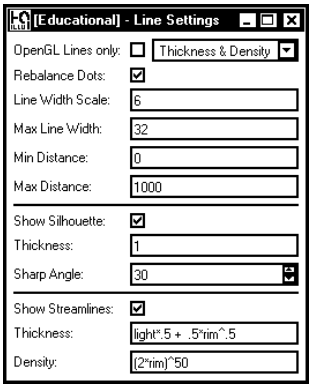


Figure 9.1: Main window and the three most important parameter dialogs of the application.

In order to explain and visualize how users are able to work with the expressions of the discussed line shading model we give an overview of an example session in the following.

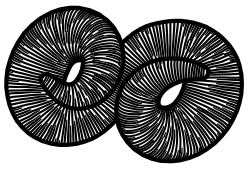


Figure 9.3:
thickness = 1
density = 1

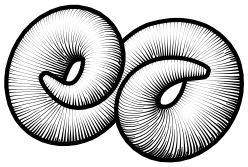


Figure 9.4:
thickness = light
density = 1

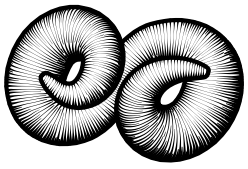


Figure 9.5:
thickness = light+rim
density = 1

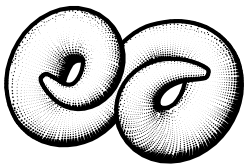


Figure 9.6:
thickness = light+rim
density = light

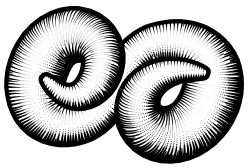


Figure 9.7:
thickness = light+rim
density = 5rim

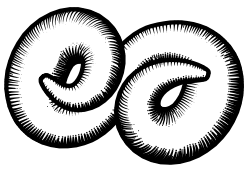


Figure 9.8:
thickness = light+rim
density = (5rim)[50]

We start with constant line thickness and constant line density. This way the appearance of the lines is only influenced by the previously discussed tapering operation and correction factor (for both see Section 6.2) as depicted in Figure 9.3. This creates the impression of a fairly constant tone. Once the user starts to modulate the line width using the illumination condition the impression of three-dimensionality and depth is created as shown in Figure 9.4. If rim shadows are added to the term this effect gets even stronger because the object contours get darker (see Figure 9.5). If now illumination is additionally used to influence the line density the image is brightened and a grid pattern is created as can be seen in Figure 9.6. In order to limit this effect to a smaller region it is sufficient to multiply the term with a constant value as demonstrated in Figure 9.7. If, in contrast, the range should be constrained from the opposite side in order to create more white space the user can, for example, take the entire term to the power of some value as shown in Figure 9.8. This yields an effect very similar to that of a highlight.

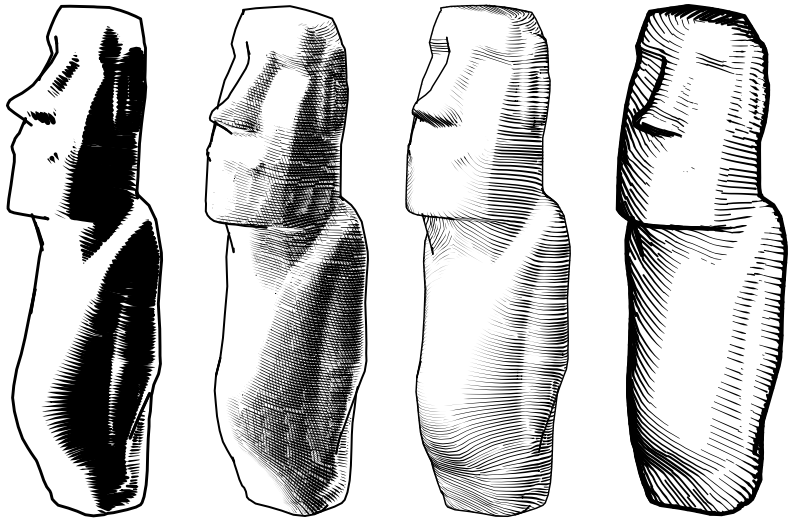
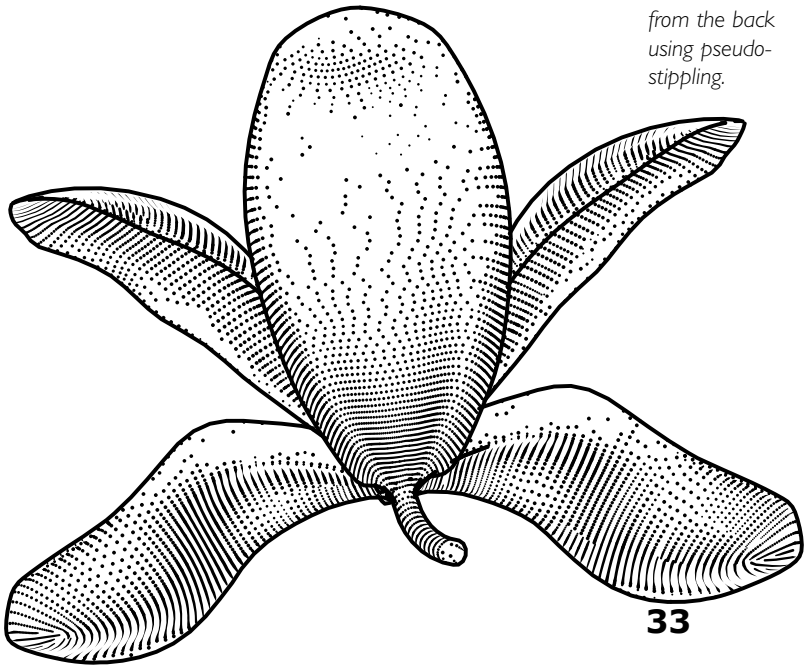


Figure 9.9: Visualization of a Moai statue using a variety of different effects.

Figure 9.9 demonstrates not only the importance of the previously discussed shading expressions but also shows the effect of varying line distances. These are important for influencing the viewer's impression of the communicated degree of detail. In addition, the combination of several layers of hatching is shown that allows to articulate the shading more clearly.

By keeping the line distance almost constant and only varying the line density to show the shading creates an effect that comes close to the one of traditional stippling (see Figure 9.10). However, a better quality of this effect could be achieved by loosening the deterministic dot placement using stochastic influences.

Figure 9.10: Flower of an orchid viewed from the back using pseudo-stippling.



10. Conclusion

As it was demonstrated with the examples the presented approach is very well suited for creating high quality illustrations. However, more work is necessary to make this easier.

A big open problem, for example, is an easy way of creating the direction field. The rather technical approach we presented above was intended to make this easy for the user. However, this way the user does not have enough possibilities to influence the field. In many cases it would be desirable to change parts of the field manually because the directions generated by the program appear rather unnatural and do not follow intuitively expected directions. At that point it would be nice to have a process that involves the user more intensively. In addition, it would be beneficial for the whole process to be able to apply several algorithms to different parts of the model. In this context we envision a 3D paint program that allows the user not only to manually comb the direction field directly on the model but also to select parts of it by painting them in order to apply specific algorithms selectively to this area. This way not only a binary application of the algorithm would be possible (i.e., apply it or do not apply it) but also a weighted influence depending on the prior painted selection process. This would rather resemble an image processing work flow where it is also possible to create a soft mask that is then used to apply a filter only to selected parts of the image.

On the other hand, there are still a few problems with the integration of the streamlines. The chosen algorithm tends to produces certain artifacts that disturb the viewer due to their clearness (see Figure 10.1). This already occurs in the 2D version of the algorithm and seems to be a fundamental problem. In such cases a streamline early collides with itself and this way blocks further streamlines as well so that the error extends to a larger area of the surface. One possibility to avoid this would be to use more care when choosing the starting vertices of streamlines. A simple yet more expensive strategy would, therefore, be to first create all children of a streamline and then choose the longest one of them. However, not only the n starting vertices of the children have to be checked then but also all possible starting vertices. This would increase the complexity of streamline integration from $O(n)$ to $O(n^2)$ and certainly lead to slowing down the process considerably. Alternatively, a simple heuristic could be tested that sorts the starting vertices created by the streamlines by how tangential their direction vectors lie compared to the original streamline. The intention would be to fill in the more homogeneous regions of the surface first. These should generally tend to contain streamlines that are more even and, hence, are more tangential.

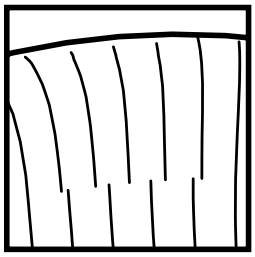


Figure 10.1: Streamline artifacts.

A completely different approach would be to hierarchically create streamlines, i.e., to recursively divide the line distances by two if no more streamlines can be created at a certain point in time. This is continued until the final minimum distance has been reached. This would also have the advantage of being able

to use the created streamline hierarchy to adjust the number of displayed lines depending on the distance of the surface to the viewer.

A different problem, finally, is related to the previous one. Since our technique works in object-space the resulting line distances are not as balanced as they would be in an image-space method. To avoid this it may be desirable to use a hybrid approach instead. This means that for the interactive work with the model the lines would still be generated in object-space. However, for the offline image generation it would be desirable in some cases to follow the approach of Hertzmann and Zorin (2000) and create the lines only after the direction field has been projected into image-space.

Literature

[Deussen et al., 1999]
O, Deussen, J, Hamel, A, Raab, S, Schlechtweg and T, Strothotte: ***An illustration technique using hardware-based intersections and skeletons.*** In Proceedings of Graphics Interface '99 (1999), Morgan Kaufmann Publishers Inc., pp. 175-182.

[Girshick et al., 2000]
A. Girshick, V. Interrante, S. Haker, T. Lemoine: ***Line Direction Matters: An Argument For The Use Of Principal Directions In 3D Line Drawings.*** In Proceedings NPAR 2000 (New York, 2000), ACM Press, pp. 43-52.

[Goldfeather, 2001]
J. Goldfeather: ***Understanding Errors in approximating principal direction vectors.***
Technical Report 01-006, University of Minnesota, 2001.

[Hertzmann and Zorin, 2000]
A. Hertzmann and D. Zorin: ***Illustrating smooth surfaces.***
In Proceedings of SIGGRAPH 2000 (New York, 2000), ACM Press, pp. 517-526.

[Interrante, 1997]
V. Interrante: ***Illustrating Surface Shape in Volume Data via Principal Direction-Driven 3D Line Integral Convolution.*** In Proceedings of SIGGRAPH'97 (New York, 1997), pp. 109-116.

[Isenberg et al., 2002]
T. Isenberg, N. Halper and T. Strothotte: ***Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes.*** Computer Graphics Forum 21-3 (September 2002), pp. 249-258.

[Hsu and Lee 1994]
S. C. Hsu and I. H. H. Lee: ***Drawing and Animation Using Skeletal Strokes.***
In Proceedings of SIGGRAPH'94 (Orlando, July 1997), pp. 109-118.

[Jobard and Lefer, 1997]
B. Jobard and W. Lefer: ***Creating Evenly-Spaced Streamlines of Arbitrary Density.***
In Proceedings of the 8th Eurographics Workshop on Visualization in Scientific Computing (1997), pp. 45-55.

[Kalnins et al., 2003]
R. D. Kalnins, P. L. Davidson, L. Markosian und A. Finkelstein: ***Coherent Stylized Silhouettes.***
In Proceedings of SIGGRAPH 2003 (New York, 2003), ACM Press, pp. 856-861.

[Leister, 1994]
W. Leister: ***Computer Generated Copper Plates.***
Computer Graphics Forum 13-1 (Mar. 1994), pp. 69-77.

[Mäntylä, 1988]
M. Mäntylä: ***An Introduction to Solid Modeling. Principles of Computer Science.***
Computer Science Press, Maryland, U.S.A, 1988, pp. 401.

[Oustromoukhov, 1999]
V. Oustromoukhov: ***Digital Facial Engraving.***
In Proceedings of SIGGRAPH'99 (New York, 1999), ACM Press, pp. 417-424.

[Pnueli and Bruckstein, 1994]
Y. Pnueli and A. M. Bruckstein: ***DigiDürer - A Digital Engraving System.***
The Visual Computer 10-5 (Apr. 1994), 277-292.

[Press, 1992]
W. H. Press, S. A. Teukolsky, W.T. Vetterling and B. P. Flannery: ***Numerical Recipes in C.***
Cambridge University Press, New York, 1992, Second Edition.

[Praun et al., 2001]
E. Praun, H. Hoppe, M. Webb and A. Finkelstein: ***Real-Time Hatching.***
In Proceedings of SIGGRAPH 2001 (New York, 2001), ACM Press, pp. 581-586.

[Rössl, 1999]
C. Rössl: ***Semi-Automatische Methoden für die Rekonstruktion von CAD-Modellen aus Punktdaten.***
Diploma Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1999.

[Rössl and Kobbelt, 2000]
C. Rössl and L. Kobbelt: ***Line-Art Rendering of 3D-Models.***
In Proceedings of WSGC'2000 (2000), The University of West Bohemia, Plzen, Czech Republic, pp. 168-175.

[Salisbury et al., 1994]
M. P. Salisbury, S. E. Anderson, R. Barzel and D. H. Salesin: ***Interactive pen-and-ink illustration.***
In Proceedings of SIGGRAPH'94 (New York, 1994), ACM Press, pp. 101-108.

[Strothotte and Schlechtweg, 2002]
T. Strothotte and S. Schlechtweg: ***Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation.***
Morgan Kaufmann, San Francisco, 2002.

[Vehmeier, 2002]
B. Vehmeier: ***Qualität in Liniendarstellungen durch lokale Informationen.***
Diploma Thesis, Otto-von-Guericke-Universität Magdeburg, 2002.

[Winkenbach and Salesin, 1994]
G. Winkenbach and D. H. Salesin: ***Computer-Generated Pen-and-Ink Illustration.***
In Proceedings of SIGGRAPH'94 (New York, 1994), ACM Press, pp. 91-100.

[Winkenbach and Salesin, 1996]
G. Winkenbach and D. H. Salesin: ***Rendering Parametric Surfaces in Pen and Ink.***
In Proceedings of SIGGRAPH'96 (New York, 1996), ACM Press, pp. 469-476.

[Zhu et al., 1997]
C. Zhu, R. H. Byrd, P. Lu and J. Nocedal: ***L-BFGS-B - Fortran Subroutines for Large-Scale Bound Constrained Optimization.*** ACM Trans. Math. Software 23-4 (1997), pp. 550-560.

[Zander et al., 2004]
J. Zander, T. Isenberg, S. Schlechtweg, and T. Strothotte: ***High Quality Hatching.***
Computer Graphics Forum (Proceedings of Eurographics), 23(3), September 2004. To appear.

Image References

[Hodges, 1989]
Elaine R. S. Hodges: ***The Guild Handbook of Scientific Illustration***.
Van Nostrand Reinhold, New York, 1989.

[Rogers, 1992]
A. W. Rogers: ***Anatomy.***
Churchill Livingston, Edinburgh • Madrid • Melbourne • New York • Tokyo, 1992.

[Tortora, 1997]
Gerhard J. Tortora: ***Introduction to the Human Body.***
Benjamin Cummings, Menlo Park, California • Reading, Massachusetts • New York, 1997,
Fourth Edition.

[Wood, 1994]
Phyllis Wood: ***Scientific Illustration***.
John Wiley & Sons, New York • Chichester • Weinheim • Brisbane • Singapore • Toronto, 1994,
Second Edition.

WWW References and Links

[ClibPDF]
http://www.fastio.com/

[F2C]
http://www.netlib.org/f2c/

[LibArt]
http://www.levien.com/libart/

[OpenNPAR]
http://opennpar.org/