Sketchy Shade Strokes

Carl Johan Sörman **Jens Hult**

Department of Computer Science
University of Gothenburg
2005

Abstract. As a contribution to the world of Non-Photorealistic Rendering, we present an algorithm for rendering sketch based shade strokes on a 3-dimensional polygon model. Animation is an important component of our scheme, thus a lot of our work has been put into the construction of a robust temporally coherent rendering system. This has been made possible by the use of a sophisticated Level of Detail structure. The model is analyzed while read into the program, extracting information such as curvature and principle directions, in order to present the model in a trustworthy manner and to bring out the features of the model. We have chosen to trade refined but few strokes for numerous expressive strokes, which produces an attractive overall picture. The algorithm is extendible in many directions and a few of these are discussed in this paper.

1 Introduction and Previous Work

Non-Photorealistic Rendering (NPR) is a branch in computer graphics where the rendering of images is not supposed to look like a photograph, but instead for instance imitate a drawing by a human. Some important topics in NPR concerns silhouettes, shading and pen strokes. There are many approaches to simulating the strokes of an artists sketch. What are the characteristics of an artistic stroke and how are these adopted to a 3-dimensional model? An introduction of various attempts at analyzing these characteristics can be read in [1], where they also present their own studies in the subject. Other approaches at artistically rendering a model are covered in [2], which shades a model by using stroke textures and investigates how silhouettes and features are drawn. [2] is a comprehensive source thoroughly discussing many of the non-photorealistic rendering techniques currently available.

The specific field discussed in this paper is how to draw the shade of a model in an NPR fashion. Our approach is based on particles on the actual 3D model, in contrast of generating textures applied to the model, which in general is referred to by the term crosshatching. In most of the existing techniques using particles, such as [3], the shades are made up of single pen strokes at each particle, but shades in sketches can also be drawn using zigzagging pen strokes. This work presents an algorithm, which renders shades with such zigzagging pen strokes in real-time, depicted in Figure 2. Our main contribution to the field is the structure of the particles and the way we use this structure to paint and animate the shades. Within the extent of our research, no other structure allows the intensity of the shades to vary during the animation, or in other words to let the intensity of the shades be constant on the screen. Instead, existing algorithms let the amplitude on their shades grow and when needed, allow new shades to pop up on the model. To clearly establish the effect, which we are trying to create, we drew reference sketches, and one of the major difficulties lies in how to transform the characteristics from the sketch into the rendering of these on a 3D model.

We also wanted to imitate an ink pen, but animation issues forced us to blend in the shades, which results in a nice smooth effect that looks more like the strokes of a graphite pencil. The ink pen is difficult to fade and the strengths of the strokes are nearly independent of pressure, whereas the strengths of the strokes made by a pencil are highly variable by both pressure and the hardness grade of the pencil. We have chosen to focus on the rendering of the shades in the sketch leaving all the other distinguishing features to existing and future algorithms.

Figure 1 An example of a reference sketch made by hand.

2 The Algorithm

The aim of this algorithm is to create sketchy shade strokes, imitating the style of an artistic sketch using a pen, rendered in real time. We have broken down the problem into four separate parts. First, since we want to animate the scene a structure is needed which supports frame-to-frame coherency. Secondly, we determine how to create the shades. This involves problems as to decide the length of the shades and to set the direction of the strokes. The third part consists of the actual painting of the strokes. The user can interact during this face by setting some visual parameters for amplitude, intensity and others. Fourth, we have to update the shading. This needs to be done every time the model has moved in respect to the camera or the light source.

Since we are working with a project leaning towards a somewhat artistic direction, it is in the eye of the beholder to determine what looks good and what does not, but by analyzing our reference sketches we have chosen to define the shadows in the following way. The shades strife towards the second principal direction at each particle point and the amplitude of the strokes are determined by the magnitude of the curvature in the first principal direction. An example of a reference sketch is depicted in Figure 1. This will help to bring out the shape of the model. The pen strokes in a wider area will be painted with greater amplitude than in a smaller one, i.e. the algorithm adapts to the size of the feature it is shading. The screen intensity of the shade aims to be constant. To keep a picture clean, we limit

the length of a shade based on a couple of criteria. The amplitude of the different strokes in a shade varies in a randomly manner; however the amplitude is still restricted by a max amplitude value determined by the curvature calculations at that particular point.

2.1 Terminology

We have adopted the general terminology of computer graphics regarding the traditional geometric components of 3D model data in real-time rendering. In addition to this terminology, we have found it necessary to introduce a number of terms for clarity in this paper.

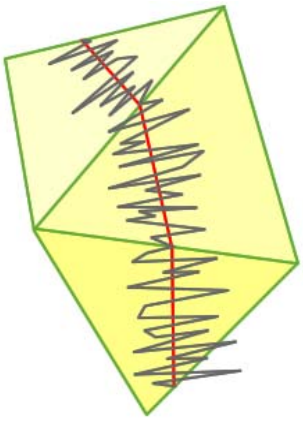

Figure 2 The picture shows three shade segments together forming a shade chain rendered in red and its shade strokes in gray, over three triangles.

- Let shade segment denote a vector within the bounds of a triangle along which shade strokes are painted.
- Let shade strokes denote the zigzag pattern along the shade segments.
- Let shade chain denote an ordered set of connected shade segments of arbitrary length.

More terms will be used throughout the paper, but since they are only used locally, they will be described in their respective contexts.

2.2 Initialize the Scene

A triangle-based model is read into the program. The model has to fulfill the following requirements; it cannot be self-intersecting, because the shade segments are positioned just above the surface of the triangles, and a self-intersecting model would cut off the segments, which does not give a very trustworthy result. In addition, every edge has to have exactly two adjacent faces, in other words, the model does not have any holes. Finally, the face normals must not point into the model, since the shading will be inside the model and not visible to the viewer. All these properties are also necessary to be achieved in order to have convincing curvature calculations of the model.

Conventional data such as vertex normals and material properties are supplied, and by analyzing these data, additional properties are stored for use in the algorithm. The area of a triangle and the three edge neighbors of a triangle are a couple of the more straightforward, but useful properties. By analyzing the geometric data, it is possible to approximate the two principal directions and their respective curvatures at each vertex.

2.2.1 Curvature Calculations

The principle directions of a surface can roughly be described as two orthogonal vectors in the tangent plane of a point on the surface, where the first principle direction points in the direction of maximum curvature and if the surface has positive Gaussian curvature at the point, then the second principle direction accordingly points in the minimum curvature direction. Otherwise, if the point has negative Gaussian curvature or perhaps more easily comprehensible a saddle shape, then the second principle direction is the lesser of the two extremes. Since we have a model represented by a volumetric point cloud, all connected by triangles, which in a sense only approximates the worldly shape or any imaginative shape, which we try to present to the viewer, it is impossible to also have exact curvature calculations. We settle for an approximation at every vertex, based on the 3-dimensional data that is provided. A few algorithms for finding curvature and principal directions of models consisting of triangles are compared by [4], with the outcome that in our particular case any of them will do. Since the nature of the effect, which we are trying to create, does not put any stronger requirements on accuracy, we settle for a rougher approximation of the principle directions.

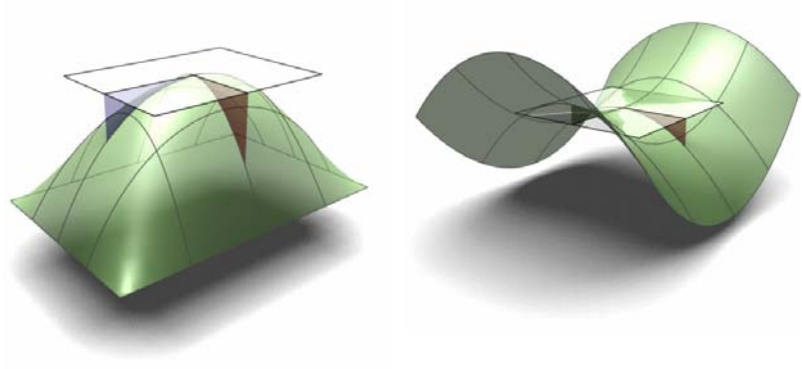

Figure 3 The left surface illustrates the occurrence of a positive Gaussian curvature at the point where the white plane tangents the surface. Conversely, the right surface illustrates a negative Gaussian curvature. The white plane is the tangent plane given by the normal vector at the point. The red and blue vectors in the white plane are the two principle directions at the point.

The fundamental steps of this algorithm consist of first creating a more accurate approximation of the normal vectors at each vertex. Rather than using the generally adopted interpolated vertex normals used in lighting calculations in real-time rendering, it is possible to make a more precise approximation to the true normal by weighing the adjacent face normals by their angles at the vertex and their respective triangle area, of which we are trying to find the unit vertex normal.

In the next step, we pick two orthonormal vectors in the tangent plane given by the above-approximated normal. These three vectors span a local coordinate system in which all calculations are made. The objective now is to estimate the Weingarten curvature matrix, which is obtained by setting up a system of equations solved by a least square method. Details are discussed in [4], where also higher accuracy algorithms are discussed and compared. There is also the issue of umbilical points, i.e. where the principle directions are not uniquely defined. For this case, we create two orthogonal reference vectors and align the principle directions accordingly. The importance of curvature and principle directions is discussed in [5]. The calculations are performed at each vertex of the model and results in four additional values stored at every vertex, namely the two principle directions and the corresponding curvature values.

2.2.2 Creating a LOD Structure

A Level Of Detail (LOD) algorithm is indispensable in order to obtain a satisfactory result when rendering the shade strokes. Two arrays of size 2^n are required to make the shade strokes adaptive. We found that setting n to eight, resulting in two arrays containing 256 elements gave a satisfactory outcome. The elements of the first array, LODU, consist of random decimal numbers in the interval [-1, 1]. The elements of the second array,

LODV, are comprised of random decimal numbers in the span of [0, 1]. The total sum of the elements in LODV is also stored. To randomize the appearances of the shade strokes further, we create five different sets of these arrays and assign one array and one random start index to each individual shade segment. Every shade segment is also assigned a LOD value, which determines the number of particles that is used in the shade segment, and the LOD arrays are used to look up their positions. A thorough description of the use of these arrays will be discussed in the next section of this paper.

2.3 Rendering the Scene

To achieve the desired effect, a few separate parts of the rendering engine can be distinguished.

2.3.1 Silhouetting

To create the outline of the model usually drawn when doing a sketch, a silhouetting algorithm needed to be implemented. This algorithm draws lines of features orthogonal to the viewing direction. Many different silhouetting algorithms has been proposed such as [6], [7], and any one of them can be used provided that it produces the sought after effect. Another set of lines also interesting when presenting the features of a model by a sketch are the so called suggestive contours [8], which are used to convey the shape of these features to the viewer. A typical such feature is a crease. The motivation for using silhouettes is obvious, since to convey the shape of a 3-dimensional model on a 2-dimensional medium, like a paper, one generally outlines the shape. In our experience, to use silhouettes, although they are not necessary, does contribute significantly to the perception of a sketch.

2.3.2 Rendering the Underlying Model

From our experiments, we found that the most suitable way to render the underlying model is simply to omit all light calculations and render it filled using the diffuse component of its material. Another pleasing method is to render it in the color of the background, typically white. This is perhaps the most accurate approach to the simulation of a sketch. Also worth mentioning, is the possibility to use a more sophisticated algorithm, for instance simulating aquarelle coloration, described in [2], [9].

2.3.3 Shade Strokes

This is the focus of the paper and where the major effort has been put. Of performance and appearance reasons, we have chosen to use simplistic pen strokes with invariant pen shape and texture. To accomplish for instance a carbon pencil stroke they need to be texturized where also pressure and swiftness are of interest. The ink pen on the other hand gives a more or less constant stroke regardless of pressure and to some point speed. In our case more and simpler strokes is a better choice over fewer and more

sophisticated strokes. For the interested, [2] investigates the topic of texturized strokes. In [10] a possibility is discussed on how to draw graphite pencil strokes onto 3D polygonal models. They thoroughly investigate the different variations of pencil hardness, pencil sharpness, paper qualities and the relation between pencil and paper. The notion pencil sharpness is here referring to the different states a pencil is in, for instance the pencil can be just recently sharpened by a pencil sharpener or be very blunt which obviously affects the appearance of the strokes. The pencil and paper relationship is determined by pressure, pencil angle, pencil rotation applicable when the pencil is not exactly round, pressure distribution and even finger position.

Another approach is the one presented in [11], where the actual strokes are completely left out and the focus is instead concentrated on smudged charcoal. This is perhaps not entirely convincing when compared to a real world charcoal sketch, nonetheless the perception of the effect is unmistakably that of such a sketch, and would possibly make a nice combination with our approach contributing with the strokes, which in turn lacks the smudge effect.

2.3.3.1 Generating the Shade Chains

In the lighting calculations, each triangle is given a shade value initialized to its max value, which we compare to the diffuse light value of the triangle. This means that the triangle is completely unshaded when the shade value is maximized. If the light value is less than the shade value, the triangle needs to be further shaded and if it is greater, then consequently it needs to be less shaded. To shade a triangle it needs at least one shade segment, and the amount that a shade segment contributes to the shade value of a triangle is determined by its length relative to the square root of the area of its parent triangle. This relation, denoted StaticShadeValue, is used since we compare a length to an area and we want a linear behavior.

Another factor in the shade equation is the strength of the shade strokes, i.e. the pencil pressure. To achieve the randomness in shade strength seen in real world drawings, each shade chain is assigned a random max alpha value; the higher value the stronger pencil pressure. If the joint shade value of the existing shade segments in a triangle does not satisfy its shade needs, then a new shade segment is required. This procedure is more explicitly described in paragraph 2.3.3.3.

To create a shade segment, we create a shade chain starting at a random point on a random edge of the current triangle and then following the interpolated second principle direction plus some random disturbance which size is adjustable by the user. Depending on which edge in the triangle we are starting at, the interpolated principle direction may point out of the triangle, which is handled by mirroring the direction around the edge it enters and since the original direction vector lied in the plane of the triangle, so does the mirrored one. Another consequence is that the direction will be pointing into the triangle. This direction is the shade chains general direction and all shade segments in the chain follow this

direction in the sense that the scalar product of the general direction and the direction of a shade segment is never negative. If the interpolated principle direction suggests a faulty direction then simply reverse it.

In order to make the generation of shade chains as general as possible and not all start in the same first updated triangles we randomize the order in which we iterate through the triangles. This is made possible using a random prime number and ensuring that it is not a divisor to the number of triangles. It is easily verified that the triangles will all be updated once, if we use this number times a counter modulus the number of triangles as look up index, when iterating through the array of triangles. There are two criteria for ending the recursion of a shade chain creation. The first is that if the triangle, which is to be entered by the shade chain, has a shade value, which is already fulfilled. Secondly, it is determined by the normal at the start point and the normal at the current triangle edge, namely when the normal deviates more than a certain amount from the start points normal, the chain ends. This makes for a more adaptable and general condition than ending at some quantity of shade segments, i.e. triangles.

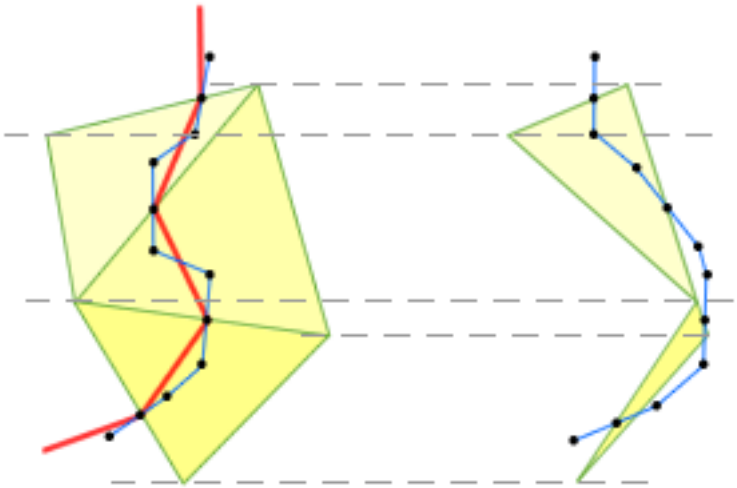

Figure 4 The automatic generation of Bezier points (black dots) over a few shade segments. The red lines represent the chain of shade segments and the blue lines connect the ordered Bezier points. The figure shows the same Bezier points seen to the left from a direction close to perpendicular to the faces of the model and to the right the view position lies in the plane of the middle triangle.

Each shade segment is assigned an amplitude value of the shade strokes at the start and end, which are set by the inverse of the interpolated curvature of the first principle direction along the respective triangle edges. We use the inverse since a smaller value should have wider strokes and vise versa.

To put it geometrically, this lets the shading adjust to the shape and size of the model, where delicate features are shaded by delicate strokes and accordingly large areas by wider strokes.

In order to make the shade chains more pleasant, we generate cubic Bezier curves [12], based on the shade segments. Each segment defines four control vertices, where the first and the last coincide with the start and end point of the segment. The second control vertex is created using a vector equal to adding the direction of the previous segment and the current one. Then if the two triangles of the two segments are convex, we project this vector onto the plane defined by the normal of the common edge between the two triangles. Otherwise, the vector lies in the plane of the triangle. The control vertex position is now established as the projected vector of length equal to 1/3 of the shade segments length. The procedure is repeated analogously to the third control point using the next shade segment instead. Special care is of course needed for the end segments of the shade chain where any method in the same spirit as the one described above is applicable. Since the nature of the shade strokes suggests a quite intense frequency, we also get a high tessellation of the Bezier curves.

2.3.3.2 Rendering the Shade Strokes

The rendering in our case iterates over the shade segments in each triangle and here the LOD structures created earlier come into play. We create a right-handed 2-dimensional coordinate system in the plane of the triangle, spanned by the segments direction vector and a vector orthogonal to both this vector and the face normal of the triangle. Let V denote the segment direction and U the orthogonal direction in the plane of the triangle. The unit length of the V axis is the length of the shade segment. The unit length of the U axis is one since we want to interpolate the amplitude along the segment. In order to describe further, how we generate the actual shade strokes, it is necessary to address the LOD structure and the assignment of a LOD value to a shade segment. A key observation, in justifying the use of a LOD structure, is that the shade strokes are by nature view dependent. For instance, two shade segments of different length should not have the same amount of zigzags, if they are at the same distance from the viewer. A shade segment seen up close will need more zigzags than when viewed from a greater distance. Using this we establish a relation between the distance from the viewer to the segment and the length of the segment, specifically by letting LODValue denote the relation we get

$$\text{LODValue} = \begin{cases} 2^{1/\text{RSL}\cdot\text{UDV}} & \text{if } \text{RSL} > 1/\text{UDV}\cdot 6 \\ 64 & \text{otherwise} \end{cases}$$

$$\text{, where RSL} = \sqrt{\frac{\left|\text{Segment}^{\perp \text{View Direction}}\right|}{\left|\text{SegmentPos} - \text{ViewPos}\right|}}$$

RSL is an abbreviation for Relative Segment Length and UDV is an abbreviation for User Defined Values. The LODValue is then used to establish the LODLevel of the segment, which is 2^n, where the integer

$n \in [1, 6]$ and $2^{n-1} < \text{LODValue} \leq 2^n$. Both these values are utilized in the rendering of the shade strokes. The actual rendering is done by sending a series of vertices to the graphics-rendering pipeline, which then connects these by straight lines, resulting in a continuous zigzag pattern. Let us describe how these vertices are generated in the coordinate system mentioned above. In fact, the resulting vertices do not ultimately lie in the coordinate system at all since we will use the V component of the points as parameter value to the associated Bezier curve. The very first vertex is positioned at the origin of the coordinate system. The V component of every k point is fixed at

$$\frac{\sum_{0}^{\text{LODLevel}*k} \text{LODV}}{\text{LODVSum}}, \quad k = 0,1,2,..., \frac{256}{\text{LODLevel}} - 1$$

These values lie in the span [0, 1), which then is sent to the Bezier algorithm returning points in world coordinates. For efficiency reasons, we use a forward differencing algorithm even though our parameter points are not uniformly spaced. This simplification is not really an issue since in the end, the mean distance between the points is the one used in the algorithm and if at all detectable, it adds to the desired perceptual roughness of shade strokes.

The U component requires a little more effort. We want to have a LOD algorithm that seamlessly transits between the LODLevels, without any pop effects. This is achieved by first letting every second point be fixed and equal to the k*LODLevelth element of LODU, where k = 0, 2, 4, …, 256/LODLevel – 2 and LODU as stated above in paragraph 2.2.2, is a value between –1 and 1. The odd points are interpolated from the straight line between two even points to the k*LODLevelth element of LODU, where k = 1, 3, …, 256/LODLevel – 1. This can be written as $U_k + t*\text{LODU}[k]$, $t \in [0, 1)$. For clarity, see the example depicted in Figure 5. U_k is the U component of the point along the line, between the two surrounding even points, with V-value equal to kth V point. The value t used in the interpolation is the LODValue compared to LODLevel, namely

$$t = 2\frac{\text{LODValue} - \text{LODLevel}}{\text{LODLevel}}$$

This creates a smooth transition between LODLevels and makes for an appealing effect when the object moves around relative to the viewer. The U value is multiplied by an interpolated amplitude value along the segment and transformed into world coordinates. There is also the issue of the strength of the shade strokes and this is where the shade segments' max alpha value is utilized.

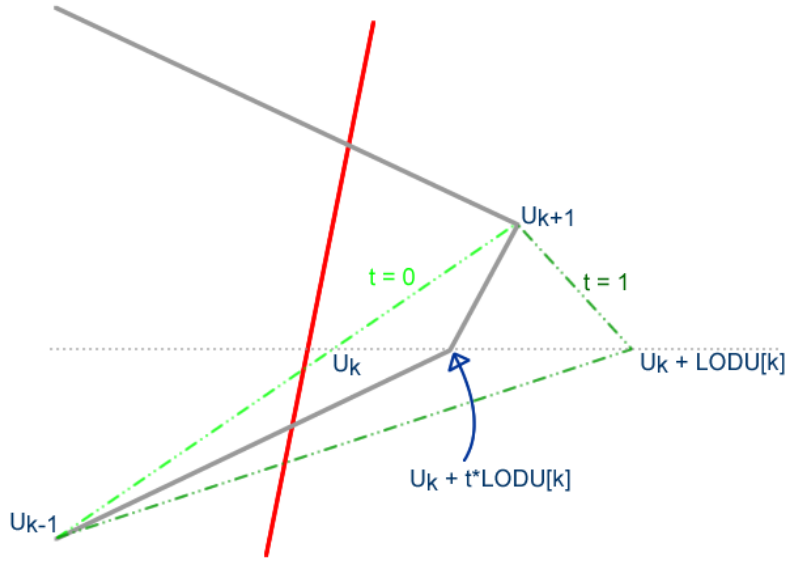

Figure 5 A close-up over a shade segment (red) portraying the interpolation of a point along some shade strokes (gray solid) which extremes are when t = 0 (light green dashed) or t = 1 (dark green dashed), though at this particular moment t is somewhere in between.

2.3.3.3 Update the shades

The update phase behaves exactly like the generation of the shade chains described in paragraph 2.3.3.1, but is described here in more depth. First, we go through all the triangles and update their light value. Then we iterate through each triangle again, but this time in a randomized order. This time we calculate their shade difference, which is the difference between their light value before and after the update. If the shade difference is smaller than a fixed value, we do not change the shade of that triangle and move on to the next one instead since the difference is not noticeable. If the opposite occurs, there are two possibilities. Either the shade difference is positive which means that the triangle has to be further shaded, or it is negative and the triangle has to be less shaded. We will explain the case where the triangle is further shaded since the two algorithms handling of the two cases are analogous, but where the following is a little more complicated.

We start by adding all the segments static shade value times their max alpha value to check if the existing shade segments can fulfil the triangles shade need. If they cannot, we set each segments shade value to its maximum, and then we generate a new shade chain as explained in paragraph 2.3.3.1. If they can, we calculate the following values

$$TotalAlpha = ShadeDifference / ShadeSum$$
$$AlphaAddition = TotalAlpha / SIT$$

where TotalAlpha is the total value which has to be spread over the segments in the triangle, ShadeSum is the sum of all the segments StaticShadeValue and SIT is the number of Segments In the Triangle. Using a random number, we iterate through the segments and for each segment where the alpha value is not equal to its maximum alpha value we add AlphaAddition to the segments alpha and subtract AlphaAddition from TotalAlpha. If a segment exceeds its allowed maximum value, the difference is added to the TotalAlpha. Eventually, when TotalAlpha is drained, the algorithm will terminate since we have checked before that the existing segments can fulfil the triangles shade need.

2.3.3.4 Manipulating the Shade Strokes

For generality, we let the user determine various settings, which alters the way the shade strokes are rendered. Some of these are self-explained while others require further clarification.

- Amplitude
- Amplitude by Curvature
- Intensity
- Shadow Density
- Maximum Chain Length
- Trueness to Curvature
- Line Width

The Amplitude determines the width of the shade strokes while the Amplitude by Curvature specifies how much the curvature influences the width of the shade strokes, from nothing to completely. Intensity affects how fast the LODValue grows. Shadow Density influences the amount of shade a single segment contributes with, which has the effect that a triangle requires either more or less segments to fulfill its shade needs. Maximum Chain Length sets maximum normal deviation between the normal at the start of a shade chain and one at the end. The direction of a shade segment is determined by the second principle direction plus some randomness, and Trueness to Curvature adjusts their proportions.

2.3.4 Extensions to the Algorithm

There are many different possibilities when simulating the inherent flickering of stop-motion animation, though we will not go into the general aspects, but instead mention how this effect is applied in our specific context. One way to simulate the effect is by adding a random amount of oscillation to the position of the vertices of the shade strokes at a constant rate in time. To add further to this effect we can randomize over the different LODV arrays over time. Another potential in the structure allows

the segments to jump around along the edges they are connected to, contributing to the flickering effect.

There are more sophisticated ways of rendering silhouettes in a Non-Photorealistic manner, some highly applicable to our purposes of human generated sketch strokes by analyzing the characteristics like speed and accuracy. We have chosen not to include any of these algorithms, for instance [13] and [14], in our project since they really are superfluous to what really is the subject, although we are convinced it would produce an attractive result.

Another interesting experiment would be to let the user determine the Shadow Density discussed in the previous section, but this time not in a global way. Instead, by adjusting it locally the user should be able to bring out the shape of the object even further. How this can be realized is left to the reader but perhaps a good approach is to let the user encapsulate the region of interest by various volumetric shapes. Also imaginable is the existence of some geometric property that suggests Shadow Density variations, which is also left for future investigations.

2.3.5 Limitations of the Algorithm

There are a few limitations to the approach proposed in this paper, though not impossible to work around. One drawback is that a shade segment only affects the shade value of the triangle it passes through rather than affecting the triangles its shade strokes cover. A small triangle may still have relatively small curvature, possibly forcing the shade strokes to spread over several triangles. This has the effect that when a higher resolution model is used, too many shade chains are generated. This can be somewhat circumvented by adjusting the Shadow Density value but still only up to the minimum amount of one segment through each triangle. Generally, this is not enough so a more inventive scheme is required; we propose two. Only wider strokes of the schemes will be covered, omitting details.

The first possibility that comes to mind is to create a LOD structure of the model and perform lighting calculations on a low-level polygon model of the actual model. This way we can assure that, every model get a proportionate amount of shade chains. The shade chains still lie in the original model maintaining the properties of the algorithm we above propose. A modification to the generation of the shade chains will be needed as to where in the full level polygon a chain should be created.

The second approach is based on B-splines. By importing, instead of the traditional polygonal model, a B-spline model or similar parametrically defined model, we can tessellate it at initialization time creating a polygon model with triangles of appropriate size, i.e. of a size suitable to our algorithm. Another benefit from this is that we get true curvature and principle directions at any point on the surface. The Bezier curves created in our design suffers from a few less attractive features, in particular when the polygon model is based on triangulated rectangles. This creates a kind of s-shape of the Bezier curve when a shade chain continues through the

diagonal of a rectangle, which depends on the automatic generation of the Bezier control points, depicted in Figure 4. This unwanted property can be avoided by using the B-spline representation. Instead of using the segments to generate Bezier curves, we create a B-spline curve on the B-spline surface which interpolates the closest points on the surface of all the start and end points of the segments of a shade chain. All these properties together vouch for strong consideration when choosing a generalization scheme.

Another issue in our algorithm is the planar shade strokes, which when rendered over an edge of the triangle it goes through, it may disappear into the model, if a neighboring triangle is concave with respect to the one it passes through. Consequently, it is rendered in the air if the triangles relation is convex. These artifacts, particularly the second one, generally add to the sketchiness of the shade strokes as if the artist sometimes shaded outside of the contours of the object.

3 Conclusion and Future Work

This paper introduced a new technique to render shades in a NPR fashion using particles on the surface. We achieved our main goal, which was to render a model, as described in the Algorithm section, in real time with some interactivity. What we believe to be our primary contribution to the field, was that by using a LOD structure, the camera can move closer towards or further away from the scene and still maintain a frame-to-frame coherency, without letting any popping of lines occur, while maintaining a constant screen intensity. We have handled lighting by letting the shades fade in and out. All shade strokes are generated completely automatically, adapting to the supplied model by analyzing its shape. The shade strokes can be manipulated to fit many different variations.

In future work we would like to speed up the algorithm, possibly by using hardware supported vertex shaders. Another potential speed up technique we would like to try is to split shade chains when a segment in the chain does not contribute with any shade and merge chains or continue on a chain instead of making new chains.

Special care is needed when developing a culling algorithm. Since the strokes sometimes are rendered outside the object, we have a problem with this. We cannot just omit back facing triangles, because popping would then occur when they become front facing. The straightforward way to solve this problem is to check if the strokes on a back-facing triangle cross a silhouette, but we estimate this to be too expensive. We see possibilities to create an efficient culling algorithm using the B-spline structure mentioned in the previous section. Finally, our algorithm lacks the ability to apply the effect to dynamic models, animated by a bones structure, which would be a valuable extension.

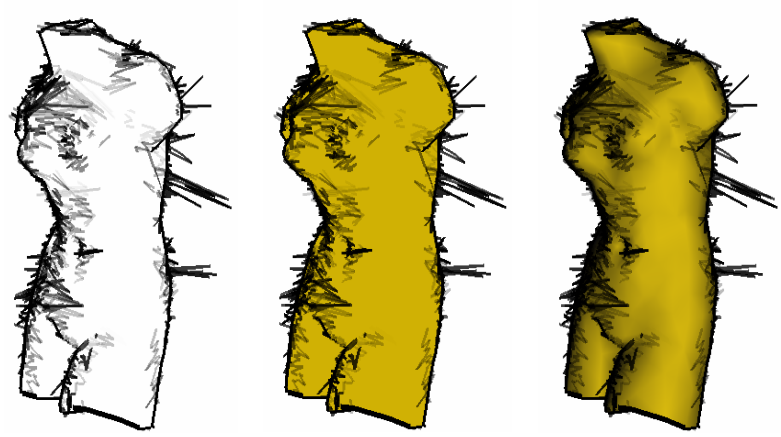

Figure 6 Three result plates, displaying the classic Venus model rendered using our algorithm. From left to right the underlying model is rendered in the background color, an unshaded color and using Gouraud shading. Note the shades appearing from behind the model, which high amplitude is due to flat surroundings around some vertices. Most of these features would not be as pronounced, if the curvature calculations utilized even more information of its local surroundings or if the curvature gets less influence over the amplitude of the shades. These features are deliberately emphasized for illustration purposes.

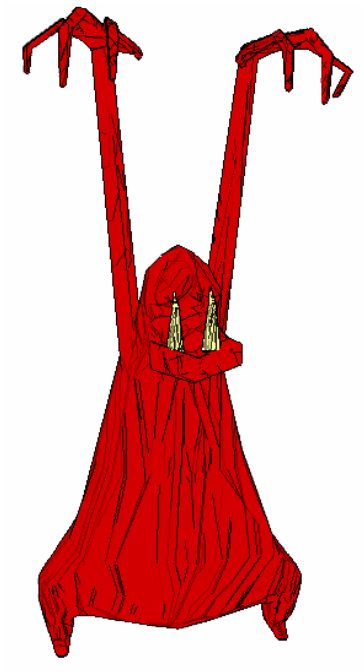

Figure 7 An illustration of the shade segments connected into shade chains on an example model. Observe that the chains follow the shape of the model.

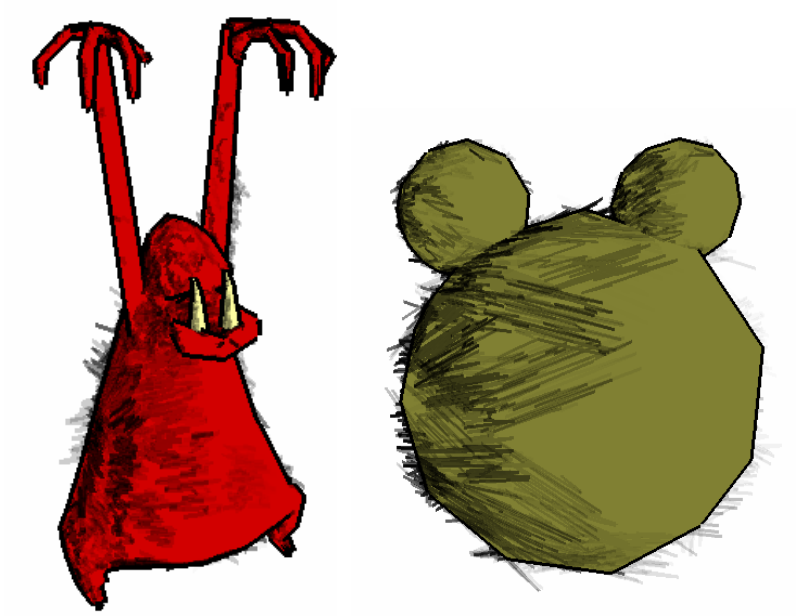

Figure 8 The left result plate shows a general example of the outcome of the algorithm, which can easily adapt to models which are greatly varying in triangle size. The right plate is a clear example of how curvature determines shade amplitude, where all spheres are identical except for scale.

4 References

[1] Christopher G. Healey, Laura Tateosian, James T. Enns, Mark Remple. Perceptually Based Brush Strokes for Nonphotorealistic Visualization. ACM Transactions on Graphics, Volume 23, Issue 1, January 2004.

[2] Thomas Strothotte, Stefan Schlechtweg. Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation. Morgan Kaufmann Publishers Inc, 2002.

[3] Derek Cornish, Andrea Rowan, David Luebke. View-Dependent Particles for Interactive Non-Photorealistic Rendering, Proceedings of Graphics Interface 2001, p.151 – 158, 2001.

[4] Jack Goldfeather, Victoria Interrante. A Novel Cubic-Order Algorithm for Approximating Principal Direction Vectors. ACM Transactions on Graphics, Vol. 23, No. 1, January 2004.

[5] Ahna Girshick, Victoria Interrante, Steven Haker, Todd Lemoine. Line Direction Matters: An Argument for the Use of Principal Directions in 3D Line Drawings. Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering, p.43 - 52, 2000.

[6] Aaron Hertzmann. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. SIGGRAPH '99 Course on Non-Photorealistic Rendering, 1999.

[7] J. D. Northrup, Lee Markosian. Artistic Silhouettes: A Hybrid Approach. Proceedings of the 1st International Symposium on Non-Photorealistic Animation and Rendering, p.31 - 37, 2000.

[8] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, Anthony Santella. Suggestive Contours for Conveying Shape. Siggraph, p. 848-855, 2003.

[9] Cassidy Curtis, Sean Anderson, Joshua Seims, Kurt Fleischery, David Salesin. Computer-Generated Watercolor. In Proceedings of SIGGRAPH 1997, p. 421-430, 1997.

[10] Mario Costa Sousa, John W. Buchanan. Computer-Generated Graphite Pencil Rendering of 3D Polygonal Models. EUROGRAPHICS, Volume 18, Number 3, 1999.

[11]Aditi Majumder, M. Gopi. Hardware accelerated real time charcoal rendering. Proceedings of the 2nd International Symposium on Non-Photorealistic Animation and Rendering, p. 59 – 66, 2002.

[12] J. J. Risler. Mathematical Methods for CAD. Cambridge University Press, 1992.

[13] Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, Adam Finkelstein. WYSIWYG NPR: Drawing Strokes Directly on 3D Models. Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques, p.755 – 762, 2002.

[14] Robert D. Kalnins, Philip L. Davidson, Lee Markosian, Adam Finkelstein. Coherent stylized silhouettes. Volume 22 , Issue 3, 2003.