CAD/Graphics'2001
August 22-24, Kunming
International Academic Publishers

ANIMATION METHOD FOR PEN-AND-INK ILLUSTRATIONS USING STROKE COHERENCY

Toshiyuki Haga Henry Johan Tomoyuki Nishita
Department of Information Science, The University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo, 113-0033, Japan
e-mail: {haga, henry, nis}@is.s.u-tokyo.ac.jp

ABSTRACT

Pen-and-ink illustrations are attractive in that they have greater ability than photorealistic images to omit unimportant details, to clarify shapes and so on. Considering these advantages, we have created pen-and-ink-style animations. In our method, we treat 3D models as inputs and generate pen-and-ink-style line drawings in which the strokes of the illustration express the features of the original 3D geometry. In the illustrations generated, the density of strokes and the shapes of strokes represent the tone and the shape of the 3D models. If the models are moving/changing, we keep the coherence between the frames in order to generate smooth animations. For this purpose, the strokes are stored using a simple data structure that is able to retain the information in the previous frame.

KEYWORDS: pen-and-ink illustrations, line drawings, non-photorealistic rendering

1 INTRODUCTION

For many years, the main subjects of study in the field of computer graphics have dealt with methods of generating realistic images such as photographs (i.e. *photorealistic rendering*) or ways to create those images efficiently. In recent years, however, the study of *nonphotorealistic rendering* has become popular, and we have recognized the power and the usefulness of nonphotorealistic rendering. In other words, images such as pen/pencil line drawings and paintings in an oil/water color style can now be generated automatically or semi-automatically.

The difference between *photorealistic* and *nonphotorealistic* imaging is the same as the difference between photographs and illustrations (or paintings, or drawings). They not only have different viewing characteristics, but also different purposes. A photograph is taken for recording and conveying a scene as it is, while the purpose of an illustration is to convey some specific parts of the information in a scene, and this is the characteristic that a photograph doesn't possess. Specifically, a pen-and-ink-style illustration contains not only interesting visual effects but also has a number of other advantages. For instance, it has the capacity to omit unimportant details, to clarify the objects' shapes by drawing their silhouettes, to attract interest through a simple expression and it is easy to reproduce or compress. Therefore, pen-and-ink illustrations can be found in many contexts, such as technical illustrations and architectural designs.

The goal of our work is to create a sequence of illustrations that are suitable for making an animation sequence that clearly shows the movement of the objects. The system presented in this paper creates a pen-and-ink illustration by drawing strokes (or lines) that reflect the geometry of the objects. If the models are moving/changing, the system can generate a smooth animation by using the coherency between frames.

In Section 2 we survey related works, in Section 3 we describe the algorithm for illustrations, and in Section 4 we explain the method we used to create animations. In Section 5 and Section 6, we discuss our results and conclusions, respectively.

2 RELATED WORK

In recent years, a number of systems have been developed to produce illustrations in a pen-and-ink-style. These systems can be classified into two categories, depending on the input data: *geometry-based systems*, which take 3D scene descriptions as input, and *image-based systems*, which produce their illustrations directly from images.

The main advantage of geometry-based systems [2, 4, 6, 7, 10, 11, 16, 17] is that they can produce illustrations whose strokes not only convey the tone and the texture of the surfaces in the scene, but they can also convey the 3D geometry of the surfaces. This is made possible by the fact that these systems can use the 3D geometry and the viewing information, and therefore can place strokes along the silhouettes

of the surfaces. Saito and Takahashi [11] introduced the concept called "G-buffers", which contain information about depth values, normals of faces and so on at each pixel, and produce various emphasized renderings by combining operations between several "G-buffers". Winkenbach and Salesin [16] proposed a method to generate pen-and-ink illustrations by using "stroke textures", which is mainly used for creating illustrations of buildings. Markosian et al. [7] presented a real-time nonphotorealistic renderer that deliberately trades accuracy and detail for interactive speed. Hertzmann and Zorin [2] proposed a set of algorithms for line-art rendering of smooth surfaces. In order to convey the surface shape, a smooth direction field is calculated.

Some image-based systems [3, 12, 13, 14] describe their advantages as the ability to greatly reduce the number of tasks required for geometric modeling and of specifying surface reflectance properties, the ability to allow much more complicated models to be illustrated (such as furry creatures and human faces), the flexibility to use any type of photograph or computer-generated image as an input and the ability to offer more direct user control. Salisbury et al. [14] generated elaborate images of complex surfaces interactively by introducing the idea of orientable textures, in which the strokes convey not only orientation, but texture and tone.

Since our main purpose is to generate smooth animations of moving/rotating/changing objects, it is necessary to make use of the objects' 3D geometry information. Thus, the system presented here is a geometry-based system. Using 3D geometry models, the system produces three types of intermediate images, (i) a (grayscale) tone image, (ii) a direction image that conveys the directions of the strokes on each face of the objects and (iii) a silhouette image. These three images are used to determine the locations and the shapes of the strokes on the illustrations. From this point of view, our system is similar to the system reported in [11], but our system can create an illustration at an interactive rate, and can make a smooth animation by using the coherence between frames.

Some animation methods [1, 5, 8, 19] for non-photorealistic rendering, have already been proposed. Meier [8] presented a technique for rendering animations in a painterly style, that provides for frame-to-frame coherence in animations by modeling surfaces as 3D particle sets. Kaplan et al. [5] proposed a particle-based method, which is similar to Meier's method. The strokes are represented using geograftals, which are geometry-based procedural objects. The details of the differences between our method and these methods are described in Section 4.

3 RENDERING METHOD

In this section, we explain the rendering process of a pen-and-ink illustration. Before we explain the process, we first describe several assumptions and definitions used by our system (see Fig. 1).

- We assume that the geometric model to be rendered is represented by a non-self-intersecting polygon mesh, and all of the edges belong to two adjacent faces.
- A "stroke" is represented by using a Bézier curve, and we express the objects' tone by using a collection of strokes.
- Each point on an object has a "direction" value for the strokes, and the system bends the strokes according to these values. See Section 3(2) for details of "direction".
- A "silhouette" line means the outline of an object, and a "ridge" line represents a feature line that is drawn inside of an object. These lines are parts of the edges that are used to construct the object (see Section 3(2)).
- Our system presumes the use of an infinite light source.

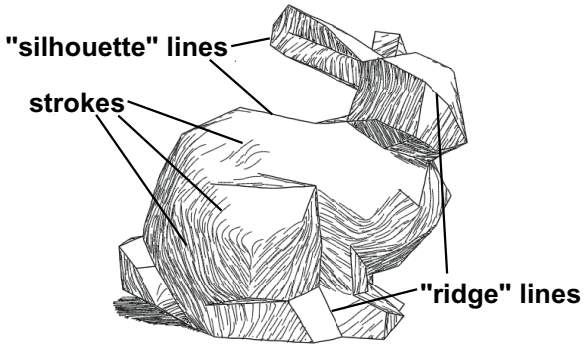


Fig. 1: Examples of each element of the illustration.

(1) Overview

The proposed system mainly aims to express the tones and shapes of an object by strokes. The density of the strokes represents the tone, and the strokes are drawn along the surfaces to represent the shapes. For example, straight strokes are drawn on a flat surface, and bent strokes are drawn on a curved surface.

The rendering process is described below.

i. The user specifies the positions of the 3D geometric models in the world coordinate system, the viewpoint, the view reference point and the light direction.

ii. For each face of the models, we calculate the tones and the directions of the strokes drawn on the face. We check whether each edge is a silhouette edge or a ridge edge.

iii. Three types of intermediate images are obtained by using the Z-buffer method (see Section 3(2)).

iv. Each stroke used to shade the illustration is generated by referring to these three images, and is added to the strokes' list (see Section 3(3) for further information on "strokes").

v. The silhouette and the ridge lines created in step **iii** and the strokes generated in step **iv** are rendered on the screen.

We describe these steps in the following subsections.

(2) Producing Three Intermediate Images

In order to generate the strokes that convey the properties of the objects, we produce three types of intermediate images (see Fig. 2). These are;

- a grayscale image that conveys the tone of the objects (the *tone-image*)
- an image that expresses the direction of the strokes at each point on the objects (the *direction-image*)[1)]
- an image that captures the silhouette and the ridge lines (the *silhouette-image*)

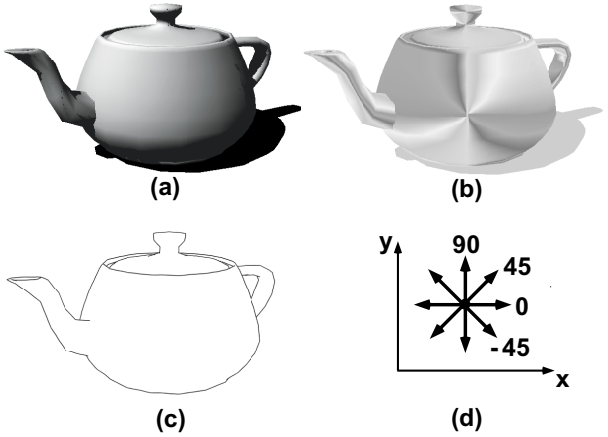


Fig. 2: The three intermediate images ((a) the *tone-image*, (b) the *direction-image*, (c) the *silhouette-image*) of a teapot, and (d) shows the "direction" in the *direction-image*.

When the positions of the input 3D geometric models, the viewpoint and the view reference point are specified by the user, the system calculates those three intermediate images by using the Z-buffer method.

Firstly, the *tone-image* is computed by rendering the models and converting the resulting image into a grayscale image, with values from '0' (black) to '255' (white). The rendering process also considers shadows that cast on the objects. To calculate shadow areas precisely, we adopt the improved Shadow Map method described in [18].

[1)] If we treat directions as colors, we can represent the direction field as an image.

Secondly, we calculate the directions of strokes on the *direction-image*. We represent the directions as values between '90' to '-90'. The values '90' and '-90' represent the direction parallel to the y-axis of a 2D screen, and the values decrease in a clockwise direction (see Fig. 2(d)) according to the projection onto a 2D screen of the vector defined by a cross product between the normal vector and the viewing vector.

Thirdly, the system checks each edge to determine whether it becomes a silhouette edge or a ridge edge or not. Before we describe the algorithm, we need to define the following terms.

> If the dot product of the face's normal vector and the vector from a point on the face to the viewpoint is positive, the face is defined as *front-face*. Otherwise, defined as *back-face*.

The algorithm for finding the silhouette and the ridge edges is as follows.

- If an edge belongs to a *front-face* and a *back-face*, then the edge is a silhouette edge.
- If an edge belongs to two *front-faces*, and the absolute value of the dot product between the two normal vectors of these faces is less than a certain value (user specified) then the edge is a ridge edge.

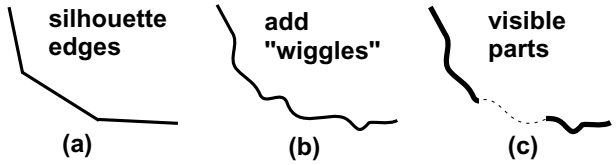


Fig. 3: Rendering the silhouette and the ridge edges into the *silhouette-image*.

We then project the silhouette and the ridge edges and produce the *silhouette-image*. At this stage we add small "wiggles" to the lines to simulate a hand-drawn appearance and we compute the visible parts of the silhouette and the ridge by using the depth information in the Z-buffer. Fig. 3(a) shows the projected silhouette and ridge edges. Fig. 3(b) shows the edges with the added hand-drawn "wiggles". And Fig. 3(c) shows the lines rendered in the *silhouette-image* with the invisible parts discarded. Therefore, the silhouette generated by this process becomes the silhouette on the final illustration. The reasons why we carry out this operation are,

- It is expensive to check and to calculate any point that divides the visible part and the invisible part, and to redraw each visible part with "wiggles".
- If we do not use "wiggly" lines in this stage, some of the strokes generated could intersect the sil-

houette or the ridge lines, resulting in an unnatural illustration.

By referring to these three images, the system generates the strokes that are required for the illustration (see Section 3(3)).

(3) Generating Strokes

Using the three intermediate images (see Section 3(2)), the system generates strokes one by one. Each stroke is generated as follows.

i. determine the position where the stroke is drawn

ii. calculate the end points of the stroke

iii. blur (or gradate) the stroke for comparison

iv. check for overdraw (in other words, check if there are too many strokes being drawn or not) on the illustration

v. if there is no overdraw, then update each image (or each buffer)

vi. if the tone of the illustration differs greatly from the *tone-image*, then generate new stroke

The basic rule for placing a stroke is as follows. We place strokes in the illustration so that the tone of the illustration *matches* that of the *tone-image*. The matching is only an approximation, because the illustration is made of black strokes on a white background, whereas the *tone-image* is grayscale. Therefore, we compare the blurred version of the illustration to the *tone-image*. In our implementation, we consider an image (called the *difference-image*) whose value at each pixel is the difference between the *tone-image* and the blurred version of the illustration. Our aim is to decrease these differences, that is, to make the value of each pixel in the *difference-image* near to '0' when the strokes are placed. The initial value of each pixel of the *difference-image* is set by subtracting the value of the *tone-image* from '255'. Specifically, we allocate the value '0' to a pixel whose tone in the *tone-image* is white (so the *tone-image* value is '255') and proportionately larger values as the tone become darker.

At the same time, we prepare the image on which the strokes are to be rendered, and we call this image the *render-image*. This image does not represent the appearance of the actual illustration, but is used to determine the distribution of the generated strokes. To achieve optimum distribution of the strokes and to reduce calculation costs, the strokes drawn on the *render-image* have no "wiggles". We initialize the *render-image* by starting with white, so there are no strokes rendered in the initial stage.

The next stroke that is about to be rendered on the screen is added to the *strokes-list*. When we create the illustration on the screen, the strokes in the *strokes-list* are rendered with "wiggles" (for details, see Section 3(4)).

Determining the stroke position

The point selected in this step becomes the center of the stroke, and the stroke extends to two directions from this point. Accordingly, we call it the *center-point* of the stroke (see Fig. 4). The system selects a pixel whose value in the *difference-image* is the largest, that is, the pixel most suitable for rendering a stroke on, and this is the pixel where the next stroke will be drawn. In other words, we regard the value of the *difference-image* as the priority, so the darker the place is, the sooner a stroke is rendered on it. Therefore, a dark place will be a place where the density of strokes is high, and indeed, we draw strokes in this order when we create an illustration by hand.

When there are several regions of the same tone, it is desirable to place strokes uniformly inside these regions. To achieve this effect, if a point is located near to existing strokes, then we set its priority to a small value. Of course, any point where the object does not exist is allocated the lowest priority, and is never selected by the system.

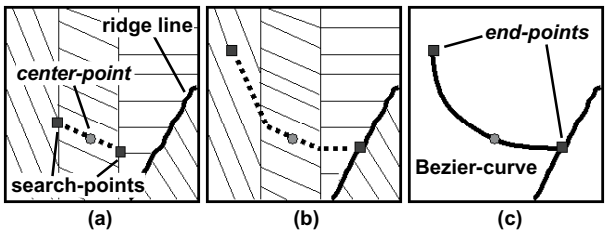


Fig. 4: Calculating the *end-points* (parallel lines on the background of each figure represent the direction on each region).

Calculating the end points of the stroke

Based on the *direction-image* (see Section 3(2)), the system calculates the *end-points* of the stroke (see Fig. 4). We begin the search from a pixel that we have defined as the *center-point*. Next, we move the search point by a unit distance (i.e. pixel size) in accordance with the direction allocated to it on the *direction-image* (Fig. 4(a)). Then we repeat this step several times (user defined) (Fig. 4(b)), and we set the final position that the search point reaches as the *end-point* of the stroke (Fig. 4(c)). If the search point reaches a silhouette or a ridge line, we automatically set that search point as the *end-point* (Fig. 4(b)) because the regions divided by a silhouette or a ridge line are considered to belong to different groups. Therefore, in this way we should avoid generating a stroke that intersects a silhouette or a ridge line. The system performs the above operation in two directions starting from the *center-point*, and thus obtains both of the *end-points* (Fig. 4). These *end-points* are the end

points of the Bézier curve that is used to represent the stroke when it is drawn on the screen.

Blurring the stroke for comparison

The *tone-image* is a grayscale image, while the color of a stroke is black. Therefore, we blur the stroke before we compare the tones in the illustration and the *tone-image*. Most of the strokes are bent, and therefore the tangential lines at each point on the stroke are different from each other. If the stroke is blurred at each point in the direction that is perpendicular to this tangential line, then the computational cost becomes very high. Instead, we define a *virtual stroke* for each of these bent strokes. The *virtual stroke* is a straight line that connects the end points of the bent stroke (see Fig. 5(a)). Each point on the stroke is then blurred in a direction perpendicular to the *virtual stroke*. From our experiments it is adequate to blur the stroke in this manner. The tone of any point on the actual stroke is dark, and the farther away a point is from the stroke, the brighter the tone becomes. The blurring function changes depending upon the tone value at a position at the *center-point* of the stroke. If the stroke is placed in a dark area, the blurred area is dark and narrow, while if it is in a bright area, the blurred area is bright and wide (see Fig. 5(b),(c),(d)). We also allow the user to specify a parameter for controlling the density of the strokes. If the value of this parameter is small, the strokes are spread sparsely in the illustration. Otherwise, the strokes are densely packed.

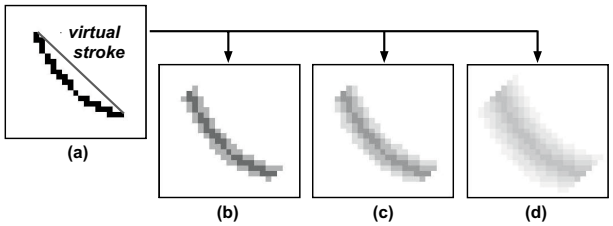


Fig. 5: (a) the original stroke, and (b,c,d) its blurred versions which are created by using different types of the blurring function.

Checking overdraw on the illustration

The system checks whether the illustration is overdrawn or not by comparing the blurred image of the stroke with the *difference-image*. This assesses whether the part of the illustration is darker than the *tone-image* when the stroke is drawn on the illustration.

In Section 3(3), we describe how we try to decrease the value of each pixel on the *difference-image* to be nearer to '0'. However, we do not try to actually make the value of the *difference-image* become exactly '0', since the filtered version of the strokes cannot match the values in the *tone-image* precisely. Instead, we try to reduce the value of the *difference-image* to below a specified tolerance value.

Therefore, we perform the checking process as follows (see Fig. 6). Consider the image (which we call the *subtract-image*) that is created by subtracting the blurred version of the image from the *difference-image*. We initialize the search point from the *center-point* of the stroke. If the value at point $S_i (i = 1, 2)$ on the *subtract-image*, which is obtained by moving the search point perpendicular to the *virtual stroke* generated on the original bent stroke, is smaller than the threshold value (Fig. 6(b)), the current search point becomes the "real" end point of the stroke(Fig. 6(c)). Otherwise, we move the search point towards each of the *end-points* of the stroke and perform the checking process again. If there is no over-darkened point on the *subtract-image*, the *end-points* of the stroke become the "real" end points.

The length of the stroke is then defined as the total distance moved by the search point.

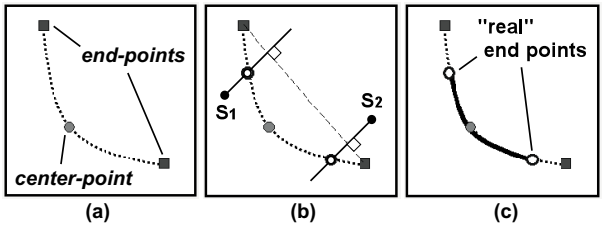


Fig. 6: (a) the skeleton of the stroke, (b) the checking process, and (c) the stroke drawn on the screen.

Updating each image and checking the termination condition

After we have generated each stroke, we check its length. Our system allows the user to specify a parameter to control the minimum length of the strokes. The system updates each image as follows.

- If the length of a stroke is smaller than the minimum length specified, then it is not drawn on the screen. As a result, the system does not update the *difference-image* and the *render-image*.
- If the stroke is long enough, then the system adds it to the *strokes-list*, and updates the *difference-image* by subtracting its values with the values of the *subtract-image*. The system also draws the stroke on the *render-image*, with no "wiggle".

If the maximum value in the *difference-image* is below the specified tolerance value, the system terminates the stroke generation process. Otherwise, the system generates a new stroke (repeating the process presented in Section 3(3)).

(4) Rendering the Illustration on the Screen

We render the final illustration on the screen by using the *silhouette-image* and the *strokes-list*. Firstly, we draw the silhouette and the ridge lines on the screen. This operation is almost the same as copying the *silhouette-image* onto the screen. Next, we draw the strokes contained in the *strokes-list*. To control the "wiggles" of each stroke, we determine a seed value [2] . This seed value is used for "wiggling" the stroke through out the animation (see Section 4). The "wiggles" help to express the impression of a hand-drawn appearance. For every stroke in the *strokes-list*, we draw a Bézier curve on the screen, which then represents the stroke. Here, we displace the control points of the Bézier curve before we draw it on the screen. The amount of displacement is calculated using two parameters, the seed value and a user-defined parameter.

In our system, for efficiency, the degrees of the Bézier curves are changed depending on the particular stage of the rendering process. When creating the *render-image* (see Section 3(3)), we approximate a stroke by using a quadratic Bézier curve, while, when creating the final illustration, we use a cubic Bézier curve. As a result, we can reduce the cost of generating strokes on the *render-image* and can realize various types of strokes on the final illustration.

4 ANIMATION METHOD

In this section, we explain our method for creating pen-and-ink-style animations. When we generate an animation, we need to consider the coherency between frames. If each frame is rendered in a photorealistic style, the generated animation exhibits coherency between frames. However, if each frame is rendered independently in a non-photorealistic style, some of the coherency is lost, since the system generates different strokes when rendering each frame. To avoid this problem, we try to redraw strokes that were drawn on the previous frame again on the current frame.

To achieve this, we borrow the ideas from the particle system [9], in the sense that strokes can also be treated as particles. There are some animation methods for non-photorealistic rendering which use the particle based brush strokes for rendering. Here, we describe the characteristics of our method by comparing it to other methods.

In Meier's [8] and Kaplan et al.'s [5] methods, the strokes are precomputed before generating the images. In other words, the number of strokes is fixed through out the animation generation process. However, when the objects are changing their shapes over time (for instance, waves in Fig. 8(c)) or when the objects are moving toward the viewer, it is necessary to add the number of strokes drawn in order to approximate the desired tones. As a result, their methods may not be able to approximate the desired tones since the number of strokes is fixed. Our method, on the other hand, allows the generation of new strokes in order to produce the correct tones.

Strokes which are computed in Meier's [8] and Kaplan et al.'s [5] methods are sticked to the surfaces of the objects and thus behave like ordinary textures. However, to produce a more artistic style animation, it is necessary to draw the same strokes differently through out the animation. In our approach, to achieve this effect, we recompute the direction field when generating new frame.

Our algorithm is as follows. Firstly, we need to consider the information about the strokes that is stored in the *strokes-list*. The information about the individual strokes drawn in the previous frame contains,

- *center-point*, which defines the position where the stroke was placed.
- *end-points* of the stroke (the end points of the Bézier curve which expresses the stroke).
- length of the stroke.
- seed value for computing the "wiggle" of the stroke.
- lifetime, which specifies the time when the stroke should disappear.
- tone value, which represents the tone of the stroke when it is rendered on the screen.

We store the seed value and the length of a stroke without modification when it is drawn on the screen, because the same stroke should have almost the same appearance throughout all of the frames. The *center-point* is transformed according to the movement of the object from the previous frame to the current frame, but the *end-points* are ignored and recalculated for each frame. This is due to the fact that the values of the *direction-image*, which effects the way that the strokes bend, are changing from frame to frame, and thus the *end-points* must be recomputed. The purpose of setting a lifetime for each stroke is to control the tone on each frame (we initialize the lifetime randomly by a value larger than the minimum lifetime depending on the stroke's position). When a stroke is just starting to be generated or is about to disappear, its tone value is set to a small number. By controlling the tone value of a stroke, we can reduce the flickering effects of the animation.

The algorithm for generating each frame of the animation is as follows.

[2] We calculate the seed value of the stroke by using the coordinates of its *center-point*. The reason why we use the coordinates of the *center-point* is to assign different seed values to strokes that are located near to each other on the illustration.

i. Before generating strokes, we update the following information about the strokes in the *strokes-list*.

- *center-point*: transform according to the movement of the object
- lifetime: decrease by a unit value
- tone value: update according to any change of the lifetime value

ii. When we start generating strokes, we choose a stroke in the *strokes-list* and check whether it can be placed or not.

iii. We treat the *center-point* of the chosen stroke as the *center-point* where the next stroke is to be drawn. For the same reason described in Section 3(3), we avoid drawing strokes near to locations where other strokes have already been drawn.

iv. In a similar way to draw a stroke on a single illustration, we calculate the *end-points* from the *center-point*, check for overdraw, and then obtain the length of the new stroke. Then we compare the length of the new stroke with the length of the original stroke. It is possible that the length of the stroke has changed, even when the stroke is positioned in exactly the same place on the object because the object is moving. Since a sudden change of length is undesirable, we discard any stroke whose length has changed at a large rate.

v. If a stroke can be drawn, we store the information of the stroke to the *strokes-list* for the current frame.

vi. When we have finished checking all the strokes in the *strokes-list*, we generate the new strokes for the current frame until the tone of the current frame satisfies the termination condition.

To express strokes which are just starting to be generated or are about to disappear, we draw them on the illustrations using thinner lines (in fact, using brighter colors, because a stroke has a width of one pixel). Although some artists use broken or dashed lines to express them, we represent these strokes by changing their thickness in the current implementation.

5 RESULTS AND EXAMPLES

The system proposed in this paper is implemented by using Java. The user can specify several parameters interactively, such as the stroke density, the magnitude of the "wiggles" on the strokes, the minimum length of the strokes and so on.

(1) Illustration

Firstly, we show the results of generating a single illustration. Fig. 7 shows examples of creating the illustrations, such as an animal and furniture. The size of each illustration is 640×480 pixels. Table 1 shows the computational time and the number of strokes applied in each illustration. The rendering is performed on a PC with a Pentium III 1GHz CPU. As shown in Table 1, the computational time is nearly proportional to the number of strokes in the illustration.

On each illustration in Fig. 7, the strokes are placed and bent along the shapes of the surfaces of the objects, and the distribution of the strokes expresses the tone of the objects and shadows.

Figure	Content	Time(sec)	Strokes
Fig. 7(a)	Triceratops	11.3	2176
Fig. 7(b)	Boat	8.7	1790
Fig. 7(c)	Face	13.9	2052
Fig. 7(d)	Furniture	20.3	4784

Table 1: The computational time and the number of strokes for each illustration.

(2) Animation

Fig. 8 shows some examples of pen-and-ink-style animations. For each example, the animation proceeds from left to right, and from top to bottom.

Fig. 8(a) is an example of changing the light direction. Our method changes the tone gradually, and this is significant, because it is very difficult to change the tone gradually when using other methods such as the texture-based strokes method.

Fig. 8(b) shows an example of zooming an object. The system draws a suitable density of strokes on each frame. It is also difficult to generate the desired zooming effect in animations by using previous methods.

Fig. 8(c) shows a more complicated example. It is a very time-consuming task for an animator to draw each frame of the animation by hand. Indeed, it is almost impossible. On the other hand, we can generate this animation easily by using the proposed method. This example also shows that the proposed algorithm can be applied even when the shapes of the objects, such as waves, change dynamically. We modelled the waves by using the Tessendorf's wave model [15].

In the animations, most of the strokes move in accordance with the movement of the object, and thus it makes it easy for the viewer to understand the movement of the object. The computational time for the first frame of the animation is the same as the time for computing a single illustration, whereas, the computational times for the other frames are about 20% shorter than the time for the first frame since the system reuses the information from the strokes in the previous frame.

6 CONCLUSIONS

In this paper, we have proposed algorithms for creating an illustration and an animation in a pen-and-ink-style. The proposed algorithms have the following properties.

- The illustration created has almost the same tone compared to the shaded models illuminated by an infinite light source.
- The strokes on the illustration express the shapes of the models well, especially when the models have curved surfaces.
- In the animation, the strokes move in accordance with the movement of the object. As a result, it is easy to understand the movement of the object.
- The coherence between the frames of the animation is mostly preserved, and the system can generate a smooth animation.

Our current system still has room for several improvements. The first of these will be to enable real-time animation. To achieve this goal, we can adopt a more efficient silhouette extraction algorithm, such as the improved Appel's algorithm [7]. The next challenges are to express the texture of the surfaces and to express various material properties of the objects (such as glass, metal, etc.) by using strokes. Also, we plan to create a more natural direction field for the strokes, and to extend the system to generate a brush-style or a pencil-style illustration by using strokes.

REFERENCES

[1] W. T. Corrêa, R. J. Jensen, C. E. Thayer, and A. Finkelstein. "Texture Mapping for Cel Animation." Proc. SIGGRAPH '98, 1998, pp.435-446.

[2] A. Hertzmann and D. Zorin. "Illustrating smooth surfaces." Proc. SIGGRAPH 2000, 2000, pp.517-526.

[3] S. C. Hsu and Irene H. H. Lee. "Drawing and Animation Using Skeleton Strokes." Proc. SIGGRAPH '94, 1994, pp.109-118.

[4] T. Igarashi, S. Matsuoka, and H. Tanaka. "Teddy: A Sketching Interface for 3D Freeform Design." Proc. SIGGRAPH '99, 1999, pp.409-416.

[5] M. Kaplan, B. Gooch and E. Cohen. "Interactive Artistic Rendering." NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering, 2000, pp.67-74.

[6] M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden, and J. F. Hughes. "Art-Based Rendering of Fur, Grass, and Trees." Proc. SIGGRAPH '99, 1999, pp.433-438.

[7] L. Markosian, M. A. Kowalski, S. J Trychin, L. D. Bourdev, D. Goldstein, and J. F. Hughes. "Real-time nonphotorealistic rendering." Proc. SIGGRAPH '97, 1997, pp.415-420.

[8] B. J. Meier. "Painterly Rendering for Animation." Proceedings SIGGRAPH '96, 1996, pp.477-484.

[9] T. Möller and E. Haines. "Real-Time Rendering." A K Peters, Ltd., 1999, pp.179-183.

[10] R. Raskar and M. Cohen. "Image Precision Silhouette Edges." Proc. ACM Symposium on Interactive 3D Graphics, 1999, pp.26-29.

[11] T. Saito and T. Takahashi. "Comprehensible Rendering of 3-D Shapes." Proc. SIGGRAPH '90, 1990, pp.197-206.

[12] M. P. Salisbury, S. E. Anderson, R. Barzel, and D. H. Salesin. "Interactive Pen-and-Ink Illustration." Proc. SIGGRAPH '94, 1994, pp.101-108.

[13] M. Salisbury, C. Anderson, D. Lischinski, and D. H. Salesin. "Scale-Dependent Reproduction of Pen-and-Ink Illustrations." Proc. SIGGRAPH '96, 1996, pp.461-468.

[14] M. P. Salisbury, M. T. Wong, J. F. Hughes, and D. H. Salesin. "Orientable Textures for Image-Based Pen-and-Ink Illustration." Proc. SIGGRAPH '97, 1997, pp.401-406.

[15] J. Tessendorf. "Simulating Ocean Water." SIGGRAPH 2000 Course Note 25, Simulating Nature: From Theory to Practice, 2000.

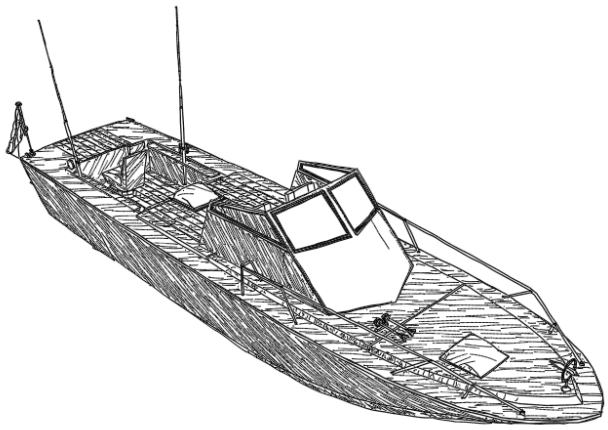
[16] G. Winkenbach and D. H. Salesin. "Computer-Generated Pen-and-Ink Illustration." Proc. SIGGRAPH '94, 1994, pp.91-100.

[17] G. Winkenbach and D. H. Salesin. "Rendering Parametric Surface in Pen and Ink." Proc. SIGGRAPH '96, 1996, pp.469-476.

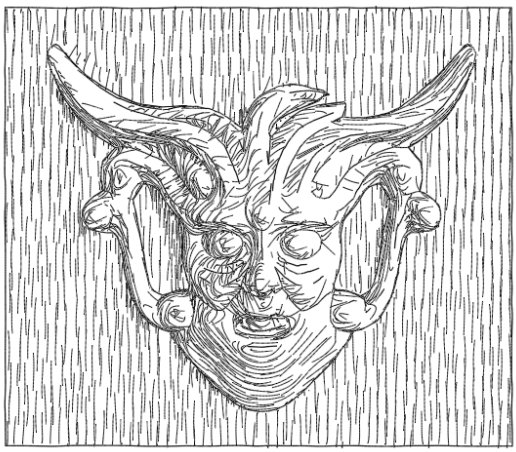
[18] Andrew Woo. "THE SHADOW DEPTH MAP REVISITED." Graphics Gems III, Academic Press, 1992, pp.338-342.

[19] D. N. Wood, A. Finkelstein, J. F. Hughes, C. E. Thayer, and D. H. Salesin. "Multiperspective Panoramas for Cel Animation." Proc. SIGGRAPH '97, 1997, pp.243-250.

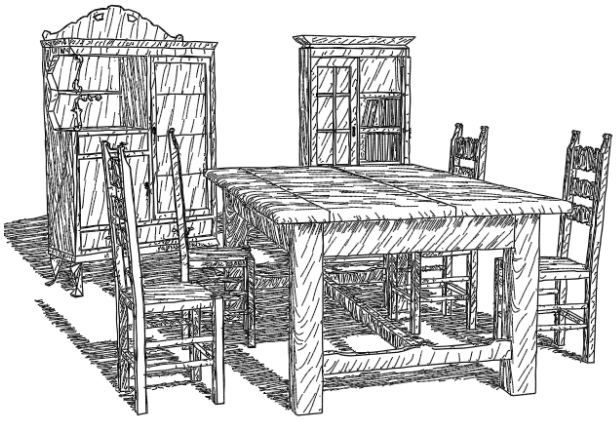
(a) Triceratops



(b) Boat

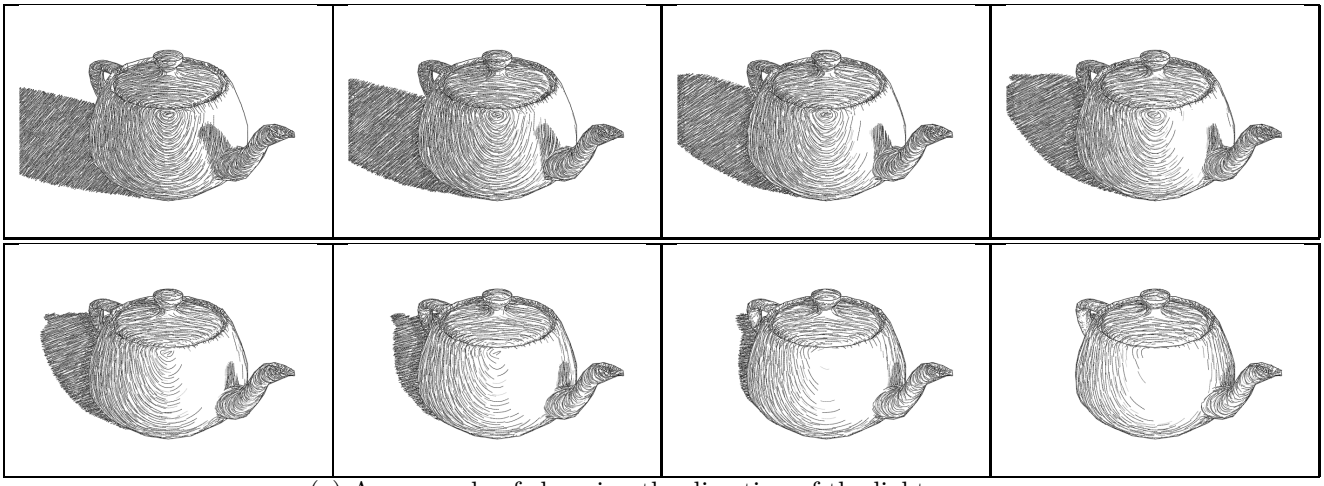


(c) Face

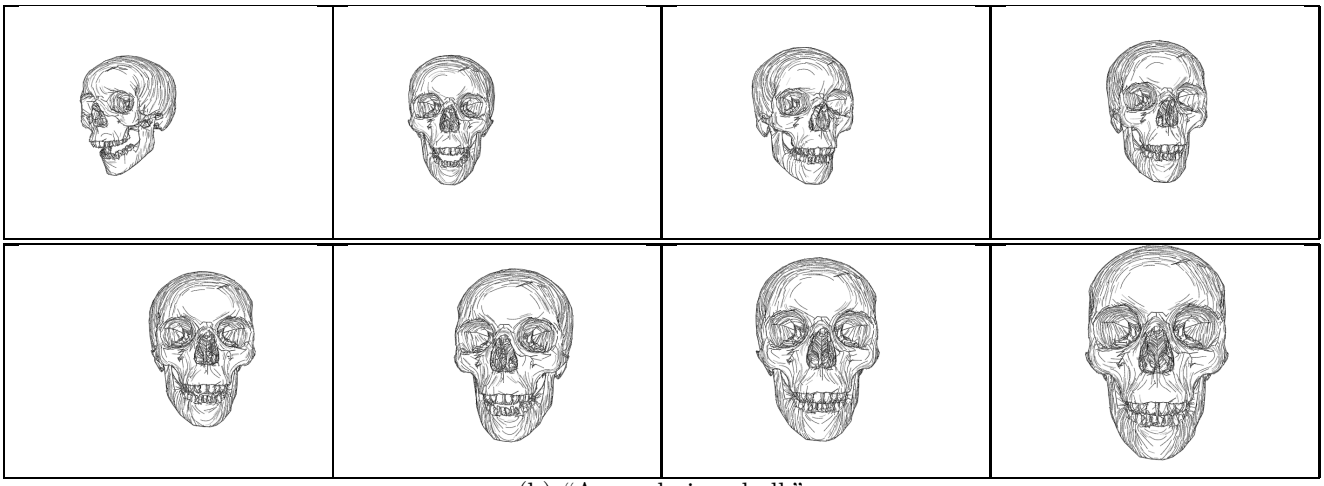


(d) Furniture

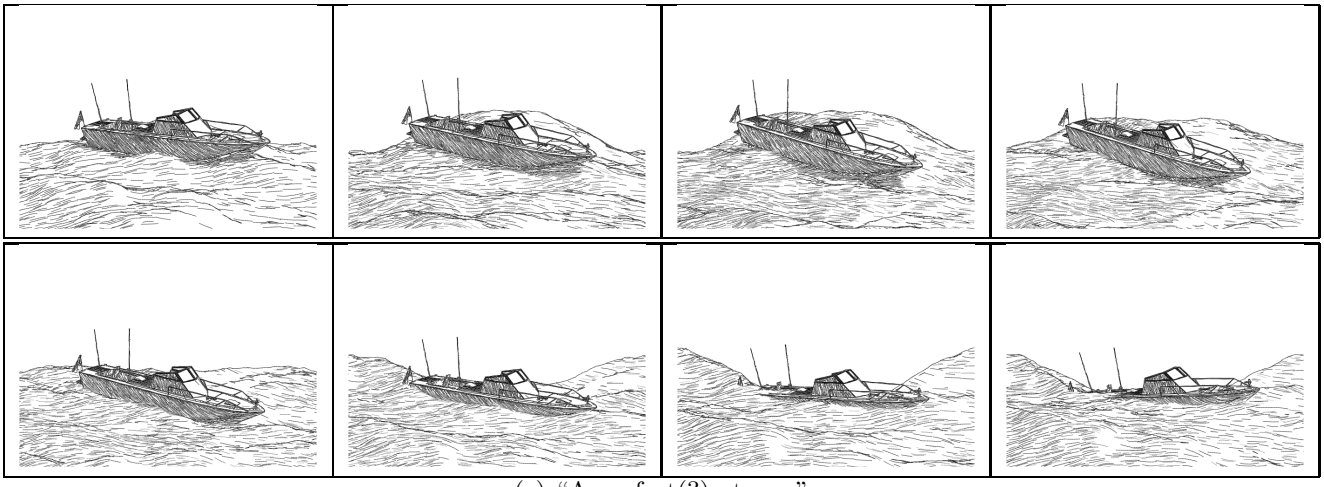
Fig. 7: Examples of the generated illustrations.



(a) An example of changing the direction of the light source.



(b) "A wandering skull."



(c) "A perfect(?) storm."

Fig. 8: Some frames of the generated animations.