Paper

Real-Time Line Drawing of 3D Models Taking Into Account Curvature-Based Importance

⟨Summary⟩ In recent years, many line drawing algorithms have been proposed for 3D shape depiction. One common drawback is that these algorithms draw only part of the necessary lines for fully depicting the shape. Cole et al.[1] have shown that these algorithms can be complementary to each other. Based on that, in this paper, we combine the previous algorithms to produce better line drawings of 3D objects. By directly superposing multiple line drawing algorithms, however, there is a risk of generating too many unnecessary lines, especially in regions where many lines are located near to each other, making them to appear over-sketched. To solve this problem, we define an attribute called "importance" for all points on each line. The importance describes which lines should be drawn in preference. In addition, we determine the width of the lines based on the number of lines in their neighborhood. Our algorithm runs on GPU, achieving real-time speed. From experimental results, our method avoids drawing excessive lines while conveying the shapes effectively.

Key words: Line Drawing, Importance, Curvature

1. Introduction

Many algorithms for Non-Photorealistic Rendering (NPR) have been proposed in recent years, particularly, line-drawing algorithms for 3D shape depiction[2]. These algorithms are very useful in cartoon rendering, industrial and medical visualization software.

There are two basic ways to draw lines from 3D shapes, the Silhouette algorithm and the Crease algorithm[2]. However, in the last few years, more complex algorithms based on curvature or its derivative[3] have been proposed, such as DeCarlo et al.[4], Ohtake et al.[5], Judd et al.[6] and Kolomekin et al.[7].

All these algorithms are biased towards certain kinds of lines and none can emulate human's drawings perfectly. They can only generate one subset of the lines that an artist would draw. Fortunately, as suggested in[1] we can combine them to generate drawings closer to those made by humans.

The problem is that just overlapping the output of previous algorithms does not yield good results. In some areas these algorithms produce similar lines or over-sketched lines, resulting in unnatural drawings that are far from the line drawings of artists (see Figure 1). Some of these line drawing algorithms include a line elimination step. It is usually performed by computing a strength value for each line and comparing it to a threshold. However, these algorithms use different methods and they are not applicable to one another.

In this paper, we propose a method for combining existing algorithms in order to increase the quality of line drawings while dealing with the problem of over-sketching. We achieve it by introducing a new attribute, "importance", which is defined on all points on each line. We define importance as a value that increases around the prominent features of shapes

(a) (b) (c) (d)

Fig. 1 Example of previous algorithms. (a) The original Happy Buddha model. (b) Result of the "Suggestive Contours[4]" algorithm. (c) Result of the "Ridges and Valleys[5]" algorithm. (d) Combination of (b) and (c), emphasizing the overlapped lines. The lines generated by these two different algorithms almost overlap in some areas, shown as dark lines in the right image.

and decreseas otherwise. Furthermore, we assign contour lines (lines that separate objects from the background) the maximum importance so that we always draw them. We find that this definition of importance depicts shapes effectively. Our proposed method computes all lines and their importances, and then eliminates some of the lines in two ways. Firstly, the lines that have less importance are simply eliminated by a user-defined threshold value. Secondly, the lines that overlap or are close to each other are eliminated based on their importances and neighborhood line's importance. Our method is designed to run on GPU and our implementation achieves real-time performance. Because of this real-time performance, users can control the amount of lines and detail level interactively by changing the parameters.

2. Related Work

2.1 Line Drawing Algorithms

The simplest and well known algorithms are the Silhouettes and Creases. Silhouettes are set of points on a 3D model surface whose normal vectors are perpendicular to the view vectors of the camera. Creases are set of edges of a 3D model, which are marked as "sharp" edges.

In recent years, some more advanced algorithms based on curvature and the derivative of curvature of surfaces have been developed, such as Suggestive Contours by DeCarlo et al.[4], Ridges and Valleys (R&V) by Ohtake et al.[5], Apparent Ridges by Judd et al.[6], and Demarcating Curvers by Kolomenkin et al.[7]. These line drawing algorithms also define the methods to eliminate unnecessary lines. These methods have a common point, comparing the algorithm-defined strength with a user-defined threshold. However, each algorithm has its own definition of strength, which is not compatible with each other. Since each strength is very different, and these line eliminating algorithms are not designed for compositing multiple line drawing algorithms, a problem occurs. The problem is that many lines overlap with other lines, and some areas become too dense, containing too many lines. Grabli's line drawing algorithm[8] allows the user to define a priority per line for eliminating unnecessary lines. However, this priority is manually specified by the users. Moreover, it sorts the lines by priority, which is difficult to process on GPU. Since some line drawing algorithms are computed on GPU, this method is unsuitable for real-time applications.

2.2 Evaluation of Line Drawing Algorithms

To evaluate these line drawing algorithms introduced in Section 2.1, Cole et al.[1] compared the lines drawn by algorithms with the ones drawn by humans. Cole et al. tested three algorithms: Suggestive Contours, R&V and Apparent Ridges.

Generally, R&V obtain good results on artificial rigid objects while Apparent Ridges and Suggestive Contours are good on smooth objects. Suggestive Contours tends to generate less lines compared to R&V and Apparent Ridges.

Cole et al. showed that no algorithm can emulate human line drawings completely. For example, for a model named Flange, about 80 to 90 percent of human's lines are produced by Apparent Ridges and R&V, and about 50 to 60 percent by Suggestive Contours. However, to achieve this accuracy, these algorithms also generate 30 percent of wrong lines, not drawn by humans.

They also studied the combination of these three algorithms. Each algorithm can only generate less than 20 percent of human drawn non-contour lines, this is, not all the lines. However when they combined these algorithms, the percentage of accuracy doubled compared to a single algorithm.

3. Proposed Method

Our basic idea is to generate lines using multiple algorithms and omit the unnecessary lines. We define an importance value for all points on each line, and then compare the importance of the line with its nearby lines. We omit the line if higher importance lines exist in the neighborhood or the line importance is smaller than a user-defined threshold.

The algorithm consists of five steps: line generation, importance computation, line-width determination, importance rendering, and line elimination.

3.1 Line Generation

For the line generation, we use a combination of multiple algorithms, referred in Sections 1 and 2. For our experiments, we have implemented the Silhouette algorithm, Crease algorithm, Suggestive Contours[4] and R&V[5].

The Crease algorithm and R&V algorithm do not depend on the camera direction, but the Silhouette and Suggestive Contours algorithms do. Since these camera-dependent algorithms can be implemented using a GPU shader language that outputs the polylines, it is a waste of time to send back the data and eliminate the lines on the CPU. Thus we implemented the following steps using the GPU shader language for gaining performance.

3.2 Importance Computation

For the importance computation, we propose a modified version of Mesh Saliency[9] as importance. The original Mesh Saliency method computes a value called saliency over all the vertices of the input model at multiple scales. The saliency value of a vertex becomes larger when that vertex is salient in many scales. That is, that vertex is part of the important shape feature. If that vertex is only salient in a few scales, the saliency value becomes smaller, for example, a punched-metal pattern on a flat surface. This property is an advantage over the existing line eliminating methods[4][5], since those methods only refer to the local curvature value.

However, the saliency computed using the original Mesh Saliency method is not straightforwardly applicable to our purpose since it is based on the mean curvature of the vertex. The saliency value becomes high, for example, in a mountain or crater, but not for a mountain range or valley. Since we want the lines over the mountain range or valleys to be important, we use the difference of principal curvatures instead of the mean curvature in the original method.

The first step is the same as in the original method; we compute the surface curvatures. There are many good ways to compute the curvature on polygonal meshes. We use Meyer's method[3]for computing the curvature. Let $\kappa(v, d)$ denote the curvature towards a direction d on the tangent plane of a vertex v. Let d_1, d_2 denote the principal curvature directions at the vertex v: $d_1 = \mathrm{argmax}_d \kappa(v, d), d_2 = \mathrm{argmin}_d \kappa(v, d)$.

Next, we compute the Gaussian-weighted average of the curvature difference $G(v, \sigma)$. Let $N(v, \sigma)$ be the v neighborhood vertices within a σ radius sphere. For each vertex $x \in N(v, \sigma)$, we project (and normalize) d_1, d_2 to x's tangent plane. Let d_1^x, d_2^x be the projected vectors. The Gaussian-weighted average of the curvature difference $G(v, \sigma)$ is computed as:

$$K(x, d_1^x, d_2^x) = \kappa(x, d_1^x) - \kappa(x, d_2^x), \quad (1)$$

$$\mathcal{G}(x, v, \sigma) = \exp(||v - x||^2 / 2\sigma^2), \quad (2)$$

$$G(v, \sigma) = \frac{\sum_{x \in N(v, 2\sigma)} K(x, d_1^x, d_2^x) \mathcal{G}(x, v, \sigma)}{\sum_{x \in N(v, 2\sigma)} \mathcal{G}(x, v, \sigma)}. \quad (3)$$

The last step is exactly the same as in the original method: we compute $G(v, \sigma)$ in multiple scales (i.e. evaluate $G(v, \sigma)$ with changing σ), normalize and compose the results for multiple scales.

Based on experimental results, our method generates better importance values than other importance computations (e.g. the original Mesh Saliency or the difference between two principal curvatures, see Figure 6).

Since the computational cost of this importance computation is $O(n \log n)$ where n is the number of vertices, it takes too long to compute it in each frame. Thus, we compute the importance in the precomputation step. The importance is computed on each vertex of the polyline, and interpolated along the line segment. Since the line drawing algorithms in Section 3.1 generate lines on the surface of the polygonal 3D model, it is easy to compute the saliency of each vertex of the line by interpolating the saliency between the vertices of the 3D model. Additionally, we specially treat contour lines, which are the lines that separate the object from the background or the

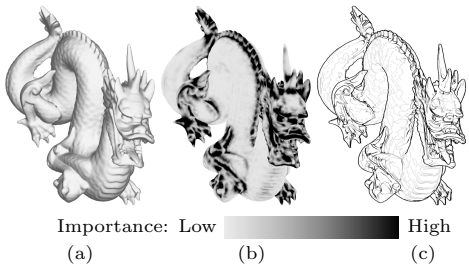


Fig. 2 An example of importance computation and mapping. (a) The original Dragon model. (b) The importance values on the surface of (a). (c) The importance values of the drawn lines. The importance is the same as in (b), except that the importance of the contour lines has been emphasized.

other objects (see Figure 2). We can easily extract the contour lines from the silhouette lines by using their depth values: pixels in contour lines are adjacent to pixels whose depth is not continuous (the background or the other objects). To avoid discontinuities in the contour lines, we scale their importance up to α times, experimentally two times, as follows: (depending on the depth difference between their two own borders)

$$r(p) = \frac{\alpha}{|2n(p)|} |D(p + n(p)) - D(p - n(p))|'$$
$$s(p) = \min(\alpha, \max(1, r(p))). \quad (4)$$

Where $s(p)$ is the importance scaling coefficient on pixel p on the line, and $n(p)$ is the orientation which is orthogonal to the line orientation, and $D(p)$ is the depth at pixel p. α is user-defined constant value.

3.3 Line Width Determination

For calculating the line width, we use the screen size, the density of lines and the importance of lines. We determine the line width according to the screen size, and we shrink the line width in dense areas to maintain the line visibility (see Figure 3).

To compute the density of the line, first we draw all lines to a density buffer as one pixel-width lines. We can use simple Bresenham algorithm[10] or the comparable algorithms since the lines are stored as polylines. To compute the density of lines, we initialize the density buffer to zero (black) and increment every time we draw a line on a pixel. Thus, this buffer stores the line density of any pixels. We scale it down using the linear mipmap filter of GPU as a fast approximation of a Gaussian filter so that the buffer stores the line density of nearby area.

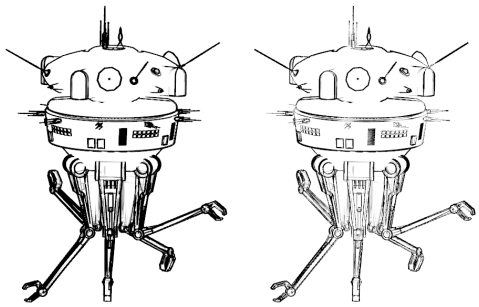


Fig. 3 An example of line width determination. The lines drawn in the left image are all the same width, and the right one is the result after applying the line width determination. The overlapping lines in dense areas become thinner (see the robot arms), but isolated lines still keep their width (see the two long antennas orienting obliquely upward).

We define the width of a line as:

$$\alpha s \cdot (\beta + d)^{-1}. \quad (5)$$

Where s is the longer one of the screen width or height in pixels, and d is the average value of the density buffer in nearby pixels (empirically, 8×8 or 16×16 nearby pixels give a good result). α and β are user-defined constants, which indicate the ratio of the line width to the screen size and the degree of effect from the line density to the line width. We define these constants as $\alpha = 0.002$ and $\beta = 0.8$.

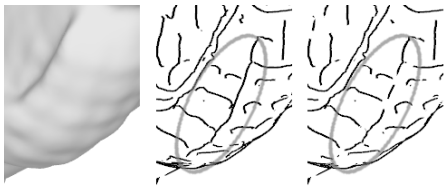
Additionally, when we eliminate the lines in Section 3.5, we do it smoothly: we linearly change the width from 100% (explicitly not eliminated) to 0% (explicitly eliminated).

3.4 Importance Rendering

For eliminating lines, we compare the importance of each pixel on the line with the nearby lines. Since we do not draw a pixel when a higher importance lines exist close to the pixel, we only need to compare the pixel importance with the maximum importance line around the pixel.

However, it is difficult to directly implement this comparison effectively using the GPU shader language. For this reason, we use a workaround that is adapted to GPU shader language, described in this and next sections.

In this importance rendering step, we compute the maximum importance around each pixel. First we



(a) (b) (c)

Fig. 4 Example of the left arm of the armadillo. (a) The original model. (b) Line drawings using directional importance buffers. (c) Line drawings without using directional importance buffers. Without using the directional importance buffers, the crossing lines are unnaturally eliminated.

draw all lines to an importance buffer, which has the same size as the screen. When drawing the line to the buffer, we draw the importance instead of the color of line, and we draw them wider (empirically, three to seven times wider than the width calculated in Section 3.3) to spread the importance into a large area, with higher importance lines overpainting lower importance lines. When drawing the importance to the importance buffer pixel, we compare the current pixel value and the line importance value and store the larger one to the pixel.

However, the above method does not consider the orientation of the lines, which means for example almost orthogonally crossed lines are treated as similar lines. This mis-treating leads to unnatural line elimination in the next step (see Figure 4). We want to treat nearby lines with similar orientations as similar lines. To realize this, we use "directional importance buffers", multiple buffers that are associated with several line orientations. We use four buffers, each associated to the orientations 0°, 45°, 90°, and 135° (see Figure 5).

When writing to each importance buffer, we multiply the importance by cosine of the angle between the line orientation and the buffer's associated orientation:

$$I_l \cos(\theta_l - \theta_{buf}). \tag{6}$$

I_l is the importance of the line (precisely, the importance of the pixel on the line). θ_{buf} is the orientation (in degree or radian) which is associated to the importance buffer, and θ_l is the orientation of the line.

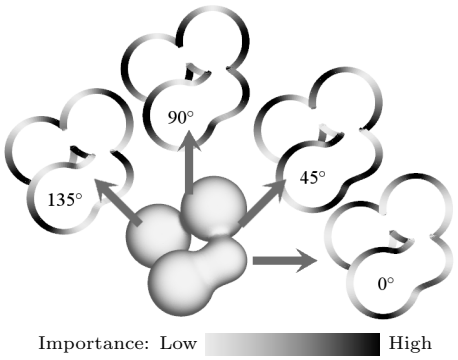


Fig. 5 An example of importance rendering to each directional importance buffer. The intensity shows the value of importance (obtained by Equation (6)).

3.5 Line Elimination

Finally, when we render a line to the screen, we eliminate the lines in two ways. The first way is to eliminate the lines by importance thresholding. By changing this threshold value, we can control the detail level of the line drawing.

The second way is to eliminate the lines by comparing the importance of the line against the value in the importance buffer, considering the orientation of the line. If the importance of the line is less than the importance read from the importance buffer, it means that there are more important lines in the neighborhood, thus we do not draw the line to the screen.

As described in Section 3.4, we consider the line orientation when writing to the importance buffer. We also consider the line orientation when reading from the buffer. When reading from the importance buffers, we choose two importance buffers which are associated to the two nearest orientations to the line orientation, and linearly interpolate the pixel values from these two buffers as:

$$\frac{\left|\theta_{buf2} - \theta_l\right| I_{buf1} + \left|\theta_{buf1} - \theta_l\right| I_{buf2}}{\left|\theta_{buf2} - \theta_{buf1}\right|}. \tag{7}$$

I_{buf1}, I_{buf2} are the pixel values of the directional importance buffers to be interpolated. $\theta_{buf1}, \theta_{buf2}$ are the orientations which are associated to the importance buffers. Note that all the theta values should be in the range $[0°, 180°)$.

4. Results

We implemented and executed our program on a desktop PC (Intel Core i7 920 2.67 GHz, 6 GB RAM,

GeForce 9800 GT 512 MB VRAM). Our implementation includes the following line drawing algorithms: Silhouettes, Creases, Suggestive Contours and R&V.

We implemented Creases and R&V using C++, and Silhouettes and Suggestive Contours using the Direct3D10 shader language. Since Silhouettes and Suggestive Contours depend not only on the shape of the model but also the camera direction, we had optimized the implementation of these algorithms to achieve real-time performance. We implemented all the steps in GPU except the precomputation (computing the curvature and the importance).

Figure 7 shows the results before and after applying our proposed method. We can see that our result images have less line density than the original images. The lines in (d) are too dense especially in area 2 and they appear like fractures on a flat surface, resulting in losing the shape feature (i.e. the flatness). On the contrary, the lines in our generated images are simplified in those areas. We drew lines in pronounced valleys and mountains (high importance) and we deleted them in shallow valleys and low mountains (low importance).

Figure 8 shows the line elimination step by step. Our method eliminates lines in relatively flat surfaces in the importance thresholding step (c), and in the next step (d) it eliminates the overlapping lines. The screwdriver image in Figure 9 shows that our method eliminates lines properly in any scale. Figure 10 shows the result of changing the line elimination threshold value for controlling the detail level.

Our method is fast enough to be used in real-time applications as shown in Table 1. Our proposed method achieved 30 fps even for a dragon model with 99 k vertices. Since the rendering are performed in real-time, we can control the results interactively by changing some of the rendering parameters (i.e. the line elimination threshold, the line width etc. See Equations (4),(5)).

Table 1 The timings of our method for some models, in millisecond per frame.

Model	Screwdriver	Dragon
Number of vertices	27 k	99 k
Without our method (ms)	8.3	28.6
With our method (ms)	10.7	33.3
Difference (ms)	2.4	4.7

The followings are the limitations of our method. Our importance computation algorithm gives high importance to shape features (e.g. the claws at the arm and the leg of the dragon) and tends to lower the importance of repeating patterns (e.g. the stripe pattern at the chest of the dragon). We cannot change this line elimination priority unless using another importance definition. Moreover, our method for line elimination decides to erase a line or not on each point on the line, resulting in some lines being splitted into several small segments.

5. Conclusions and Future Work

In this paper, we have proposed a method to generate line drawings of 3D models by combining the previous methods while avoiding over-sketching based on the importance of lines.

Compared to the previous work, our method is better at generating more lines to depict the 3D models while eliminating unnecessary lines, especially in areas where many of them would overlap. The results are clearer and crisper line drawings. We also focused in making our algorithm work in real-time so that it can be used in games and interactive applications, like medical imaging and CAD programs. Users can control the amount of lines and detail level interactively by changing the parameters.

Our method and existing algorithms generally require the curvature on the surface. A real-time method for computing the curvature in the screen space is proposed by Kim et al.[11]. However, since we need the curvature in the 3D model space, we do not use this algorithm and compute the curvature in the precomputation step. As future work, we would like to develop a real-time algorithm for rendering deformable objects, that is the curvature is changing in each frame.

References

1) F. Cole, A. Golovinskiy, A. Limpaecher, H. S. Barros, A. Finkelstein, T. Funkhouser, and S. Rusinkiewicz, "Where do people draw lines?," in *SIGGRAPH '08: ACM SIGGRAPH 2008*, (New York, NY, USA), pp. 1–11, ACM, 2008.
2) T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte, "A developer's guide to silhouette algorithms for polygonal models," *IEEE Comput. Graph. Appl.*, vol. 23, no. 4, pp. 28–37, 2003.
3) M. Meyer, M. Desbrun, P. Schroder, and A. H. Barr,

"Discrete differential-geometry operators for triangulated 2-manifolds," *Visualization and Mathematics III*, pp. 35–57, 2003.

4) D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella, "Suggestive contours for conveying shape," in *SIGGRAPH '03: ACM SIGGRAPH 2003*, (New York, NY, USA), pp. 848–855, ACM, 2003.
5) Y. Ohtake, A. Belyaev, and H.-P. Seidel, "Ridge-valley lines on meshes via implicit surface fitting," in *SIGGRAPH '04: ACM SIGGRAPH 2004*, (New York, NY, USA), pp. 609–612, ACM, 2004.
6) T. Judd, F. Durand, and E. Adelson, "Apparent ridges for line drawing," in *SIGGRAPH '07: ACM SIGGRAPH 2007*, (New York, NY, USA), p. 19, ACM, 2007.
7) M. Kolomenkin, I. Shimshoni, and A. Tal, "Demarcating curves for shape illustration," in *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008*, (New York, NY, USA), pp. 1–9, ACM, 2008.
8) S. Grabli, E. Turquin, F. Durand, and F. Sillion, "Programmable style for npr line drawing," in *Rendering Techniques 2004 (Eurographics Symposium on Rendering)*, ACM Press, june 2004.
9) C. H. Lee, A. Varshney, and D. W. Jacobs, "Mesh saliency," in *SIGGRAPH '05: ACM SIGGRAPH 2005*, (New York, NY, USA), pp. 659–666, ACM, 2005.
10) J. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems journal*, vol. 4, no. 1, pp. 25–30, 1965.
11) Y. Kim, J. Yu, X. Yu, and S. Lee, "Line-art illustration of dynamic and specular surfaces," in *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, (New York, NY, USA), pp. 1–10, ACM, 2008.

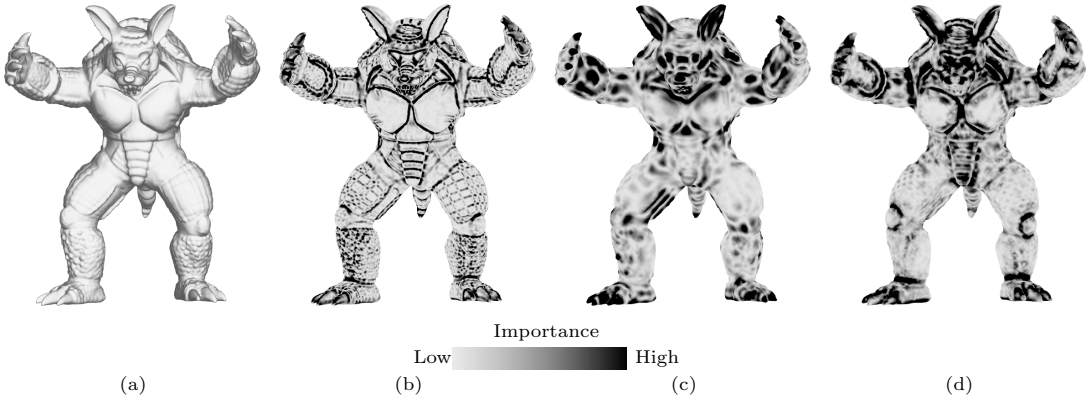


Fig. 6 Examples of importance computations. (a) The original 3D model. (b) The difference between two principal curvatures. (c) Mesh saliency. (d) Our proposed method. Our proposed method produces high importance at large valley or large mountain areas (the base of the feet or arms, the breast muscle) but not at small features (the rough pattern of the legs).

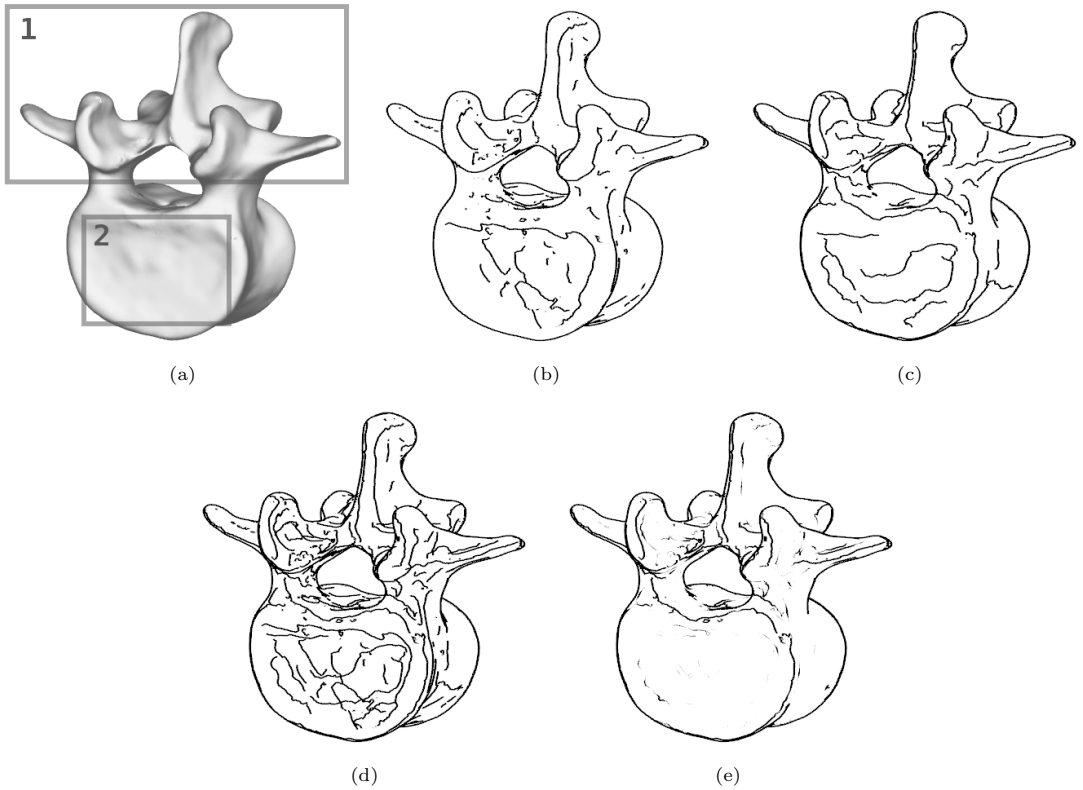


Fig. 7 A comparison between the existing methods and our method. (a) The original 3D model. The two boxes (area 1 and area 2) are for later references. (b) Suggestive Contours. (c) Ridges and Valleys. (d) Overlapping (b) and (c). (e) Our proposed method. Compared to (b), (c) and (d), our proposed method eliminates unnecessary lines in relatively flat surfaces (see area 2). Also, our method eliminates some overlapping lines compared to (d).

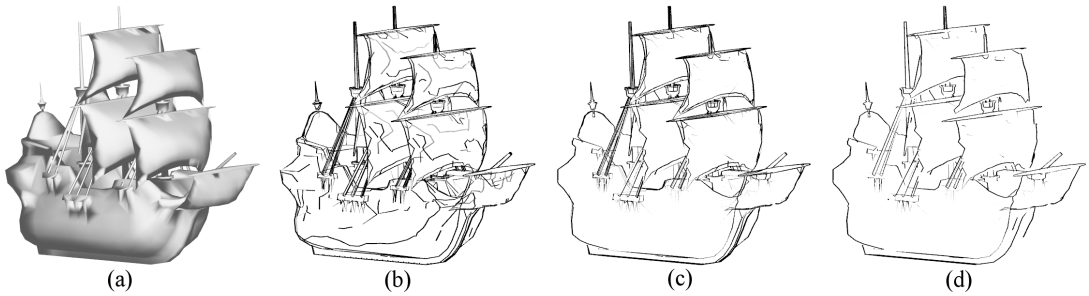


Fig. 8 An example of importance-based line elimination. (a) The original 3D model. (b) A drawing with overlapping lines. No line elimination applied. (c) Line elimination by importance thresholding applied on (b). (d) Overlapping line elimination applied on (c). We can see that lines on smooth surface (see sail and hull) are eliminated in (c), and the parallel overlapped lines (see pole and rope) are eliminated in (d).

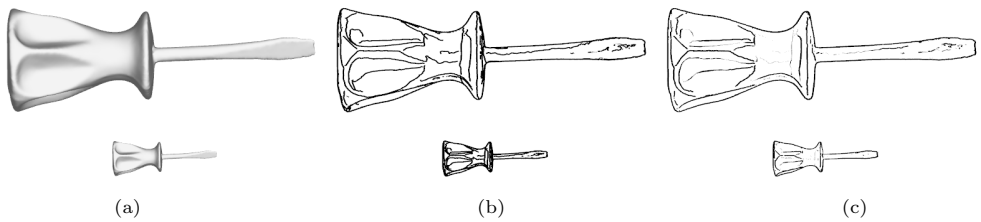


Fig. 9 An example of changing the scale of the object. (a) The original 3D model. (b) Results of Suggestive Contours and R&V overlapped. (c) The output of our method. Our method can automatically adjust the number of lines when the scale is changed.

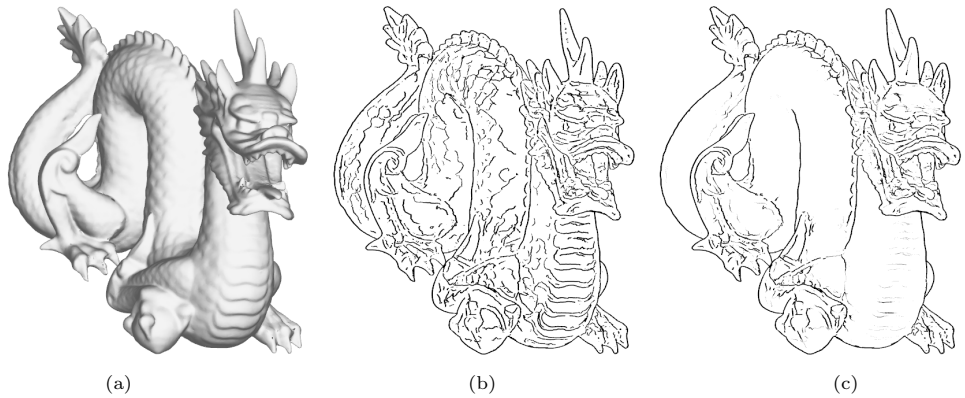


Fig. 10 An example of changing the importance threshold value. (a) The original 3D model. (b) The output of our method with a low importance threshold value. (c) Another output of our method with a high importance threshold value. User can control the detail level of the lines interactively by controlling the threshold value.