EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling (2005)
Takeo Igarashi, Joaquim A. Jorge (Editors)

A Simple-to-Implement Method for Cutting a Mesh Model by a Hand-Drawn Stroke

J. Mitani

Department of Computer Science, University of Tsukuba

Abstract

In the field of Computational Geometry, the design of 3D models using hand drawn strokes has been well-studied in recent years as a way to improve user interfaces. When using hand-drawn strokes to design 3D models, an algorithm for cutting a model by a stroke is required. Previous algorithms have concentrated on precision and are expensive to implement. This paper gives priority to simplicity and robustness rather than precision. Firstly, mesh vertices near to the stroke are moved so that they lie on the stroke, to avoid numerical error. Then the stroke is simplified so that it crosses a triangle at most two times. With this approach, the number of patterns of triangle division that a system has to implement is reduced to only three. This reduces the time a developer must take to implement a cut operation for a sketching interface.

Categories and Subject Descriptors (according to ACM CCS): J.6: Computer Aided Engineering [Computer Aided Design]

1. Introduction

In the field of Computational Geometry, the design of 3D models using hand drawn strokes has been well-studied in recent years as a way to improve user interfaces. The investigations can be categorized into two groups: one is analyzing strokes as a gesture [ZHH96] and replacing the strokes with a command for generating primitives; the other is to apply user strokes directly to a 3D model. The latter approach is more intuitive and such interfaces have commonly been used in recent studies [IMT99, SFMT04]. In such interfaces, an algorithm for cutting a model by a stroke is required. Cutting a model by a stroke corresponds to generating a swept surface perpendicular to the screen and cutting the model by this surface (Fig.1 (a) (b)). Generally, the stroke (the track of the mouse cursor) is obtained as an array of discrete points, so the swept surface is expressed as a polygon. When the original model is represented by a triangulated mesh model, we can obtain the resulting cut model by executing a boolean operation for polygons. Alternatively, the same effect can also be achieved by cutting triangles by a stroke on a projected screen (Fig.1 (c)).

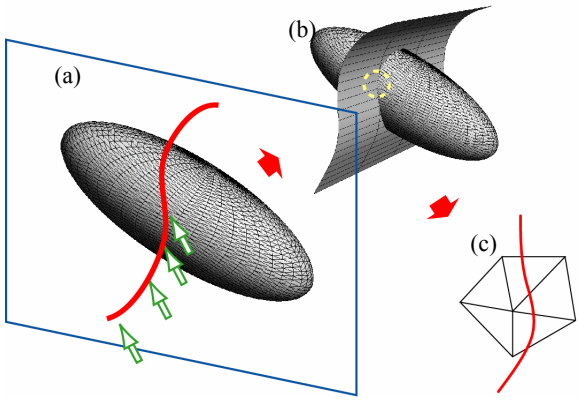


Figure 1: *Cutting operation using a hand drawn stroke.*

Although dividing triangles by a polyline is not difficult geometrically, implementing this operation is problematic since we cannot avoid numerical errors caused by the representation of real numbers in a computer. It is easy for topological contradiction to occur as a result of these numerical errors. To avoid this, some investigators used a topology priority method [Sug00]. However, implementing this robustly is a time-consuming (and expensive) cod-

ing task, and if our objective is to improve a user interface, we do not want so much trouble over such a comparatively minor problem.

In this paper, I propose a new, simple-to-implement method for cutting a mesh model using a stroke. This paper places more value on simplicity and robustness than precision. Both literally and metaphorically, it cuts corners, avoiding inessential detail to achieve a simple cutting operation for a sketching interface.

2. Target application and problems in usual implementation

2.1 Target application

This paper is aimed at applications in which a mesh model is cut by user strokes and separated into segments. As shown in Fig.2, the mesh model is cut by lines that the user draws on the model as projected onto a screen. This particular application cuts only the front, visible part of the model. Although other applications may require a cutting operation that separates a model completely, i.e. also cuts the back at the same time, there are no essential differences.

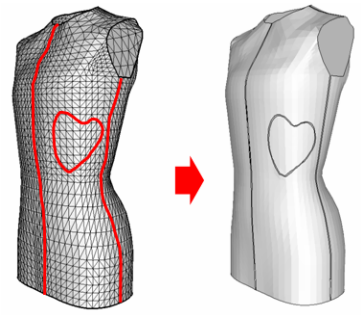


Figure 2: *Our target: a cutting operation for triangulated models.*

2.2 Problems of usual approach

Cutting a triangulated mesh model by a stroke on a projected screen is achieved by cutting the individual triangles of the model by the stroke. For the results to correspond exactly to each point on the cut-line, each triangle should be divided by the exact stroke line, and the separated parts triangulated as shown in Fig.3.

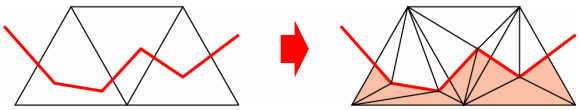


Figure 3: *Dividing triangles by a stroke.*

With this approach, the points of intersection of the stroke and the triangle edges are found first. But when a stroke passes very close to a vertex, numerical error can make it impossible to reconstruct a consistent topology. Even in the absence of numerical error, triangulating polygons after dividing triangles by a stroke is cumbersome to implement when, for example, the triangulation is done by the CDT (Constraint Delaunay Triangulation) algorithm. Another problem is that this operation potentially generates tiny and thin triangles. If we wish to continue editing the model, it is desirable that the triangles are uniformly-sized and not thin. So there are three problems with using strokes directly to generate cut lines.

- Numerical error can cause topological contradiction.
- Implementing triangulation algorithms such as CDT requires disproportionate effort.
- Tiny and thin triangles are frequently generated.

2.3 Related work

Nealen et al. [NSAC05] moved the vertices that lie near a stroke so that they lie on the stroke (I also use this approach for the first part of the operation). After this they moved other vertices using a relaxation operation to improve triangle shapes. This is a reasonable approach but it also requires time-consuming implementation. Turquin et al. [TCM04] proposed a method for designing garments using strokes. In their method they moved vertices to strokes, but the vertices have to be arranged initially as a grid. Krishnamurthy et al. [KL96] proposed a method to segment mesh models using user strokes. This method was developed for generating B-spline surfaces, and it is not applicable for mesh segmentation.

Some investigators propose algorithms that give priority to topological validity over geometric precision in order to avoid topological contradictions caused by numerical errors [SI89]. In [IH03], after their system executes operations in response to user strokes, the mesh model is reconstructed by removing tiny and thin triangles to improve the quality of the mesh. These approaches are effective in solving the problems, but implementing them is far harder even than implementing CDT (personally, I do not even want to implement CDT!).

The simplest approach of all is not to divide any triangles, but use edges in the mesh model as segments of the cut line. However, with this approach, the generated cut line depends entirely on the structure of the original mesh.

The method proposed in this paper divides a triangle into no more than four pieces. Hence it can generate better

results than the method without any triangle division while being simpler to implement than the usual approaches. The proposed method is a good compromise between the two extremes.

2.4 Targeting operation

Strokes are generated by motion of the human hand, and the reproducibility is poor, so sketching interfaces are not used in situations where accurate results are required. In an application such as Teddy [IMT99] it is enough that the rough outline of a model is defined by a stroke. We do not have to apply the exact coordinates of the stroke to the model. Furthermore, the triangulated mesh representation is itself an approximation of curved surfaces, so we do not require a level of detail that cannot be represented by triangles of the size of those in the original model. Thus, the targeting operation characteristics are as follows:

- does not require that the cut matches the stroke exactly.
- does not require details too small to be represented by the triangles of the original model.

To make problem simpler, this paper assumes:

- strokes are not self-intersecting.

These assumptions are reasonable and commonly acceptable. This paper proposes a method that achieves a mesh cut operation that requires implementation of only 3 patterns of triangle division. The implementation is straightforward and does not require checks for numerical error. The generated triangles are neither tiny nor thin.

3. Methodology

3.1 Terminology

The following terms are used in this paper.

- **stroke-vertex**: A vertex of the stroke.
- **stroke-segment**: A line segment that has stroke-vertices at both ends.
- **model-vertex**: A vertex of the mesh model.
- **model-edge**: An edge of a triangle in the mesh model.
- E_i: The ith model-edge. Each model-edge has a unique ID.
- V_j: The jth model-vertex. Each model-vertex has a unique ID.

3.2 Model-vertices move

Before the main processing stage, model-vertices that lie close to a stroke on the projected screen are moved so that they lie exactly on the stroke.

The idea of moving vertices of models so that they lie on user strokes has also been proposed by Biermann et al. [Bie01] for boolean operations on subdivided surfaces. By this means, we can avoid the instability caused by numerical errors that is often generated when a stroke-vertex and a model-vertex are close to one another. At the same time, we can avoid generating thin triangles.

For the method proposed in this paper, this operation is performed as follows.

Move to a stroke-vertex

Search for the nearest stroke-vertex in screen coordinates to each model-vertex, and if the distance between the two is smaller than a threshold, move the model-vertex so that it lies exactly over the stroke-vertex in a plane that is parallel to the screen. Then set a flag for the stroke-vertex to indicate that it lies on a model-vertex.

Move to a stroke-segment

Search for the nearest stroke-segment in screen coordinates to each model-vertex except those moved by the previous operation, and if the distance between the two is smaller than a threshold, move the model-vertex to the nearest position on the stroke-segment. Then add a new stroke-vertex at this position and set a flag to indicate that the stroke-vertex lies on a model-vertex (Fig.4).

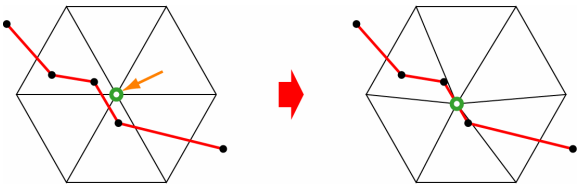


Figure 4: *Moving a vertex of the model close to a stroke.*

3.3 Simplification of a stroke

For simple implementation, the stroke is simplified to reduce the number of patterns of triangle division. The following two steps of simplification are applied.

First simplification

When a stroke crosses triangles as shown in Fig.5(a), generate new stroke-vertices at intersecting points (b), then remove stroke-vertices except those at the intersecting points and those for which the flag was set at the previous model-vertices move operation. When a stroke-vertex lies very close to a model-edge, determining whether a stroke-segment and the model-edge intersect becomes unstable, sometimes causing the strange result that a stroke crosses a triangle without any intersecting points! So if the distance

between a stroke-vertex and the nearest model-edge is smaller than a threshold, it is assumed that they intersect. Although this can result in multiple crossing points existing on a single edge, the second simplification which follows will remove such excess crossing points.

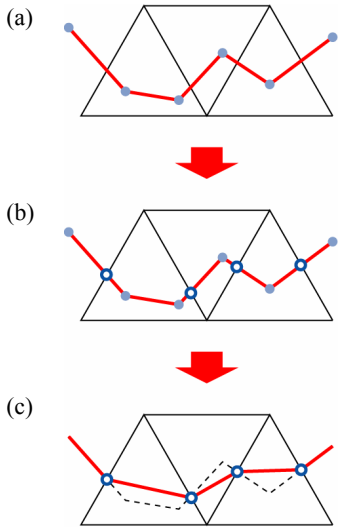


Figure 5: *Cutting by a stroke that does not contain on-face-vertices.*

With this simplification, the detail of strokes inside triangles is ignored, so we no longer represent fine details of the stroke. This simplification makes it easy to implement triangulation of the areas generated by dividing the triangle, because we do not have to consider complicated triangulation. After this simplification, each stroke-vertex must lie on a model-edge or model-vertex. Hence every stroke-vertex can be identified from the model element E_i or V_j that the stroke-vertex lies on. From now on, the elements that stroke-vertices lie on (E_i or V_j) are stored in an array $\boldsymbol{A}$; ex. $\boldsymbol{A} = (E_8, E_7, E_6, V_3, E_6, E_5)$.

Second simplification

Even after the first simplification, when a stroke crosses a single triangle multiple times, the number of possible patterns of triangulation is infinite, so implementation is still difficult.

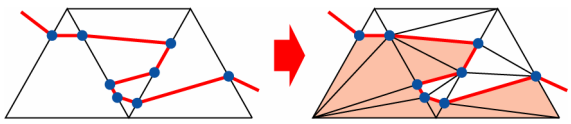


Figure 6: *Cutting by a stroke that does not contain on-face-vertices.*

After the first simplification, a second simplification is applied such that each triangle is not divided into more than two. When a stroke crosses a triangle only once, the number of cases we must consider is only two. One is intersection at a model-vertex and a model-edge (Fig.7 (a)), the other is intersection at two model-edges (Fig.7 (b)).

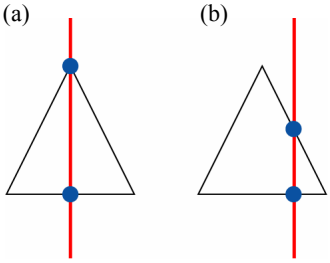


Figure 7: *A stroke intersecting a triangle at two points.*

In both cases, there are exactly two points of intersection, and corresponding stroke-vertex elements (E_i or V_j) are neighbours in array $\boldsymbol{A}$. If there are more than two elements that belong to the same triangle in $\boldsymbol{A}$, these elements have to be removed from $\boldsymbol{A}$, leaving only two. The second simplification checks all triangles and searches for elements of A that belong to that triangle. The one that appears first in A and the one that appears last in A are labelled (Fig.8(a)). During this process, if more than two elements are found, the second simplification removes all those elements that are between the first element and the last (Fig.8(b)).

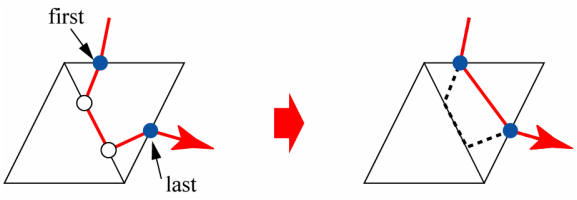


Figure 8: *A stroke crosses a triangle multiple times.*

```
foreach(Triangles t) {
    for(int i = 0; i < A.size; i++) {
        if(A[i].belongs(t)) break;
    }
    for(int i = A.size - 1; j >= 0; j--) {
        if(A[i].belongs(t)) break;
         }
    }
    if(j - i > 1) {
        A.remove(i + 1, j - 1);
    }
}
```

Figure 9: *Algorithm for the second simplification.*

This algorithm is shown in Fig.9 as pseudo-code. In the code, `A[i]` stands the (*i*-1)th elements in ***A***, `A.size` is the number of elements in ***A***. The method `A[i].belongs(t)` is true if `A[i]` belongs to triangle `t`, and `A.remove(a, b)` removes elements `A[a]` to `A[b]` from ***A***. The result of this algorithm depends on the order in which it is applied to triangles.

Fig.10 shows an example of the simplifications as applied to a stroke and the triangles shown in (a). Although this is an exaggerated example, in which a stroke crosses very close to vertices and edges, topologically complicated situations like this can easily occur. By applying the first simplification, the stroke-vertices that lie in a triangle are removed and the result becomes as shown in (b). From (c) to (h), the second simplification is applied step by step. The large black points in Fig.10 are stroke-vertices that the algorithm decides to retain in ***A***. The white points are to be removed. A grey triangle is the triangle under consideration at that step. Overall, the input stroke in (a) is simplified to that in (i).

3.4 End points and angle points

As shown in the previous example, the two simplifications produce a stroke which crosses each triangle at most once, so the division of a triangle can be represented by only two patterns as shown in Fig.7. Although this is very effective in simplifying implementation, angle points disappear. When a user inputs an angle point such as Fig.11, retaining the sharp angle may be important.

To keep angle points of a stroke, other cases are added to the set of allowed patterns. These contain the end point of a stroke in a triangle. For these cases, there are two patterns of intersection between a stroke and a triangle, as shown in Fig.12. One is intersection at a model-vertex (a), and the other is intersection at a model-edge (b).

When a stroke-vertex corresponds to an angle point, the stroke is divided at this point into two strokes, and a new end point is generated (Fig.13). An angle point can be defined as one where the angle between neighbouring edges is smaller than a threshold.

Thus the total number of patterns which can result from a stroke intersecting a triangle is four, as shown in Fig.14(a)~(d). The corresponding triangle division is shown below. Since case (d) (an end point and crossing at a model-edge) can be achieved by performing (c) and then (a), the number of patterns that we have to implement is only three; (a), (b) and (c). With only these patterns, we can achieve all of the triangle division that is required for a cutting operation.

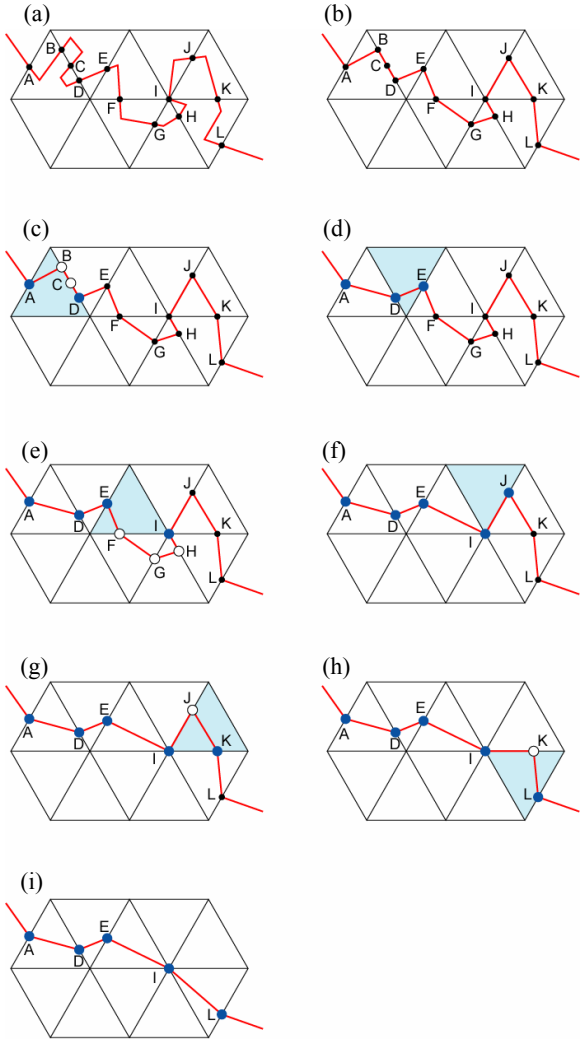


Figure 10: *An example of the second simplification.*

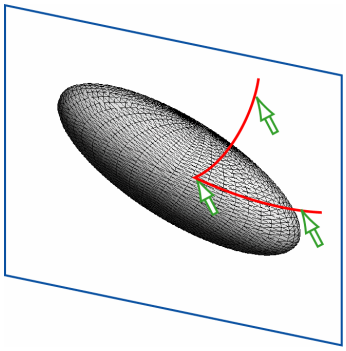


Figure 11: *A stroke that contains a sharp angle.*

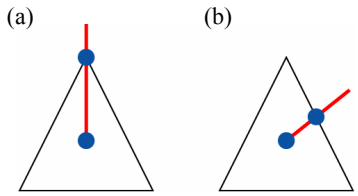


Figure 12: *A stroke that has an end point in a triangle.*

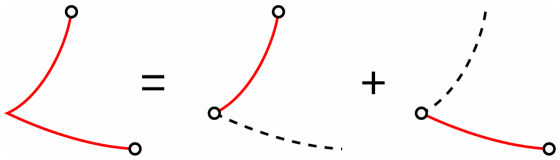


Figure 13: *A stroke that contains a sharp angle.*

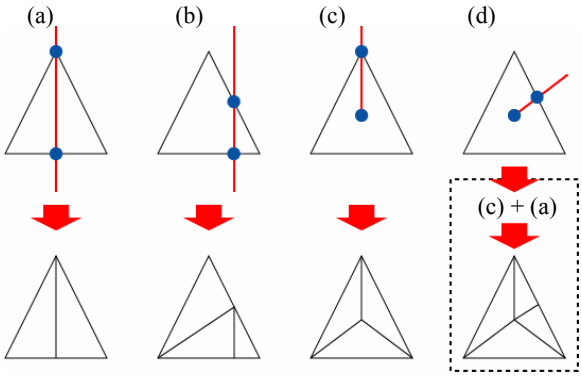


Figure 14: *Patterns of triangle division.*

3.5 Summary of the method

1. Move model-vertices that lie close to a stroke so that they lie exactly on the stroke.
2. Divide the stroke into multiple strokes at each sharp angle.
3. Apply the following operations to each division of the stroke
   a. Divide any triangle that contains an end point of a stroke into three triangles as Fig.14(c).
   b. Apply the first simplification.
   c. Apply the second simplification
   d. Divide triangles that are crossed by a stroke as Fig.14(a)/(b) according to the pattern.

4. Results

A prototype system was implemented in C++ on a Windows PC (2.0GHz CPU, 1.0GB RAM). The results of cutting a triangle mesh model by a user stroke are shown in Fig.15 and Fig.16. In Fig.15, (a) shows the detail of the original stroke, (b) shows the result of the model-vertex move, (c) shows the result of the second simplification, the points are the elements in array ***A***, and (d) is the result of mesh division.

Fig.16 shows the result for a stroke that has an angle point and two endpoints.

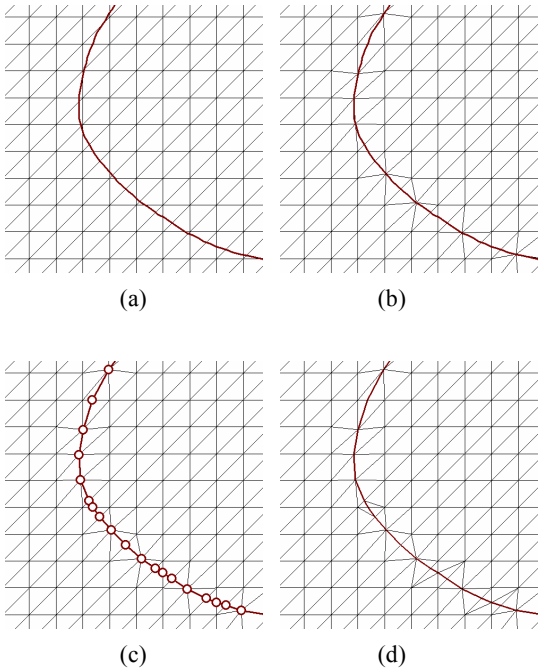
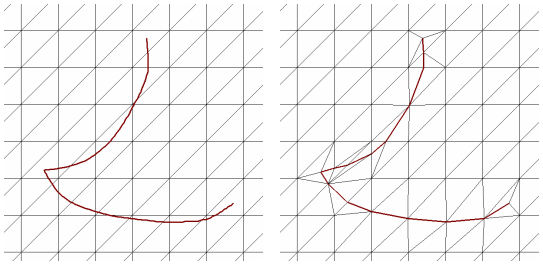


Figure 15: *Result obtained with our implementation.*



(a) (b)

Figure 16: *Result for a stroke that contains a sharp angle.*

5. Conclusion

This paper proposes a method for cutting a mesh model by a stroke that can be easily implemented. By applying two stages of simplification, the number of patterns of triangle division that must be implemented is reduced to only three. It can be implemented easily without any need to consider numerical error. The results are not as accurate as those produced by stricter approaches, but are much better than those produced by the simplest approach of not dividing any triangles and using edges in the mesh model as segments of the cut line.

6. Future Work

The method proposed in this paper is for cutting the front of a model only. Cutting both front and back to separate a model, using the same basic idea to cut both the front and the back, will require some study of methods for generating cross-sectional surfaces.

To preserve the detail of stroke lines, adaptive subdivision as in [KS99] could be added as a preprocessing operation.

References

[IH03] IGARASHI T., HUGHES J. F.: Smooth Meshes for Sketch-based Freeform Modeling. ACM Symposium on Interactive 3D Graphics (2003), 139-142

[IMT99] IGARASHI T., MATSUOKA S., TANAKA H.: Teddy: A Sketching Interface for 3D Freeform Design. In Proc. SIGGRAPH '99 (1999), 409-416.

[KL96] KRISHNAMURTHY V., LEVOY M.: Fitting smooth surfaces to dense polygon meshes. In Proc. SIGGRAPH '96 (1996), 313-324.

[KS99] KHODAKOVSKY A., SCHRÖDER P.: Fine level feature editing for subdivision surfaces. In ACM Solid Modeling Symposium (1999), pp. 203–211.

[NSAC05] NEALEN A., SORKINE O., ALEXA M., COHEN-OR D.: A Sketch-Based Interface for Detail-Preserving Mesh Editing. In Proc. SIGGRAPH '05 (2005), to appear.

[SFMT04] OWADA S., NIELSEN F, OKABE M., IGARASHI T.: Volumetric Illustration: Designing 3D Models with Internal Textures. ACM Transactions on Graphics, Vol.23, No.3 (2004), 322-328.

[SI89] SUGIHARA K., IRI M.: A solid modelling system free from topological inconsistency. Journal of Information Processing, Vol.12, No.4 (1989), 380-393.

[Sug00] SUGIHARA K.: How to Make Geometric Algorithms Robust. IEICE Transactions on Information and Systems, Vol.E83-D, No.3 (2000), 447-454.

[TCM04] TURQUIN E., CANI M. HUGHES J. F.: Sketching garments for virtual characters. In Proc. EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling (2004).

[ZHH96] ZELEZNIK R.C., HERNDON K.P., HUGHES J. F.: SKETCH: An interface for sketching 3D scenes. In Proc. SIGGRAPH '96 (1996), 163-170.