MANCHESTER
1824
The University of Manchester

The University of Manchester Research

SEED

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Bao, X., & Howard, T. L. J. (2014). SEED: Sketch-Based 3D Model Deformation. In *The Proceedings of NICOGRAPH International 2014* (pp. 53-64). The Society for Art and Science.

Published in:
The Proceedings of NICOGRAPH International 2014

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date:09. Jun. 2022

SEED: Sketch-Based 3D Model Deformation

Xin Bao **Toby L. J. Howard**

School of Computer Science, The University of Manchester

{baox, toby} (at) cs.man.ac.uk

Abstract
Traditional techniques for deforming virtual 3D models require the user to explicitly define a set of control points, a region of interest (ROI), and to define precisely how to deform the ROI using the control points. In this paper we present SEED, a sketch-based 3D model deformation approach, which allows the user to perform a deformation using only a single control stroke, eliminating the need to explicitly select control points, the ROI and the deforming operation.

1 Introduction

In 3D modelling software, deformations are usually used to add, to remove, or to modify geometric features of existing models to create new models with similar but slightly different details. To perform a deformation, the user is usually required to explicitly define *where* – the region of interest (ROI) – and *how* – the control points and the deforming operation – the deformation will take place. However, the explicit selection of these elements makes the deformation process somewhat unintuitive, especially for people with little experience of 3D modelling. As applications requiring virtual 3D model processing become more and more widespread, it becomes increasingly desirable to lower the "difficulty of use" threshold for users.

When people sketch with pen and paper, observation shows that they naturally "over-sketch": they redraw a slightly different curve from their original one, near the part of the drawing where they wish to make a modification [1]. Inspired by the idea of over-sketching, we propose a sketch-based deformation approach, SEED (Silhouette-Expanding Easy Deformation), which utilises the strokes that users sketch to guide the deformation of the silhouette of 3D models. In contrast to existing approaches which require the user to select the control points and the ROI manually, SEED estimates this information from a single stroke made by the user.

This paper describes the structure and operation of the SEED system, and covers the following main features: (1) the system uses a sketch-based 3D model deformation approach that performs a deformation to an existing 3D model with only one stroke that the user sketches. The selection of control points, ROI and deforming operation are done implicitly; (2) the system smooths geometric silhouettes so that they can be used as control points in the deformation directly, and eliminates unnecessary details in the geometric silhouettes which will otherwise affect the deformation; (3) we automatically detect the ROI, using the matched silhouette segments as the seeds of ROI selection in a foreground growing algorithm involved in the mesh segmentation; (4) we perform two-stage ROI remeshing, which firstly subdivides the ROIs before deformations, to ensure enough vertices participate in deformations, and then restores the original vertex density after deformation, to maintain a consistent look between the ROI and the rest of the mesh; and (5) we exploit the GPU to optimise our algorithms, which makes SEED responsive enough for real-time interaction.

1.1 Related Work

Gesture-driven user interfaces, rooted in natural human action, have gained significant attention in recent years. Since the appearance of Igarashi et al's "Teddy" in 1999 [2], many approaches have been proposed to utilise sketching operations in the creation and the modification of virtual 3D models. Our approach falls in this category, and in particular we focus on modifying existing 3D models through intu-

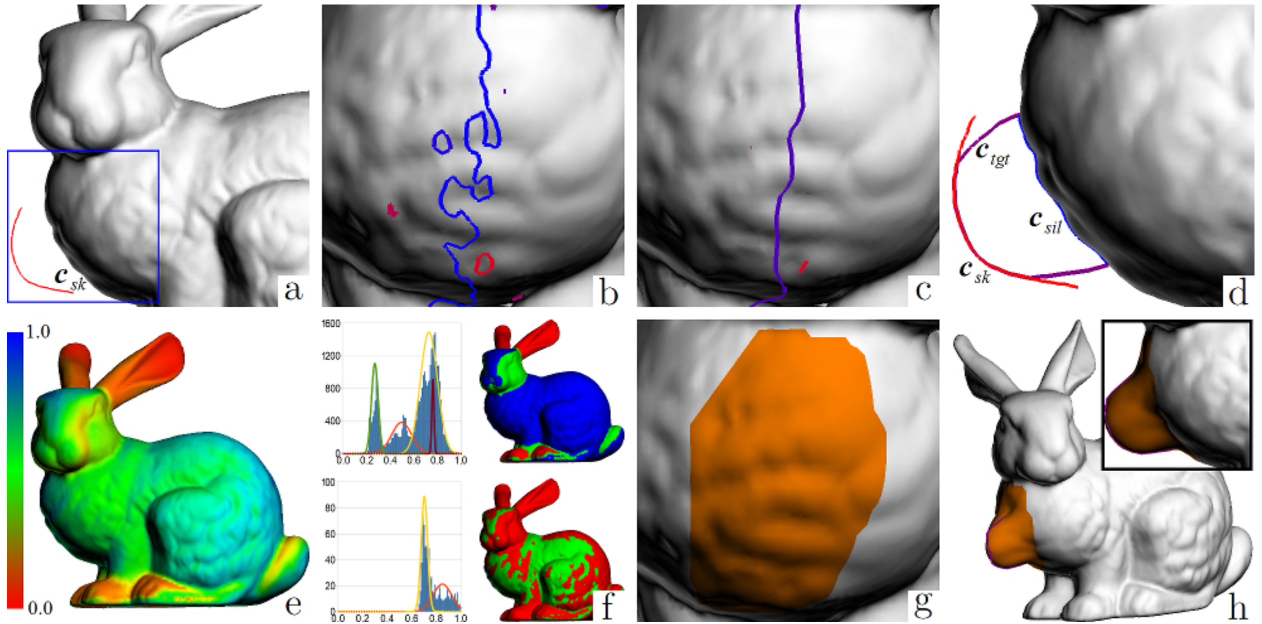

Figure 1: The working procedure of SEED as described in Section 2. (b), (c) and (g) are the block in (a) observed after rotating the model $\pi/2$ around the y-axis.

itive and fast interaction. In many sketch-based 3D model deformation approaches, such as [3, 4, 5, 6], the selection of the ROI is explicit, where the user is required to sketch a set of strokes, to give the algorithms enough information to define the required deformations. In order to reduce the strokes needed in the interaction, one possible solution is to estimate the ROI from information implied in the underlying geometric features of the models. A key idea is interactive mesh segmentation, which uses relatively small sets of user interactions as hints for dividing a mesh into geometrically meaningful sub-meshes. In the following sections, we will review related work.

Sketch-Based 3D Model Deformation

We begin our review with the work of Draper [3], who proposed a 3D deformation approach which used a user-sketched stroke as a "handle", which the user drags to deform the model elastically according to a grid of control points around the handle. Cherlin [7] used a similar user-stroke as the shape of the cross-section of a rotating surface. Kara [4, 8] proposed approaches that used two strokes with one indicating the original shape of the silhouette and the other guiding the algorithm to deform the surface. Similarly, Kho's work [5] also uses a two-stroke deformation, separately indicating the control points and the deformation. While the above approaches utilise user strokes to deform a model, they do not use the idea of "over-sketching". In comparison, Nealen's [6, 9] and Zimmermann's [10] approaches more directly reproduce the over-sketching concept in 3D model deformation, in which pieces of silhouette segments extracted from models are used to guide a detail-preserving Laplacian deformation.

While most of these approaches require the user to explicitly select the control points and/or the ROI [3, 4, 5, 6], Zimmermann's [10] technique requires only one stroke to detect the corresponding control points and ROI, which is similar to our approach.

Interactive Mesh Segmentation

Like sketch-based modelling, interactive mesh segmentation has recently become a fruitful area of study because of the popularity of touch-based devices. Fan, in his comparative study [11], points out that there are generally four types of interactive mesh segmentation approaches: sketching along the segmentation boundary [12], foreground and background sketching [13, 14], sketching crossing the segmentation boundary [15, 16], and foreground-only sketching [17, 18]. While most of these approaches require more than one stroke, or a particular type of stroke (such as along-boundary, or crossing-boundary sketching), foreground sketching approaches, which use only one stroke, are the most pertinent to our own research. As a result, we adopt Fan's foreground sketching segmentation approach [17], which gradually adds similar patches to the foreground collection while the stroke continues, while the similarity is measured by the

Gaussian mixture model of the shape diameter function (SDF) [19] of the mesh.

2 System Overview

SEED is a sketch-based 3D model deformation approach, with an interface inspired by Teddy [2]. The user's input stroke $\mathbf{c}_{sk}$ is used to guide different phases of the deformation algorithm, as shown in Figure 1. When $\mathbf{c}_{sk}$ is received (Figure 1a), we first extract (Figure 1b) and smooth (Figure 1c) the geometric silhouette curves of the model from the orientation viewed by the user. Next, the silhouette is segmented according to visibility, with hidden silhouette segments being removed from the segment set. Then, $\mathbf{c}_{sk}$ is used to match the nearest visible silhouette curve segment $\mathbf{c}^*_{sil}$, which will be processed according to $\mathbf{c}_{sk}$ to create the expected new silhouette shape $\mathbf{c}_{tgt}$ (Figure 1d). At the same time, the shape diameter function of the mesh is computed (Figure 1e), and used to fit two Gaussian mixture models (GMM) (Figure 1f). According to the GMMs, a region-growing-based binary cut is performed to detect the ROI (Figure 1g). Following that, the ROI is deformed by moving $\mathbf{c}^*_{sil}$ towards $\mathbf{c}_{tgt}$ using Laplacian deformation (Figure 1h). Optionally the ROI can be remeshed before and after the deformation to avoid narrow triangles (triangles with angle smaller than $\pi/6$, hereinafter) in the result. Figure 2 shows some examples of the deformation process described above.

The concept of Zimmermann's [10] approach is similar to SEED, using only a single stroke to extract silhouette curves, to match silhouette segments, and to detect and to deform the ROI. However, there are several differences between SEED and Zimmermann's approach. Firstly, SEED uses geometric silhouettes as control points, which naturally contain the depth information needed in the deformation, while Zimmermann uses the image silhouette, needing a helper scheme to maintain the depth information. Secondly, we adopt a foreground painting mesh segmentation approach [17] to detect the ROIs according to the characteristic of the vertices, while the ROI selection in Zimmermann's approach is done by asking the user to sketch particular shaped strokes that imply the size of the ROIs, according to which the ROIs are marked by intersecting a series of spheres with the meshes. However, this ignores the geometric information of the mesh, such that the size of the ROI is based on strokes rather than the mesh; further, the particular shape of the strokes also reduces the intuitiveness of the approach, since the strokes do not resemble over-sketched strokes. Lastly, SEED allows the detected ROI to be remeshed before and after the deformation phase, which is useful when the vertex count of the ROI is too low and direct deformation may lead to unpleasant artefacts with stretched and non-uniformly distributed triangles.

3 Silhouette Processing

Functioning as the guide of the ROI detection and Laplacian deformation, the silhouette plays an important role, and it is used throughout the processing phases. In this section, we introduce the main operations in the silhouette processing phase, including the extraction and smoothing of the silhouette, hidden silhouette removal, searching for silhouette segments which match the stroke user $\mathbf{c}_{sk}$, and establishing a mapping between the matched silhouette segments and the stroke.

3.1 Extraction and Smoothing

The geometric silhouette of a 3D model is a set of curves which separates the front-facing parts of the model from the back-facing parts. Hertzmann proposed an approach [20, 21] for approximating the geometric silhouette as curves on a triangle mesh by connecting silhouette points obtained by interpolating vertices around the ground truth of the silhouette. Considering the relationship between the vertices and edges of a triangle mesh $\mathbf{S}$ and its silhouette $\mathbf{C}_{sil}$, there are three situations (viewed in orthogonal projection from the positive z-axis):

1. A vertex $\mathbf{v} \in \mathbf{S}$ is on $\mathbf{C}_{sil}$ (Figure 3-1), if the z-component of the normal of the vertex satisfies $n_z = 0$; or
2. An edge $\mathbf{e}(\mathbf{v}_0, \mathbf{v}_1) \in \mathbf{S}$ intersects $\mathbf{C}_{sil}$ at a point $\mathbf{p}^*$ (Figure 3-2), if the z-components of the normals of the edge's two endpoints satisfy $n_{0,z} \cdot n_{1,z} < 0$; otherwise
3. $\mathbf{v}$ or $\mathbf{e}$ is not related to $\mathbf{C}_{sil}$.

Assuming that the normals of the points on $\mathbf{e}$ changes linearly, the intersection $\mathbf{p}^*$ in the second situation can be approximated as

$$\mathbf{p}^* = \alpha \mathbf{v}_0 + (1 - \alpha)\mathbf{v}_1 \ , \tag{1}$$

where $\alpha = \frac{n_{1,z}}{n_{1,z} - n_{0,z}}$. Thus, we have two types of silhouette points: vertex-based silhouette points, which are located at $\mathbf{v}$ in the first situation, and edge-based silhouette points, which are located at $\mathbf{p}^*$ in the second situation. By connecting each silhouette point to its neighbour, we obtain a set of curves which approximate the true silhouette, as shown in Figure 3.

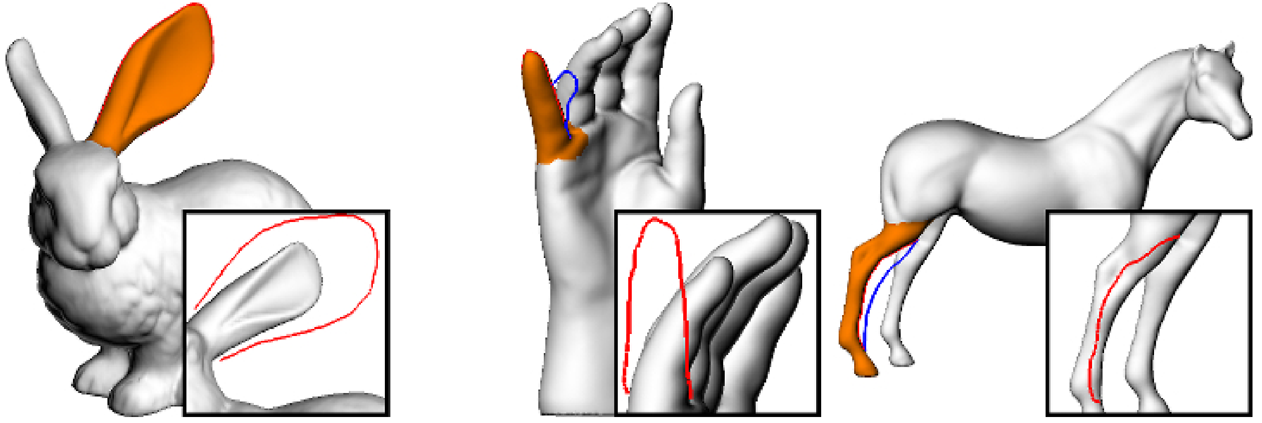

Figure 2: Some examples of deformation with SEED. The red curves are the deforming strokes; the blue curves are the original silhouette; the deformed regions are shown in orange.

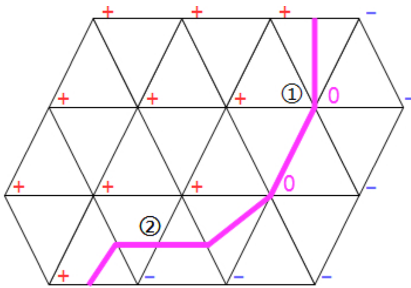

Figure 3: Three types of relationships between the silhouette and the mesh elements. +/-/0: the signs of the z-components of the normals of the vertices. Magenta lines: the trace of the silhouette.

These curves can be used directly as silhouette curves in silhouette deformation [6], however, as shown in Figure 1b, sometimes they are not suitable for the deformation when there are a lot of small geometric details on the surface, since it is difficult to find a unique mapping from the silhouette points to the points on the stroke, which is required for an intuitive deformation.

From experiments with users, we observed that when asked to illustrate the silhouettes of 3D models, they tended to ignore fine details and preferred smoother curves. With this in mind, and the requirement of establishing a unique mapping between silhouette points and stroke points, we smooth the silhouette curves by relaxing the silhouette criterion $n_z = 0$. We introduce two energy functions for each silhouette point $\mathbf{p}$:

$$\begin{cases} E_{geo}(\mathbf{p}) = |n_{p,z}| \\ E_{smth}(\mathbf{p}) = \theta^{\perp}(\mathbf{p}) \end{cases} , \qquad (2)$$

where the operator $\theta^{\perp}(\cdot) \in [0, \pi]$ computes the angle between the two neighbouring silhouette points by projecting them onto the plane perpendicular to the normal of the given point. $E_{geo}(\cdot)$ describes the distance of the silhouette approximation $\widetilde{\mathbf{C}}_{sil}$ from the ground truth $\mathbf{C}_{sil}$, and $E_{smth}(\cdot)$ describes the smoothness of $\widetilde{\mathbf{C}}_{sil}$, as shown in Figure 4. To smooth $\widetilde{\mathbf{C}}_{sil}$, we maximise $E_{smth}(\cdot)$, while allowing $E_{geo}(\cdot)$ to vary, while not exceeding the threshold $\epsilon \in \mathbb{R}^{+}$. Through experiment we have found setting $\epsilon = 0.2$ gives good results.

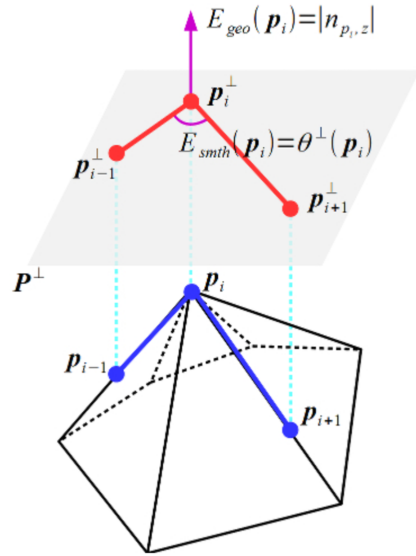

Figure 4: The energy terms used in silhouette smoothing. $\mathbf{p}_i$, $\mathbf{p}_{i-1}$ and $\mathbf{p}_{i+1}$ (blue dots) are silhouette points. $\mathbf{P}^{\perp}$ is a plane perpendicular to the normal $\mathbf{n}_{p_i}$ of $\mathbf{p}_i$. $\mathbf{p}_i^{\perp}$, $\mathbf{p}_{i-1}^{\perp}$ and $\mathbf{p}_{i+1}^{\perp}$ (red dots) are the projections of $\mathbf{p}_i$, $\mathbf{p}_{i-1}$ and $\mathbf{p}_{i+1}$ on $\mathbf{P}^{\perp}$.

To apply the energy optimisation, we adopt a greedy Snake algorithm [13] to iteratively improve the smoothness of $\widetilde{\mathbf{C}}_{sil}$. The algorithm, like a snake adjusting its posture to better adapt to its environment, takes a set of points (known as "snaxels") approximating a curve, and gradually adjusts their locations to improve the energy distribution of the curve. At the beginning of the optimisation, we associate each silhouette point with a snaxel. According to the type of corresponding silhouette point, a snaxel may be vertex-based or edge-based, determined according to Equation 1. To optimise the energy of the snaxels, we iteratively relocate, split or remove them with respect to $E_{geo}(\cdot)$ and $E_{smth}(\cdot)$, as well as their relationship to neighbouring snaxels, as follows:

1. A snaxel $\mathbf{s}_i$ will be removed from the mesh (Figure 5a), if itself and its two neighbour snaxels, $\mathbf{s}_{i-1}$ and $\mathbf{s}_{i+1}$, are within the same triangle; or,
2. a vertex snaxel $\mathbf{s}_i$ will be split into several edge snaxels $\mathbf{s}'_i$ located at corresponding edges connecting $\mathbf{s}_i$ according to $\mathbf{s}_{i-1}$ and $\mathbf{s}_{i+1}$ (Figure 5b), if $E_{smth}(\mathbf{s}_i) < \pi$; or,
3. an edge snaxel $\mathbf{s}_i$ will be relocated along its edge (Figure 5c) so that $E_{smth}(\mathbf{s}_i)$ is maximised while keeping $E_{geo}(\mathbf{s}_i) < \epsilon$; if doing so $\mathbf{s}_i$ is moved to an endpoint of its edge, $\mathbf{s}_i$ will be converted to a vertex snaxel.

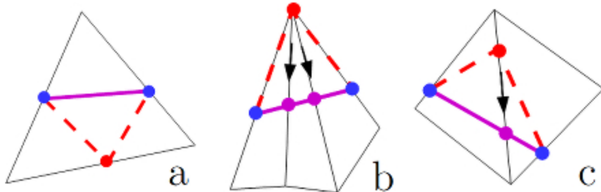

Figure 5: Snaxel operations: (a) snaxel removal, (b) snaxel splitting, and (c) snaxel relocation. The blue/red/purple dots are respectively the original/modified/new snaxels.

Starting from an end point of a given silhouette curve, we apply the above operations to the curve iteratively until no further splitting or removing occurs. Since there might be more than one silhouette curve comprising $\mathbf{C}_{sil}$, the iterative optimisation is applied to each curve separately.

3.2 Hidden Silhouette Removal

As we have mentioned, in traditional hand illustration, users often over-sketch. To accommodate this natural tendency, we remove invisible silhouette segments before matching the silhouette curves to the sketch. Hertzmann pointed out [20] that the visibility changes in silhouette curves happen at key points such as cusps and self-intersections, as well as at intersections between different silhouette curves on the image plane, as shown in Figure 6. Correspondingly, to remove hidden silhouette segments, we firstly detect these key points on the image plane. (We define a cusp as occurring when 3 consecutive silhouette points form an angle smaller than $\pi/2$ on the image plane.) With these key points, the silhouette curves can be divided into a collection of silhouette segments, each of which has the same visibility. However, due to the smoothing operation performed prior to this step, there may be some points, which were originally visible but are now covered with triangles. To avoid such "false invisibility", we assume that a curve is visible if and only if there is at least one point on it which is visible. This assumption is reasonable since the affected segments are those curves that are originally visible, so originally invisible curves will not become visible.

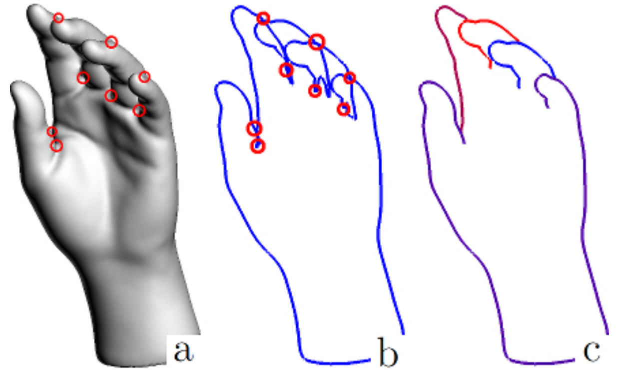

Figure 6: Hidden silhouette removal: (a) the model; (b) the silhouette; (c) the silhouette after removing invisible segments. The red circles indicate the points at which the visibility changes.

In order to test the visibility of these silhouette segments, the visibility of each silhouette point must be tested, which involves traversing the triangles of the mesh. Because SEED is based on orthogonal projection, the test can be simplified to a 2D triangle-point containment test. Thus, we first use a simple bounding box test to cull irrelevant triangles; for the remaining candidate triangles, the triangle-point containment test is performed. We implement this test on the GPU, with a separate thread for testing each triangle and silhouette-point. We have found that a GPU-based test is twice as fast as a multi-threaded CPU counterpart.

3.3 Matching Silhouette Segments

Again taking into account our observations of traditional sketching, we assume that the user's stroke is intended to control the deformation of the nearest silhouette segment to the stroke. We define the distance D between a silhouette segment $\mathbf{c}_{sil}$ and the user's stroke $\mathbf{c}_{sk}$ as the average of the image plane distance between the closest silhouette/stroke point pair:

$$D(\mathbf{c}_{sil}, \mathbf{c}_{sk}) = \frac{1}{n}\sum_{i=1}^{n} ||\mathbf{p}_i, \mathbf{q}_i^*||^{\perp} \ , \qquad (3)$$

where n is the number of points in the stroke; $||\cdot||^{\perp}$ computes the image plane distance between two 3D points; $\mathbf{p}_i$ is a stroke point; and given $\mathbf{p}_i$, $\mathbf{q}_i^*$ is a point that is the closest to $\mathbf{p}_i$ in $\mathbf{c}_{sil}$. With this definition, we compute the distance between every silhouette segment and the stroke, and choose the closest one, from which we retrieve the matched silhouette segment $\mathbf{c}_{sil}^*$ which is between the closest silhouette points of the two endpoints of the stroke.

3.4 Mapping Matched Silhouette Segment to Stroke

The points of the matched silhouette segment $\mathbf{c}_{sil}^*$ can be used directly as the control points in the deformation, and to guide the movement of the control points we need to establish a mapping between these points and the points on the stroke $\mathbf{c}_{sk}$. Thus, we search the closest points on $\mathbf{c}_{sk}$ from $\mathbf{c}_{sil}^*$ by using relative positions. Given the points $\mathbf{p}_i$ of a given curve $\mathbf{c}$, its relative position r_i^c on $\mathbf{c}$ is defined as:

$$r_i^c = \frac{l_{(0,i)}}{l_c} \ , \qquad (4)$$

where $l_{(0,i)}$ is the length of the curve from one endpoint $\mathbf{p}_0$ of $\mathbf{c}$ to $\mathbf{p}_i$; and l_c is the length of the whole curve. The use of the relative positions in mapping one curve to another mimics the behaviour of an elastic rope: the curve will deform uniformly when an outside force is applied to the curve, as shown in Figure 7.

Once the relative positions of points on $\mathbf{c}_{sil}^*$ and $\mathbf{c}_{sk}$ are computed, we map a silhouette point $\mathbf{q}_j$ to a stroke point $\mathbf{p}_i$ if their relative positions r_j^{sil*} and r_i^{sk} are the same. Usually there will not be a direct match, in which case we interpolate the closest stroke points of $\mathbf{q}_j$, in term of relative position, to obtain the matched point $\mathbf{p}_j^*$ on $\mathbf{c}_{sk}$, as follows:

$$\mathbf{p}_j^* = \alpha\mathbf{p}_i + (1-\alpha)\mathbf{p}_{i+1} \ , \qquad (5)$$

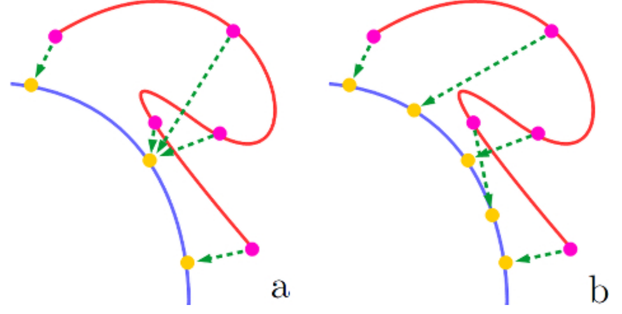

Figure 7: Searching silhouette/sketch point pairs with (a) Euclidean distance and (b) relative positions. Blue/red curve: silhouette segment/sketched stroke.

where the coefficient $\alpha = \frac{r_{i+1}^{sk} - r_j^{sil}}{r_{i+1}^{sk} - r_i^{sk}}$.

After matching a stroke point for each silhouette point, we then assign the matched stroke points with the depth value of their corresponding silhouette point, to derive the target curve $\mathbf{c}_{tgt}$. Since there is a gap between $\mathbf{c}_{sil}^*$ and $\mathbf{c}_{tgt}$, we merge the position of the point pair $\mathbf{p}_i \in \mathbf{c}_{tgt}$ and $\mathbf{q}_i \in \mathbf{c}_{sil}^*$ to achieve a smooth transition according to their relative positions r_i, as shown in Figure 1e:

$$\mathbf{p}_i' = f(r_i)\mathbf{p}_i + [1 - f(r_i)]\mathbf{q}_i \ , \qquad (6)$$

where the merging ratio $f(r_i)$ is defined as

$$f(r_i) = \begin{cases} g(r_i/\omega) & r_i < \omega \\ g(\omega - r_i/\omega) & r_i > 1-\omega \\ 1 & \text{otherwise} \end{cases} \ ; \qquad (7)$$

where $g(\cdot)$ is a monotonically increasing mapping from $[0,1]$ to $[0,1]$; and the threshold $\omega > 0$ controls the degree of merging. Through experiment, we choose $\omega = 0.2$ and use the square root function for $g(\cdot)$.

4 Automatic ROI Detection

The goal of SEED is to require the user to sketch as few strokes as possible to deform a 3D model. We note that in traditional illustration, an over-sketched stroke not only gives information on how the illustration will be deformed, but also hints where to deform and the size of the region of interest (ROI). How can we mimic human recognition in order to estimate the ROI, based on the stroke sketched by the user ? One way to solve this problem is to use mesh segmentation techniques, which use implied geometric information to separate different parts of a mesh. Inspired by Fan's work [17], we utilise the shape diameter function [19] to compute a mesh segmentation on the fly, according to which the ROI can be estimated.

4.1 GPU-Based Shape Diameter Function Evaluation

The Shape Diameter Function (SDF) is a volume metric used in mesh segmentation. It gives a measure of the "thickness", or diameter, of a given vertex in a mesh, by casting a set of rays, typically around 30, from the vertex towards the opposite direction of its normal, as shown in Figure 8. The rays cast are constrained to a cone, typically with an angle of 60 degrees, in order to eliminate noise caused by small details of the mesh.

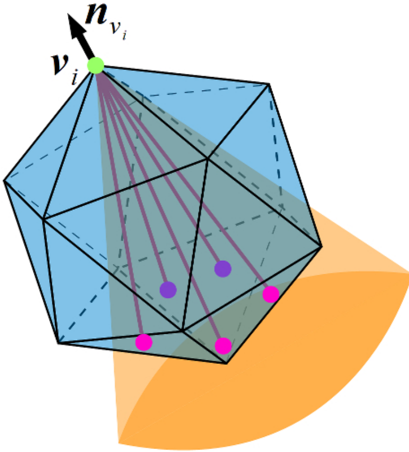

Figure 8: SDF evaluation. Green dot: a vertex $\mathbf{v}_i$ of the mesh, with normal $\mathbf{n}_{v_i}$. Purple lines: rays cast from $\mathbf{v}_i$. Magenta/purple dots: the intersections of the rays and the triangles of the mesh.

Because the evaluation of the SDF is computationally expensive due to the ray casting process, we subdivide the space using an octree, which greatly improves the run-time performance; however, the ray casting algorithm is still relatively slow (Table 1: CPU). To make the algorithm fast enough for real-time interaction, we adopt a GPU-based algorithm to accelerate the ray casting, vectorising the octree to store in GPU memory. Instead of looping through every vertex and every ray, we create a thread for each vertex-ray combination. According to the GPU used in our implementation, 32 rays are used for each vertex to fit in a thread warp, within which the performance of global memory accessing can be boosted with a coalesced access mechanism [22]. The performance is at least twice as fast as its CPU-based counterpart in our experiments (Table 1: GPU), including the time to vectorise the octree.

Table 1: Run-time performance of the ray casting for SDF evaluation.[a]

Model	CPU	GPU
Lo-Res Bunny[b]	56.5 ms	34.2 ms
Mid-Res Bunny[c]	558 ms	261 ms
Hi-Res Bunny[d]	4.93 s	2.04 s

a. The timing data in the table are obtained by averaging the duration of 100 executions. The CPU used in our tests is an Intel Core i7-3630QM; the GPU is an NVIDIA GeForce GT-650M. The settings are consistent throughout our experiments.
b. 5k facets, hereinafter the same.
c. 30k facets, hereinafter the same.
d. 180k facets, hereinafter the same.

4.2 Gaussian Mixture Model Fitting

Once the SDF values of the mesh vertices have been computed, we can use the values to guide the mesh segmentation. To extract similar geometric features, we fit the SDF values to two Gaussian mixture models (GMM): one for the whole model, and the other for the "foreground" of the selection, as shown in Figure 1g. For the GMM of the whole model, we sample 2048 SDF values from the entire mesh to do the fitting; for the foreground, we use the vertices around the matched silhouette segment, or seeds, to do the fitting. For some low resolution models with a small vertex count, the number of seeds may be too low to be used in fitting the GMM. To solve this issue, we expand the seed area to the vertices whose geodesic distance to the nearest seed vertex is within a tolerance (we experimentally use 0.1), as shown in Figure 9.

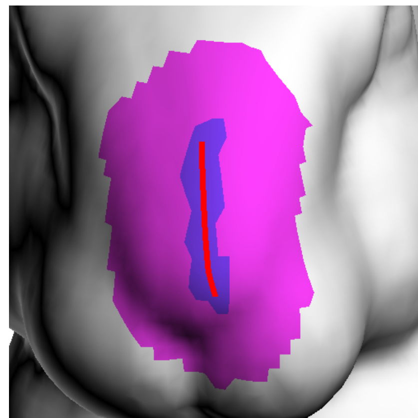

Figure 9: The original (blue) and the expanded (purple) seed region. The red curve is the matched silhouette segment.

To fit the GMMs, we employ a greedy expectation-

Table 2: Run-time performance of CPU-based greedy GM (4 components) algorithm and its GPU-based counterpart.

Test Case	CPU	GPU
Random Numbers[a]	356 ms	23.6 ms
Lo-Res Bunny[b]	250 ms	9.51 ms
Mid-Res Bunny[b]	241 ms	9.04 ms
Hi-Res Bunny[b]	261 ms	12.0 ms

a. 2048 random numbers uniformly distributed in $[0, 1]$.
b. 2048 samples from the SDF.

maximisation (EM) algorithm [23, 24]. The main operations here are summing up values and finding extreme values, which can be greatly accelerated with a reduction-based algorithm on the GPU [25]. In such an algorithm, the memory used to share temporary results (the "workspace") must be synchronised between all the threads. Restricted by the GPU specification [26], the maximal number of threads that can be synchronised internally during a single GPU program call is 1024, beyond which the synchronisation of threads must be done externally by CPU scheduling, which is expensive. Considering this, we use (at most) 2048 SDF samples to fit the GMM, so that each thread will handle 2 samples, which maximises the potential of GPU memory usage and results in a performance gain as much as 20 times compared to using the CPU (Table 2).

4.3 Detecting the ROI

The goal of ROI detection is to mark some vertices as "selected", while leaving other vertices alone. This is a binary cut problem, which labels the whole vertex set with two different colours by optimising appropriate energy functions. We adopt the energy function scheme of [17], which uses a connectivity term Ec and a smoothness term Es to define the energy function to be minimised:

$$E = \sum_{\mathbf{v} \in \mathbf{S}} Ec_v + \lambda \sum_{\mathbf{v} \in \mathbf{S}} Es_v \quad (8)$$

where $\mathbf{S}$ is a triangle mesh; $\mathbf{v}$ is a vertex in $\mathbf{S}$; and λ is the weight of mixing the two terms. We set λ to 0.2 throughout our experiments to balance the aggressiveness of the ROI expansion and the smoothness of the border of the ROI. With the energy function, we use a simple region growing algorithm starting from the foreground seeds to detect the ROI: by traversing the unselected neighbour vertices of the foreground, we compare the energy of assigning a vertex to the background against the energy of assigning it to the foreground. If the energy of foreground is smaller than the background energy, we mark it as "selected". The algorithm stops when the border vertices of the ROI reach a balance of foreground and background energy.

4.4 Smoothing the border of the ROI

The result of the above ROI detection could be a border which is a zigzag. In order to smooth the border such that the ROI and the rest of the mesh maintain a smooth transition, we adopt a greedy algorithm. We categorise the vertices in the ROI into three types based on their neighbouring vertices, and process them accordingly:

1. An ROI vertex is called "convex" if it has no more than two neighbouring ROI vertices, as shown in Figure 10a. All convex ROI vertices will be removed from the ROI.
2. An ROI vertex is called "concave", if it has only one neighbouring vertex that is not an ROI vertex, as shown in Figure 10b. All non-ROI neighbour vertices of concave ROI vertices will be added to the ROI.
3. Otherwise, nothing will be done.

A priority queue is used to store the convex and the concave vertices and their neighbours. While the queue is not empty, we pop the first element from the queue, and process it according to its type. Once a convex or concave vertex is processed, all its neighbour vertices are added to the queue. The process continues until there are no convex or concave vertices in the queue. A demonstration of the effect of the algorithm is shown in Figure 10c.

5 Deforming and Remeshing ROI

Inspired by [6], we adopt a Laplacian mesh editing [27, 28] approach to move the matched silhouette segment to the target curve, while keeping the boundary of the ROI attached to the rest of the mesh. The nature of Laplacian deformation is to stretch the mesh to fit the target points, which will result in narrow triangles when deforming a mesh which has a relatively small vertex count, or is not evenly-triangulated. To obtain a deformation with a consistent and smooth look, we remesh the ROI before and after the deformation, in case the direct deformation result is unsatisfactory due to an irregular triangulation.

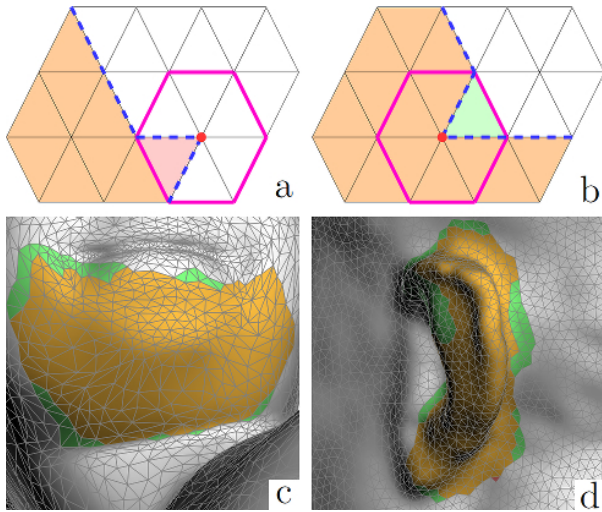

Figure 10: ROI border smoothing: (a) deselecting a convex vertex; (b) selecting all neighbour vertices of a concave vertex; (c) and (d) ROI smoothing example. In (a) and (b), the orange/white triangles are ROI/unselected regions; the red dots are the convex/concave vertices; the red/green triangles are newly deselected/selected ROI triangles. In (c) and (d), the orange regions are the original ROIs, and the red/green triangles are the deselected/additional ROI triangles after applying smoothing.

5.1 Laplacian Deformation

Laplacian deformation involves solving a linear system trying to satisfy deforming constraints, while keeping the Laplacian coordinate of each vertex unchanged as much as possible. The matrix of the linear system is a sparse matrix, since the computation of Laplacian coordinates only depends on a small collection of vertices neighbouring a given vertex. Therefore, we employ a direct solver [29] to solve the linear system, the matrix of which is constructed using cotangent operator [30]. Since the operator only uses weighted connection information of a given vertex, there is no correlation between the three components of the space/Laplacian coordinates. As a result, we can decompose the linear system to obtain three smaller-sized sub- linear systems for each coordinate component independently, without affecting the solution of the original linear system. By using three threads to solve the sub- linear systems simultaneously, the duration of solving Laplacian system can be shortened (Table 3: ST & MT).

Nealen suggested to adjust the orientations of the vertices to improve the quality of the mesh after deformation [6, 9]. However, this will include a rotation for each vertex in the deformation, which will result in a denser matrix; what's more, the coordinate components are no longer uncorrelated, so that we cannot decompose the matrix into smaller matrices for multi-threaded acceleration. As a result, to adjust vertex orientation, the duration of the deformation will be much longer than deforming without it (Table 3: AdjNorm). Since run-time performance is more important than mesh quality in our system, we do not adjust vertex orientation by default, and leave it as an option to users.

Table 3: Run-time performance of solving linear systems with different methods.[a]

Model	AdjNorm[b]	ST[c]	MT[d]
Lo-Res Bunny	28.3 ms	3.71 ms	2.24 ms
Mid-Res Bunny	146 ms	35.0 ms	19.4 ms
Hi-Res Bunny	2.41 s	364 ms	161 ms

a. Deforming the bunny models with the stroke shown in Figure 2a.
b. Deforming and adjusting the orientations of the vertices.
c. Single-threaded deformation.
d. Multi-threaded deformation.

5.2 ROI Remeshing

An issue of Laplacian deformation is that it only stretches the vertices to perform the deformation, without adjusting the connectivity and the number of the vertices. This is generally acceptable when dealing with a well-triangulated mesh and/or the scale of the deformation is relatively small. For meshes not well-triangulated or for a relatively massive deformation, however, the resulting mesh can be full of narrow triangles, which will affect the appearance of the mesh. In order to overcome this problem, we remesh the ROI before and after the deformation. To do this, we firstly retrieve the average edge length of the ROI, according to which the remeshing algorithm will be guided. For the remeshing before the deformation (Figure 11c), the average edge length will be scaled according to the "degree of stretching", which is defined by the ratio of the length of the matched silhouette segment to the length of the target curve. The purpose of scaling the guide edge length is to try to maintain the average edge length of the ROI after the remeshing. Since the deformation will always introduce stretched triangles (Figure 11d), we perform another remeshing after the deformation trying to restore the average edge length of the original ROI (Figure 11e). The remeshing technique we use is incremental remeshing [31, 32], which gradually optimises the connectivity and the distance of the ver-

tices, leading to a uniformly distributed mesh.

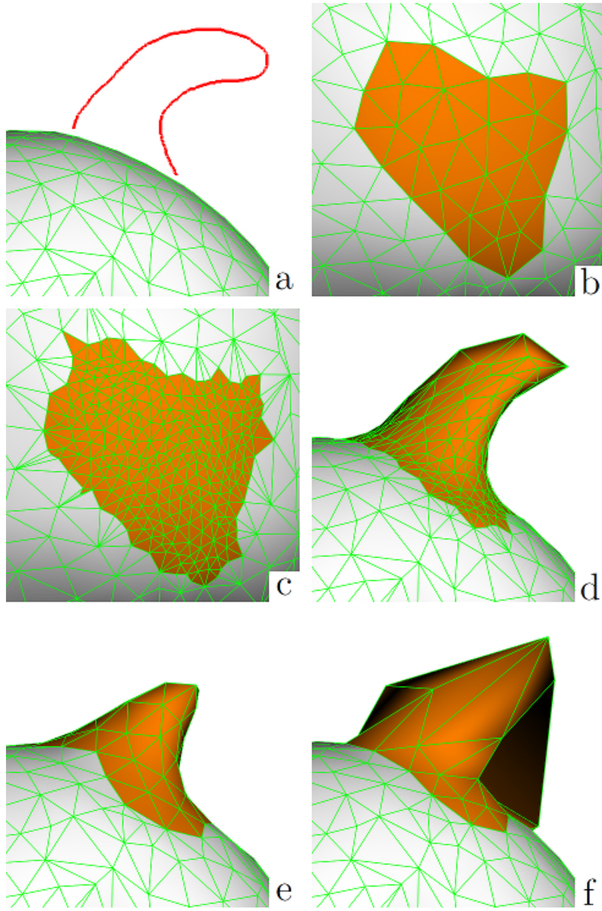

Figure 11: ROI remeshing and deformation: (a) deforming stroke; (b) the ROI; (c) remeshing before deformation; (d) deformation result; (e) remeshing after deformation; (f) deformation without remeshing.

6 Usability Study

We recruited 24 subjects to participate in a usability study, during which each subject was asked to perform a series of tasks, split into three "modules", and to rate their satisfaction with the outcome of their tasks.

6.1 Silhouette Smoothing

The first test module evaluated the user experience of the silhouette smoothing. We prepared a series of models to demonstrate a part of the silhouette curve with and without smoothing. The subjects were shown these models, the sequence of which was randomised. For each model both the smoothed and unsmoothed silhouette segments were presented to the user without revealing which was which. After displaying each model and the corresponding silhouette segment, the subjects were asked to rate how the presented silhouette segment met their expectation, from 0 (totally unexpected) to 4 (exactly as expected). The result (Table 4: 1) shows consistency in preference of the smoothed silhouette segment using our algorithm, which implies that people tend to ignore small details in the silhouette, and that the sacrifice of accuracy in favour of smoothness is a reasonable strategy.

Table 4: Average user expectation ratings from our usability study of SEED[a].

Test	Rating	
1	without smoothing 2.63(±0.48)	with smoothing 3.28(±0.18)
2.1	3.04(±0.42)	
2.2	manual 2.10(±0.31)	automatic 3.66(±0.09)
3.1	2.97(±0.32)	
3.2	3.17(±0.31)	

a. On a scale of 0 to 4, see text for definition. The numbers in parentheses are the standard deviation of the data.
1. Satisfaction of silhouette extraction.
2.1. Satisfaction of ROI detection.
2.2. Ease of use of ROI selection.
3.1. Satisfaction of deformation.
3.2. Responsiveness of system.

6.2 ROI Detection

The second module evaluated the user experience of the ROI detection. Similar to the silhouette extraction experiment, a subject was presented with random models showing the detected ROI. They were asked to rate whether the detected ROI met their expectation, using the same scale as above. In addition, we asked the subjects to use both the sketch-based ROI detection and traditional paint and lasso, sketching operations, to perform given ROI selection tasks, after which they were asked to rate the ease of use, from 0 (extremely awkward) to 4 (extremely easy), of both selection approaches. The results (Table 4: 2.1 and Table 4: 2.2) indicate that the shape of ROIs detected with our algorithm are generally acceptable to the users, and also that the ROI detection algorithm required less effort than the manual selection.

6.3 Sketch-Based Deformation

The third module evaluated the overall user experience of SEED. In this part of the experiment, the subjects were asked to complete tasks to apply specific deformations to given models, after which they were asked to rate the compliance of the deformed shape to their expectation, as well as the speed of the real-time experience, rating both from 0 (extremely unacceptable) to 4 (excellent). The results, which we find encouraging, are summarised in Table 4: 3.1 and Table 4: 3.2.

7 Conclusion and Future Work

We have presented SEED, a system for intuitive real-time deformation of 3D models. While the system performs robustly, and has been favourably assessed in our usability studies, there are a number of areas where improvements can be made. One drawback of our ROI detection algorithm is that sometimes it tends to produce a ROI larger than the user expects, and we are investigating alternative mesh segmentation techniques to give more intuitive results, while not sacrificing performance. Again considering performance, we have used the GPU to greatly improve the run-time performance of the ray casting involved in the evaluation of the shape diameter function. However, the performance could be further improved if the octree could be directly generated on GPU, speeding up not only the creation phase itself, but also reducing the time for vectoring and loading the octree to the GPU. We also intend to further exploit the GPU by implementing a GPU-based direct sparse linear system solver, in contrast to iterative solvers such as Cusp [33], to speed up our deformation solving phase.

In our current system, the user can only deform the silhouette of a given model. We intend to experiment with deforming feature lines, as in, for example [6]. The challenge here is to detect the feature lines according to the user's sketch, while maintaining real-time performance.

8 Acknowledgements

This research is funded by the UK/China Joint Scholarship for Excellence. We thank all our volunteer subjects for their assistance in our user experiments.

References

[1] Luke Olsen, Faramarz F Samavati, Mario Costa Sousa, and Joaquim A Jorge. Sketch-based modeling: A survey. *Computers & Graphics*, 33(1):85–103, 2009.

[2] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3D freeform design. In *ACM SIGGRAPH 2007 courses*, page 21. ACM, 2007.

[3] Geoffrey Draper and Parris K Egbert. A gestural interface to free-form deformation. In *Graphics Interface*, volume 2003, pages 113–120, 2003.

[4] Levent Burak Kara and Kenji Shimada. Construction and modification of 3D geometry using a sketch-based interface. In *Proceedings of the Third Eurographics conference on Sketch-Based Interfaces and Modeling*, pages 59–66. Eurographics Association, 2006.

[5] Youngihn Kho and Michael Garland. Sketching mesh deformations. In *ACM SIGGRAPH 2007 courses*, page 41. ACM, 2007.

[6] Andrew Nealen, Olga Sorkine, Marc Alexa, and Daniel Cohen-Or. A sketch-based interface for detail-preserving mesh editing. In *ACM SIGGRAPH 2007 courses*, page 42. ACM, 2007.

[7] Joseph Jacob Cherlin, Faramarz Samavati, Mario Costa Sousa, and Joaquim A Jorge. Sketch-based modeling with few strokes. In *Proceedings of the 21st spring conference on Computer graphics*, pages 137–145. ACM, 2005.

[8] Levent Burak Kara, Chris M D'Eramo, and Kenji Shimada. Pen-based styling design of 3D geometry using concept sketches and template models. In *Proceedings of the 2006 ACM symposium on Solid and physical modeling*, pages 149–160. ACM, 2006.

[9] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. FiberMesh: Designing freeform surfaces with 3D curves. *ACM Transactions on Graphics (TOG)*, 26(3):41, 2007.

[10] Johannes Zimmermann, Andrew Nealen, and Marc Alexa. Sketching contours. *Computers & Graphics*, 32(5):486–499, 2008.

[11] Lubin Fan, Min Meng, and Ligang Liu. Sketch-based mesh cutting: A comparative study. *Graphical Models*, 74(6):292–301, 2012.

[12] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. *ACM Transactions on Graphics (TOG)*, 23(3):652–663, 2004.

[13] Zhongping Ji, Ligang Liu, Zhonggui Chen, and Guojin Wang. Easy mesh cutting. *Computer Graphics Forum*, 25(3):283–291, 2006.

[14] Juyong Zhang, Chunlin Wu, Jianfei Cai, Jianmin Zheng, and Xue-cheng Tai. Mesh snapping: Robust interactive mesh cutting using fast geodesic curvature flow. *Computer Graphics Forum*, 29(2):517–526, 2010.

[15] Youyi Zheng and Chiew-Lan Tai. Mesh decomposition with cross-boundary brushes. *Computer Graphics Forum*, 29(2):527–535, 2010.

[16] Youyi Zheng, Chiew-Lan Tai, and OK-C Au. Dot scissor: A single-click interface for mesh segmentation. *Visualization and Computer Graphics, IEEE Transactions on*, 18(8):1304–1312, 2012.

[17] Lubin Fan, Kun Liu, et al. Paint mesh cutting. *Computer Graphics Forum*, 30(2):603–612, 2011.

[18] Pengfei Xu, Hongbo Fu, Oscar Kin-Chung Au, and Chiew-Lan Tai. Lazy selection: A scribble-based tool for smart shape elements selection. *ACM Transactions on Graphics (TOG)*, 31(6):142, 2012.

[19] Lior Shapira, Ariel Shamir, and Daniel Cohen-Or. Consistent mesh partitioning and skeletonisation using the shape diameter function. *The Visual Computer*, 24(4):249–259, 2008.

[20] Aaron Hertzmann. Introduction to 3D non-photorealistic rendering: Silhouettes and outlines. *Non-Photorealistic Rendering. SIGGRAPH*, 99, 1999.

[21] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526. ACM Press/Addison-Wesley Publishing Co., 2000.

[22] NVIDIA OpenCL best practices guide, version 1.0. `http://developer.download.nvidia.com/compute/cuda/2_3/opencl/docs/NVIDIA_OpenCL_BestPracticesGuide.pdf`, 2009.

[23] Sanjoy Dasgupta. Learning mixtures of Gaussians. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 634–644. IEEE, 1999.

[24] Nikos Vlassis and Aristidis Likas. A greedy EM algorithm for Gaussian mixture learning. *Neural Processing Letters*, 15(1):77–87, 2002.

[25] Matthew Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning, 2012.

[26] CUDA C programming guide, version 5.5. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2013.

[27] Olga Sorkine, Daniel Cohen-Or, Yaron Lipman, Marc Alexa, Christian Rössl, and H-P Seidel. Laplacian surface editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 175–184. ACM, 2004.

[28] Yaron Lipman, Olga Sorkine, Daniel Cohen-Or, David Levin, Christian Rossi, and Hans-Peter Seidel. Differential coordinates for interactive mesh editing. In *Shape Modeling Applications, 2004. Proceedings*, pages 181–190. IEEE, 2004.

[29] Piotr Wendykier. CsparseJ: A Java port of concise sparse matrix package. `https://sites.google.com/site/piotrwendykier/software/csparsej`, 2009.

[30] Mark Meyer, Mathieu Desbrun, Peter Schröder, Alan H Barr, et al. Discrete differential-geometry operators for triangulated 2-manifolds. *Visualization and mathematics*, 3(2):52–58, 2002.

[31] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 19–26. ACM, 1993.

[32] Mario Botsch and Leif Kobbelt. A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 185–192. ACM, 2004.

[33] Cusp: A C++ templated sparse matrix library. `https://code.google.com/p/cusp-library/`, 2013.