CSE 528 Final Report: Cross-Boundary Brushes

Yuren Huang

Department of Computer Science, Stony Brook University

Abstract

This project implements a representative type of sketch-based mesh cutting: cross-boundary brushes. Given a triangular mesh, the system takes one or multiple user strokes as input and in real time produces a cut across the strokes. There are two types of brushes available to meet different goals: the part-brush, designed for cutting semantic parts, based on isolines of a harmonic field; and the patch-brush, built to extract flatter surface patches, using region growing with a face-based metric. The system is easy to use and generally produces satisfying results.

1. Algorithm Overview

This project is based on [1], the details of which have been largely introduced in the paper report and presentation.

1.1. part-brush

The essence of the part-brush is the isolines of a harmonic field. Given a number of user strokes, a harmonic field is computed by solving a Laplace's equation

$$\Delta\Phi = 0$$

with boundary constraints $\Phi(x) = 1, \forall x \in U$ and $\Phi(x) = 0, \forall x \in V$, where $U = \{p_1, p_2, ..., p_c\}$ and $V = \{q_1, q_2, ..., q_c\}$ are the corresponding sets of start and end mesh vertex indices of the user strokes, c being the number of strokes, and Δ being the Laplace operator with cotangent weighting.

Define $f : S^2 \to \mathbb{R}$ to be the solution of

$$\Delta f = 0, f(x_1) = 0, f(x_2) = 1 \text{ for some } x_1, x_2 \in S^2.$$

Is it possible that $\exists x \in S^2$ s.t. $f(x) > 1$ or $f(x) < 0$?

1.2. *patch-brush*

The patch-brush applies region growing algorithm with the following cost function:

$$cost_{ij} = (1 + |\theta_{ij}|)|\vec{n_i} - \vec{n_j}|,$$

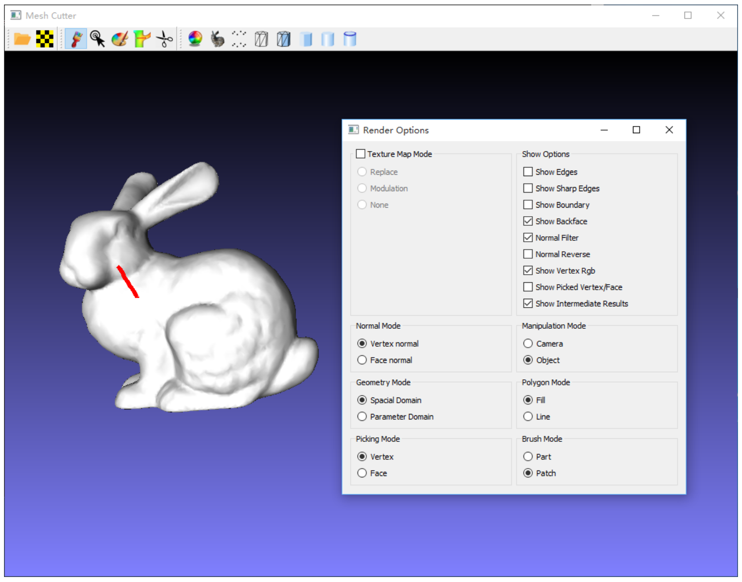
where $\theta_{ij} = \frac{\vec{n_i} \cdot \vec{e_{ij}}}{|\vec{e_{ij}}|}$, and $\vec{e_{ij}}$ denotes the vector from the center of face i to that of its adjacent face j. Every time, an ungrouped face with smallest $cost_{ij}$ is absorbed by its neighboring group.

2. Implementation

Like many major graphics projects, this project is written in C++ with OpenGL. The developing environment is Microsoft Visual Studio 2013, with QT Plugin. Details of the libraries used will be given in Section 4. The system has three critical components: the user interface (the `CMeshViewer` class), the part-brush functions (the `CIsoline` class) and the patch-brush functions (appearing as sub-modules in `CMeshViewer`).

2.1. *User Interface (`CMeshViewer`)*

As shown in Figure 1, the user interface is built with QT. It has a handy toolbar which includes all the core functionalities. For some buttons, hotkeys are also available to speed up the operation. Besides, a comprehensive set of parameters could be set via a menu.



Texture Map Mode ☐
Replace / Modulation / None

Show Options
☐ Show Edges
☐ Show Sharp Edges
☐ Show Boundary
☑ Show Backface
☑ Normal Filter
☐ Normal Reverse
☑ Show Vertex Rgb
☐ Show Picked Vertex/Face
☑ Show Intermediate Results

Normal Mode: [✓ Vertex normal] Face normal
Manipulation Mode: Camera [✓ Object]
Geometry Mode: [✓ Spacial Domain] Parameter Domain
Polygon Mode: [✓ Fill] Line
Picking Mode: [✓ Vertex] Face
Brush Mode: Part [✓ Patch]

Figure 1: The QT-based user interface

For a sketch-based system, how to deal with user strokes is the key part. As common, here a user stroke consists of continuous pixels on the near (in contrast to far) 2D plane over the mesh, as shown in Figure 1. The width of the stroke is enlarged to provide better visual effects. For every stroke, the projections of its starting and ending points are computed to obtain the corresponding vertices and faces on the mesh, which will later serve as constraints or initial conditions of the algorithm.

Since frequent changes of displaying occur in this project, it is not practical to separate everything from the viewer. Thus some core functions are integrated into the `CMeshViewer` class, like `cutMesh()` (label every vertex the part it belongs to, given the isoline) and `regionGrow()` (the whole region growing algorithm, not complicated).

2.2. Part-brush (`CIsoline`)

The discrete version of the Laplace equation in Section 1.1 is the linear system $A\Phi = b$, where

$$A = \begin{bmatrix} L \\ WP \end{bmatrix} \text{ and } b = \begin{bmatrix} 0 \\ WB \end{bmatrix}.$$

$L = D - C$, where $C_{ij} = \frac{1}{2}(\cot\alpha_{ij} + \cot\beta_{ij})$, with α_{ij} and β_{ij} being opposite angles to the edge (i, j). For the pair (i, j) not being an edge, $C_{ij} = 0$. And D is the row sums of C. The positional weighting matrix W, the positional matrix P and the positional vector b are fully explained in [1]. As suggested by the paper, the `CHOLMOD` package is used to solve this linear system (more precisely, to solve the equivalent normal equation $A^T A\Phi = A^T b$).

The **most difficult** part (which the paper doesn't cover at all) is the extraction of isolines. Here the method applied to this problem could be called "Meandering triangles" (there is an algorithm of the same name [2], but the algorithm here is original work). The idea is very simple: since an intersection point of an isoline and the mesh is either a vertex or a point on an edge, we find them one by one.

Step 1 Find an edge such that the isovalue lies between the field values of its two endpoints. The isoline must cross this edge. Mark this edge. (If the isovalue equals one endpoint's field value, simply mark that vertex. This should be considered as in extreme cases an isoline only goes along edges but never cross an edge.)

Step 2 From the edge (or vertex) found in Step 1, find the next edge (or vertex) the isoline crosses. This can be done by searching the neighborhood (for edge case, search the other four edges in adjacent triangles; for vertex case, search the 1-ring neighborhood).

Step 3 When again reaching the starting edge (or vertex), it means we have found a loop, which is the isoline. Done.

Details of this algorithm are in the function `getIsoline()` in `CIsoline`.

It should be noticed that in the project an "isoline" is described by a `ISO` struct, consisting of its isovalue and a vector of `LOOP` structs. This design considers the possibility of finding multiple isolines (`LOOP`s) corresponding to one isovalue. In the end, those isolines not passing through the user strokes will be filtered.

In the selection of isolines, the paper merely says that isoline with the smallest metric should be choosed but doesn't mention the possibility of positive concaveness term Δ_i. If that occurs, conflict occurs, since a smaller positive final metric M_i could mean a smaller centerness C_i. Thus we normalize the Δ_i's to $[-1, 0]$ in advance. (To recall these metrics, please refer to my paper report or the paper [1].)

2.3. Patch-brush (Sub-modules in `CMeshViewer`)

Since the patch-brush is relatively simpler, it is not written as an independent package. The essence is the `regionGrow()` function. An algorithm of this can be found in [3]. Notice that here a vector is used to store the candidate face list, instead of queue or priority-queue, because we need to update the whole list every time we delete a face from it with minimum cost.

Normal filtering is automatically supported and can be optionally turned off in the menu. It is a prepocessing step.

3. Conclusion

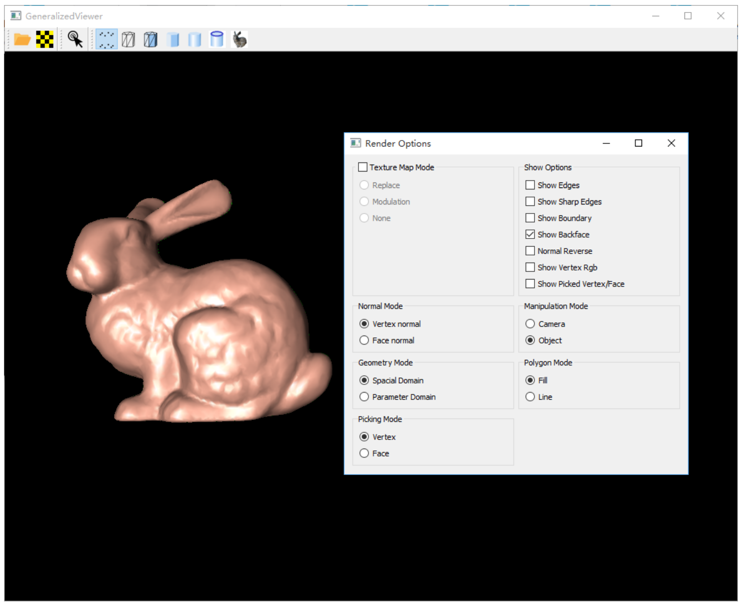
This project implements nearly all the functionalities in [1]. Surely, it has several weaknesses. First, it cannot deal with meshes with large size. Second, it cannot guarantee to work on a mesh with arbitrary topology. Though many tests have been done, some unknown mistakes may still exist inside the project. Anyway, it can be a good sample program to demonstrate basics of sketch-based mesh cutting, with a friendly user interface and satisfying performance on a variety of simple models.

4. Acknowledgement

1. The viewer is originally from David Gu as a framework in an assignment (not sure if there is an online version I can cite, I attached a copy). Some basic mesh operations are provided. However, to support the functions in this project, a great deal of new things are added, as can be compared. A picture of the original UI is shown in Figure 2.
2. SuiteSparse `CHOLMOD` [4] is used to compute the harmonic field.

5. Results

See following pages.



Texture Map Mode ☐
- Replace
- Modulation
- None

Show Options
- ☐ Show Edges
- ☐ Show Sharp Edges
- ☐ Show Boundary
- ☑ Show Backface
- ☐ Normal Reverse
- ☐ Show Vertex Rgb
- ☐ Show Picked Vertex/Face

Normal Mode: [✓ Vertex normal] Face normal

Manipulation Mode: Camera [✓ Object]

Geometry Mode: [✓ Spacial Domain] Parameter Domain

Polygon Mode: [✓ Fill] Line

Picking Mode: [✓ Vertex] Face

Figure 2: The original user interface from Gu

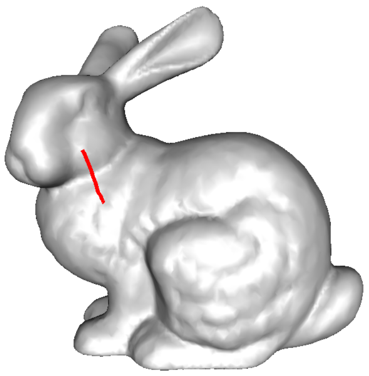
References

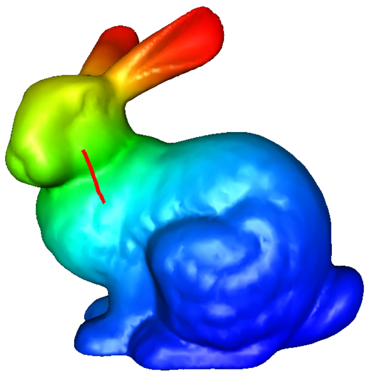
[1] Youyi Zheng and Chiew-Lan Tai. Mech decomposition with cross-boundary brushes. In *Computer Graphics Forum (In Proc. of Eurographics 2010)*, volume 29, 2010.

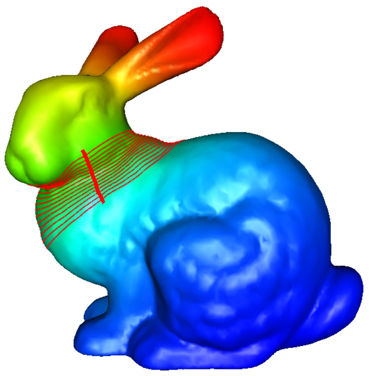
[2] Wikipedia. Marching squares — wikipedia, the free encyclopedia, 2015. [Online; accessed 4-December-2015].

[3] Huai-Yu Wu, Chunhong Pan, Jia Pan, Qing Yang, and Songde Ma. A sketch-based interactive framework for real-time mesh segmentation, 2007.

[4] Timothy A. Davis. User guide for cholmod: a sparse cholesky factorization and modification package, 2009.


(a) A stroke on a bunny


(b) The harmonic field


(c) 15 uniformly distributed isolines

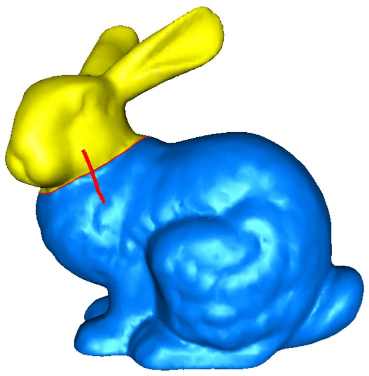

(d) The cutting result

Figure 3: The part-brush in slow motion

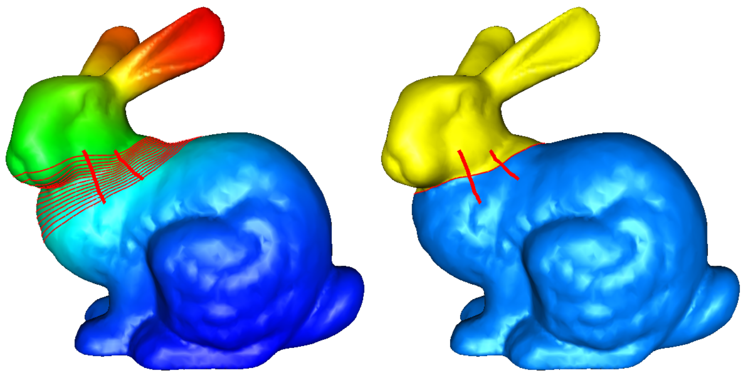


Figure 4: Part-brush: Dynamic updating

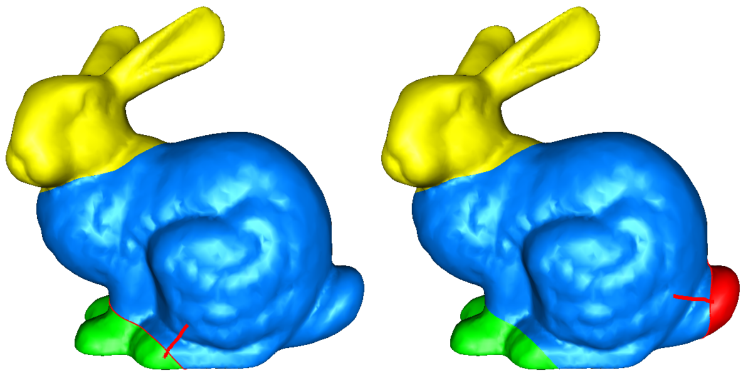


Figure 5: Part-brush: Cut multiple parts

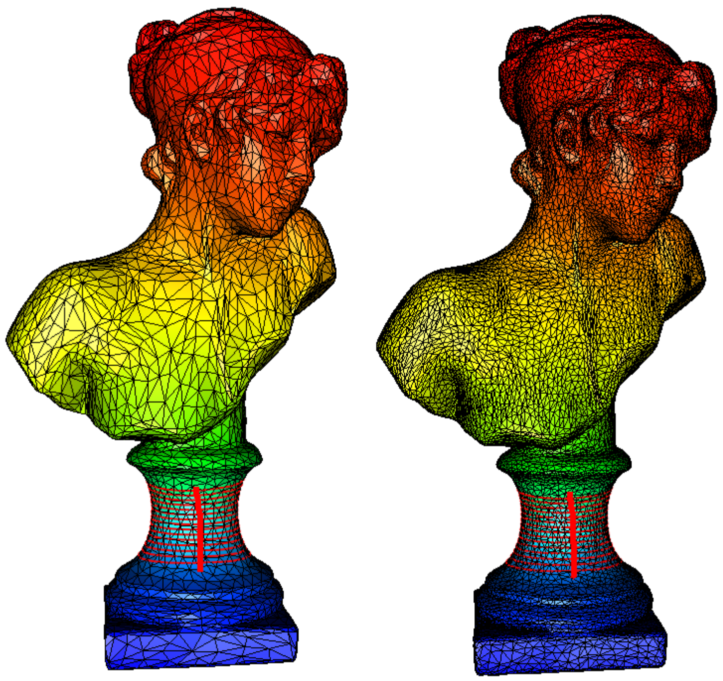


Figure 6: Part-brush: Insensitivity to tessellations. Left: 8k faces; Right 28k faces

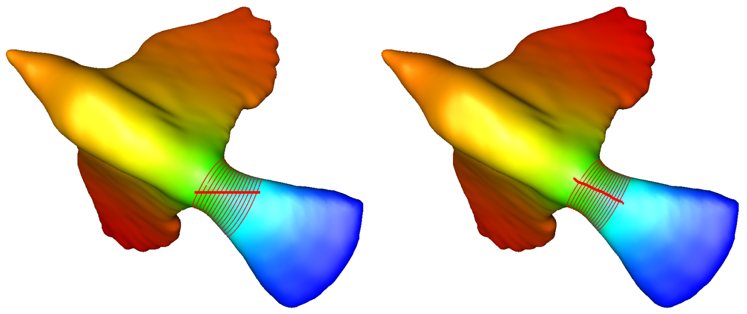
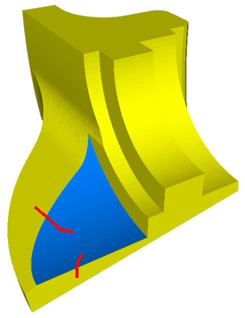
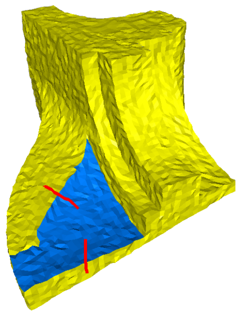


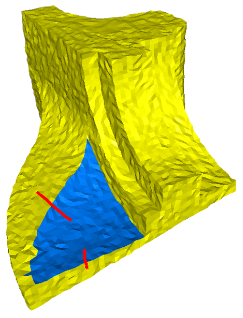
Figure 7: Part-brush: Insensitivity to stroke variations.



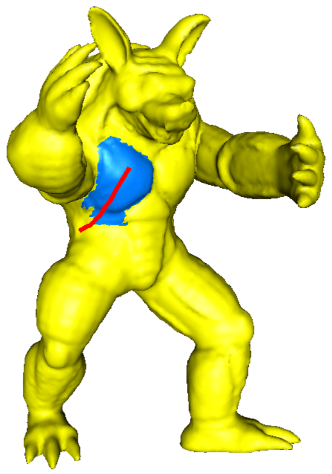
(a) No noise



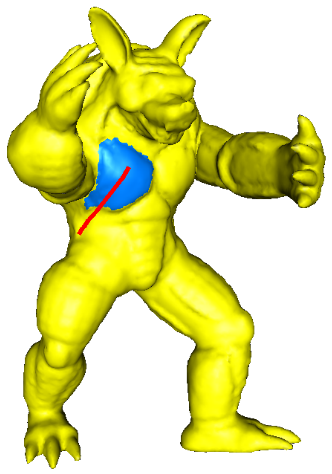
(b) With noise and without normal filtering



(c) With noise and with normal filtering



(d) Without normal filtering



(e) With normal filtering

Figure 8: Patch-brush: Robustness against noise and improvement, with normal filtering.