Report Number: WUCSE-2006-58

2006-01-01

A Thesis on Sketch-Based Techniques for Mesh Deformation and Editing

Raquel Bujans

The goal of this research is to develop new and more intuitive ways for editing a mesh from a static camera angle. I present two ways to edit a mesh via a simple sketching system. The first method is a gray-scale editor which allows the user to specify a fall off function for the region being deformed. The second method is a profile editor in which the user can re-sketch a mesh's profile. Lastly, the types of edits possible will be discussed and our results will be presented.

2006-58

A Thesis on Sketch-Based Techniques for Mesh Deformation and Editing

Authors: Raquel Bujans

Corresponding Author: rb3@cec.wustl.edu

Abstract: The goal of this research is to develop new and more intuitive ways for editing a mesh from a static camera angle. I present two ways to edit a mesh via a simple sketching system. The first method is a gray-scale editor which allows the user to specify a fall off function for the region being deformed. The second method is a profile editor in which the user can re-sketch a mesh's profile. Lastly, the types of edits possible will be discussed and our results will be presented.

Type of Report: Other

SEVER INSTITUTE OF TECHNOLOGY

MASTER OF SCIENCE DEGREE

THESIS ACCEPTANCE

(To be the first page of each copy of the thesis)

DATE: January 5, 2006

STUDENT'S NAME: Raquel A. Bujans

This student's thesis, entitled A Thesis on Sketch-Based Techniques for Mesh Deformation and Editing has been examined by the undersigned committee of three faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Master of Science.

APPROVAL: ______________________ Chairman

____

____

Short Title: Sketch-Based Mesh Editing Bujans, M.Sc. 2006

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

A THESIS ON SKETCH-BASED TECHNIQUES FOR MESH DEFORMATION AND EDITING

by

Raquel A. Bujans, B.S. Computer Science

Prepared under the direction of Professor Cindy Grimm

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

August, 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

ABSTRACT

A THESIS ON SKETCH-BASED TECHNIQUES FOR MESH DEFORMATION AND EDITING

by Raquel A. Bujans

ADVISOR: Professor Cindy Grimm

August, 2006
Saint Louis, Missouri

The goal of this research is to develop new and more intuitive ways for editing a mesh from a static camera angle. I present two ways to edit a mesh via a simple sketching system. The first method is a gray-scale editor which allows the user to specify a fall off function for the region being deformed. The second method is a profile editor in which the user can re-sketch a mesh's profile. Lastly, the types of edits possible will be discussed and our results will be presented.

To my family, friends, and all the wonderful people at Washington University

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis describes two techniques for performing mesh editing. In our system, shown in figure 1.1, artists first load a 3D mesh into the program. Artists can position the mesh as they likes by clicking a point on the mesh and dragging, or by using the controls available on the left-hand side of the screen. Once artists have identified an area of the mesh that they would like to edit, they can alter the mesh using either of the two editors we have developed.

Mesh editing can be a very tedious task. In the worst case scenario, artists may be forced to edit a mesh vertex by vertex, if they are trying to introduce fine scale changes in a detailed mesh. Such a task quickly becomes infeasible as models increase in complexity. Some techniques have been developed that allow a user to specify larger scale deformations that can be applied across a region of vertices simultaneously ([21], [15]). For example, there are commonly used tools that allow an artist to scale, translate, or rotate a section of a mesh. Such tools simplify the overall task, but the artist may have to sacrifice some fine scale detail in exchange for ease of use.

If a complex deformation is desired the mesh may also have to be rotated around and edited from several camera angles. Not all viewpoints can be seen at

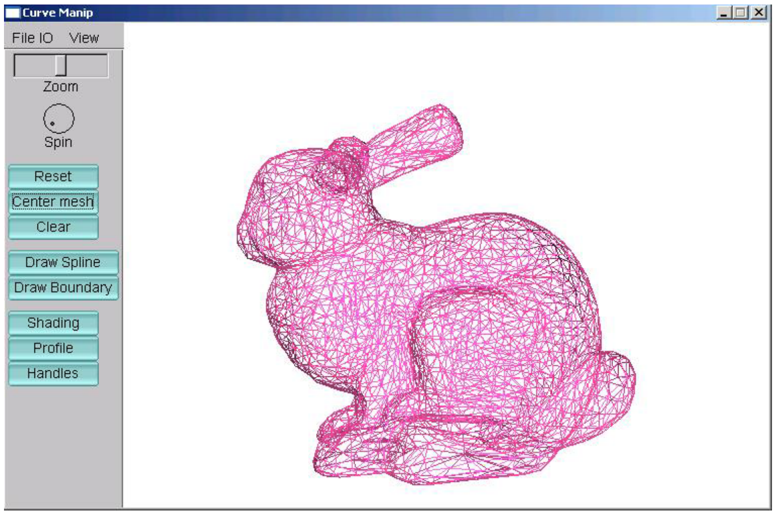


Figure 1.1: Editing System Layout.

once, which means an edit made from one angle may look completely unacceptable from another. Any attempts to reconcile the two viewpoints may introduce further unwanted visual artifacts.

The techniques presented in this thesis aim to resolve these issues by allowing an artist complete artistic control from one viewpoint. Any edits made should look good from all angles, regardless of initial orientation. The techniques work from sketch-like gestures as input, allowing the artist to specify their artistic "intent" without having to be overly precise. The primary contributions of this thesis are the unique user interfaces created for the two sketch-based editors, as well as the application of corrected matrix linear interpolation to mesh deformations. Our work ties in many existing algorithms into one cohesive whole in a manner that has never before been attempted.

The thesis is organized as follows. Previous work relevant to sketch-based mesh editing is discussed in Chapter 2, followed by essential background information in Chapter 3. Chapter 4 presents an overview of the system, while Chapters 5, 6, and 7 discuss the details of the shading, profile, and direct editor, respectively. Chapter

8 discusses how the mesh deformations are performed. Finally, in Chapter 9 the types of edits our system is capable of are explored, and in Chapter 10 we discussed improvements that could be made to our system.

Chapter 2

Previous Work

2.1 Sketch-based User Interfaces

Sketch-based user interaction is currently an active field of research. The field seeks to develop applications for a wide variety of purposes, from aiding in automobile design to rapidly drawing deformable cartoons. Sketch-based user interfaces are appealing because they more closely approach the movements of a traditional artist.

Early sketch-based deformation systems took their inspiration from traditional sculpture techniques [21]. Singh and Fiume introduced wires (see figure 2.1)as an easy way of editing 3D meshes. In their system, an artist sketches wires on the mesh which are then bound to the geometry. By manipulating the wires, the artist deforms the geometry. Singh and Fiume's technique combines the level of control available in free-form deformations with the ease of use in sketch-based UIs. Our research aims to build directly off this sort of UI approach, but uses a different underlying deformation technique.

Other techniques have also aimed to tackle the problem of editing from multiple camera angles. Cohen et. al. developed a system [4] in which an artist could sketch

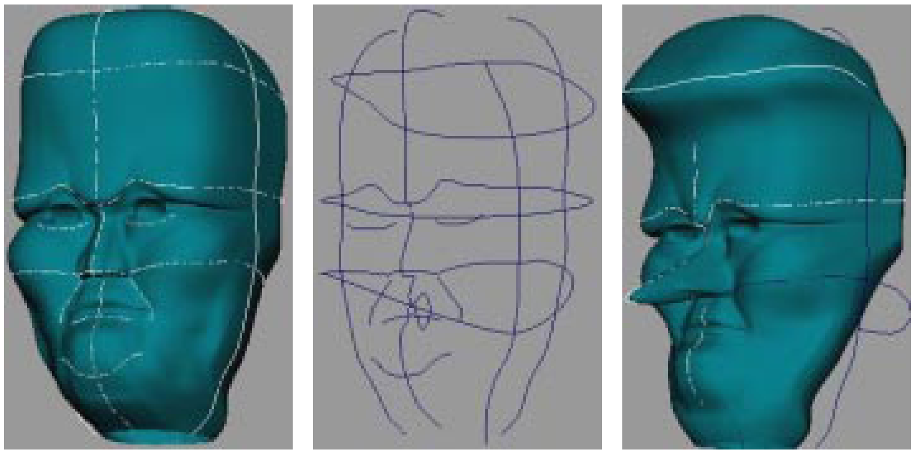


Figure 2.1: Singh and Fiume's Wires, here used to edit the nose of the mesh

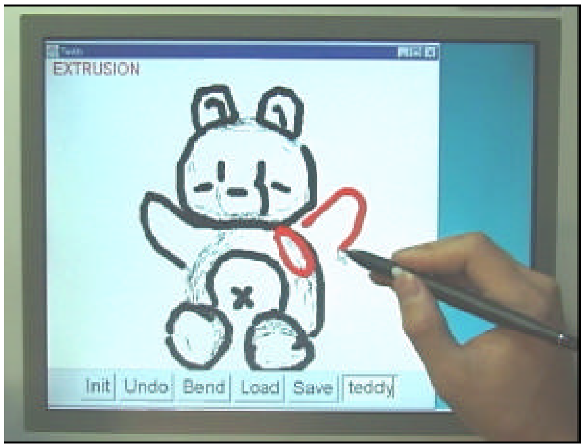


Figure 2.2: Igarashi's Teddy

a 3D curve in space, followed by sketching its projected shadow on a plane. Using the extra information provided by the shadow, their system could infer the exact placement of the curve in space. If users edit the curve or the curve's shadow, they redefine the curve's 3D placement. The goal of this work was to rapidly create approximately correct curves, which an artist can later refine in another program. Our system implements an idea similar to the plane-sketching idea in the profile editor. We project the mesh's profile along a curve onto a plane and let the artist redefine it.

Sketch-based UIs are useful for a wide range of applications, but are particularly useful for design. The ability to rapidly sketch a prototype can save a large amount of money and time. An artist can use a sketch-based UI to create 3D models from scratch, as in the Igarashi's Teddy program [11]. In Teddy (see figure 2.2), an artist draws a 2D silhouette and the program builds a 3D model from it. Emphasis is again put on rapid prototyping without having to bog the artist down with too many details. However, Teddy is only capable of producing round, blobby shaped models. The level of control a user has over the type of shapes created is limited. In a similar vain, [22] presents a system for using a sketch-based UI to rapidly prototype automobile designs, [20] present a system for sketching architectural designs, and [19] present a technique for building models from 3D sketches scanned in with camera phones.

Another use for sketching interfaces is in animation. An artist may want to specify that a portion of a mesh be deformed over time. Kho and Garland addressed this need in [15]. Their work builds off the wires idea. They create a system in which a user sketches a reference curve along a part of the mesh. Then the user sketches a new curve along the new desired path for the reference curve. By interpolating between the original and deformed positions, a smooth animation can be attained.

2.2 Deformation Techniques

Interpolating between deformations is also an active area of research. Our goal is to perform a deformation that acts “correctly”, i.e. the resulting mesh looks as we would expect it to. Translating, scaling, or rotating one part of the mesh should not cause unrelated parts of the mesh to suddenly grow or shrink. Furthermore, none of these operations should cause a mesh to lose volume. Several papers have tried to address these issues. While these works address topics worthy of further research, our work does not attempt to simulate any realistic physical properties nor preserve volume.

In Singh's wires, axial deformations were used. In such deformation, a reference wire is specified around which a fall off function is set. The fall-off function specifies by what percentage the area around the reference curve will deform. The percentage grows smaller as distance increases from the curve. Deformations made using wires generally produce pleasing results. However, successive transformations are not commutative and may accumulate error. If we are trying to specify deformations to interpolate between, we may run into problems. For example, if a deformation is specified, one would expect that applying one-half of the total deformation twice would produce the same effect as applying the entire deformation once. However this is not the case. Our system presents editing techniques similar to wires but with the corrected interpolation presented by Alexa [1].

Another way to head off deformation problems is by representing the mesh in a different type of coordinate system. Lipman et. al. present a type of rotation-invariant coordinates [17]. In their paper they discuss a way to represent vertices on the mesh by encoding the relative differences between them. Therefore no matter what rigid-body deformation is performed, the mesh should be able to preserve local topology correctly. Volumetric graph laplacian coordinates [24] could also be used, as could mean value coordinates [14].

Chapter 3

Background

This chapter provides the background information necessary for this thesis. Here we present the fundamental concepts and general terminology used in our research.

3.1 Meshes

A *mesh* is a list of vertices, and connectivity information for faces and edges. The mesh can be rendered in wireframe mode or shaded with lighting applied. An example of a typical mesh is depicted in figure 3.1. 3D meshes used in this research were taken from a library of meshes available in the Media and Machines lab. The meshes we

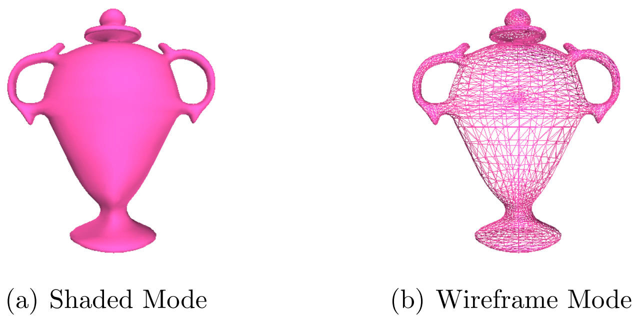

(a) Shaded Mode (b) Wireframe Mode

Figure 3.1: An example mesh

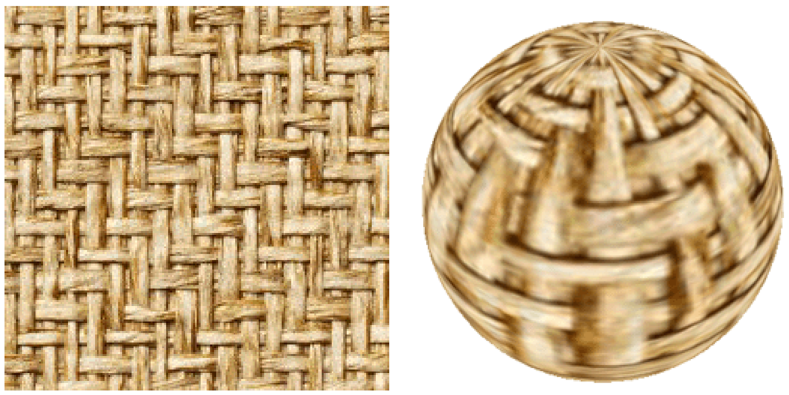

(a) Texture image (b) Image mapped onto a sphere

Figure 3.2: Example of a texture image mapped onto geometry

tested are commonly used through Computer Graphics literature. They have either been scanned in from real world models or created by hand using modeling software.

3.2 Splines

We represent curves drawn on the mesh as *splines*. A spline is a piecewise polynomial representation of a curve. Any point along the spline can easily be computed. We build splines using the mouse generated input points as control points. The spline curve always falls within the convex hull of its control points. Specifically, we use non-uniform rational b-splines (NURBS) to represent curves in our system. We can evaluate any point on the spline by calculating $C(t) = b_i(t)g_i$, where b_i is a piecewise polynomial and g_i is a control point, and t ranges between 0 and 1. Splines allow artists fine control over their drawing operations and have a low computation cost.

3.3 Texture Mapping

Texture mapping is the process of mapping a 2D image, or *texture map*, onto a 3D surface. See figure 3.3 for an example. The texture map is generally a rectangular image whose width and height are a power of 2. Each pixel in the texture, or *texel*, will map to at least one pixel on the surface. We refer to any texel's coordinates in the image as (u, v), where u and v range between 0 and 1.

Texture mapping is done in two steps. First, *texture coordinates* must be set. We say a mesh point is assigned a *texture coordinate* when a point on the image (u, v) is constrained to lie at a specific location on the mesh. Commonly, a texture coordinate will be set for each vertex in the mesh being painted by the image. Second, for other pixels without a specific texture coordinate, final color is interpolated from one or more texels in the image.

In our research, we use texture mapping to map a gray scale image onto the area of the mesh being deformed. The textured image conveys information that otherwise would have had to be specified through the surface properties of the mesh's vertices. Although we do not use textures to represent very high levels of detail, we find it is an easy, mesh-independent way of specifying detail at a resolution higher than a mesh's default polygon count.

3.4 Mesh Modification

The way a mesh's surface is parameterized greatly affects its ability to approximate the modeled surface, its ability to render aesthetically pleasing textures, and the amount of mesh data that needs to be stored. A well parameterized mesh also lends itself to more numerically stable calculations. The goal is keep enough vertices, edges, and faces in areas of high detail such that the high-frequency information in the model is captured. In areas that are almost planar however, we don't need to capture much information and can get away with storing less data. As the user edits a mesh, an area of high curvature may be introduced, which will in turn need more detail in

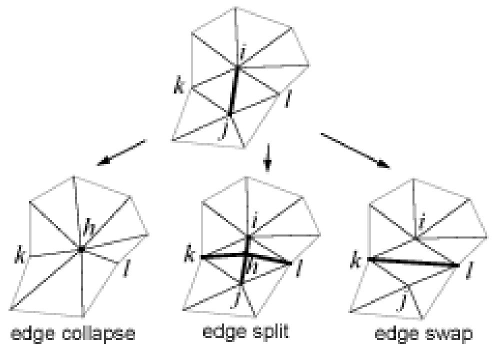
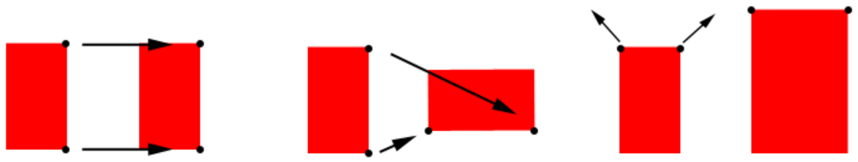


Figure 3.3: Mesh simplification operations

order to accurately capture the curve. We dynamically retriangulate the mesh after each edit, so that the number of polygons in a region scale proportionally with the region's curvature. If, after editing, an area becomes mostly planar, we can reduce the polygon count.

In a good triangular mesh parametrization, we should strive to have all triangles be roughly the same size and equilateral. We can do so by performing three types of operations on the mesh: edge swapping, edge splitting, and edge collapsing. Our algorithm for evaluating these operations is based directly off a commonly established technique ([10], [16]). Figure 3.3 shows examples of each type of operation. Before describing the operations in detail, let us formally define some terms. An *edge* in the mesh is a connection between two vertices $e = (u, v)$. A *face* is a collection of vertices connected by edges, and has at least three vertices, $f = (u, v, w)$. When deciding whether to perform a mesh modification operation, we only consider a small subarea on the mesh, called the *neighborhood*.

An *edge collapse* happens when we take an edge (v_i, v_j) and replace it with a vertex v_r. An *edge split* is when we take one edge and form two new edges from it, each one with half the length of the original. New edges are then drawn between the neighboring vertices and v_r. In an *edge swap*, we take an edge between two vertices



(a) Translation (b) Rotation (c) Scale

Figure 3.4: Geometrical point transformations

(v_i, v_j) and replace it with an edge between two other vertices from the same faces (such as (v_k, v_l)). Our system often needs to increase the resolution of our mesh, and hence we mostly lean towards performing edge splits. However computation costs can be lower in mostly planar regions by performing edge collapses. Edge swaps are performed in any area of the mesh deemed appropriate, regardless of curvature. The implementation details for these operations can be found in [10] and [16].

3.5 Geometrical Transformations

In order to deform a mesh, we perform *geometrical transformations* on the mesh's vertices. Since edges and faces are defined in terms of vertices, if we transform the vertices then the entire mesh is transformed. There are three types of transformations performed in our research - translation, rotation, and scaling. All three operations can be done using matrix algebra. If a 3D point is represented as 3D column vector, we can multiply it by a square matrix representing the transformation in order to find the point's new location.

In order to move vertices around properly, we must scale, rotate, and translate them in the proper order. We can combine the scaling, rotation, and translation matrices into one matrix in order to save on computation. However, if they are combined in the wrong order, we may end up making an unintended transformation.

Matrix multiplication is not commutative and thus the order in which operations are applied is important. For example, we know that rotation and scaling must occur about the origin. Hence any time we wish to rotate or scale a mesh, we first translate it so that it is centered at the origin, perform any rotations and scales, and then translate back an equivalent distance. Lets say we would like to deform points $p_1, p_2, \ldots, p_n$. We first translate so that the points are centered about the origin. We then perform any desired rotations, followed by any desired scales, followed by any new translations on the points. Lastly, we translate the points back an equal and opposite amount to counter the centering operation. Mathematically, we state this as:

$$M = T(-c) * T * S * R * T(c)$$

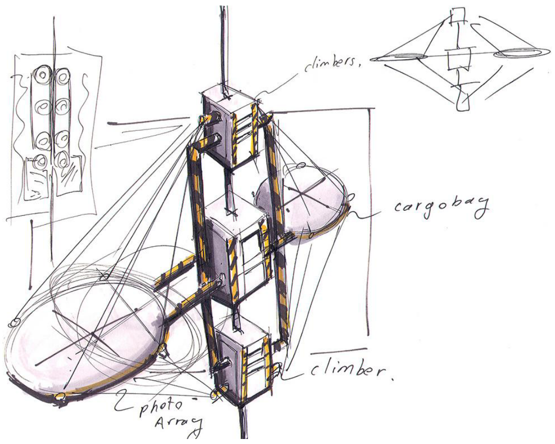
where M is the combined matrix, $T(c)$ is matrix that translates the set of points to be around the origin, R is the rotation matrix, S is the scale matrix, and T is the new translation matrix.

3.6 Sketching

Sketching is a technique employed by artists when they are trying to quickly express an idea (see figure 3.5). An artist draws the most important elements of an image in a rough, approximate manner. Sketching gives someone the immediate impression of an idea, without the need to specify details. It is quick, easy to do, and requires few resources. If a sketch does not come out as desired, the artist can change it by drawing on top of existing lines. In our research, we will use the sketching technique in both of our editors. The idea is to allow an artist as much control as possible without requiring him to make exactly precise movements.

(a) Van Gogh

(b) Da Vinci



(c) Nyein Aung, Liftport Group

Figure 3.5: Sketch Drawings

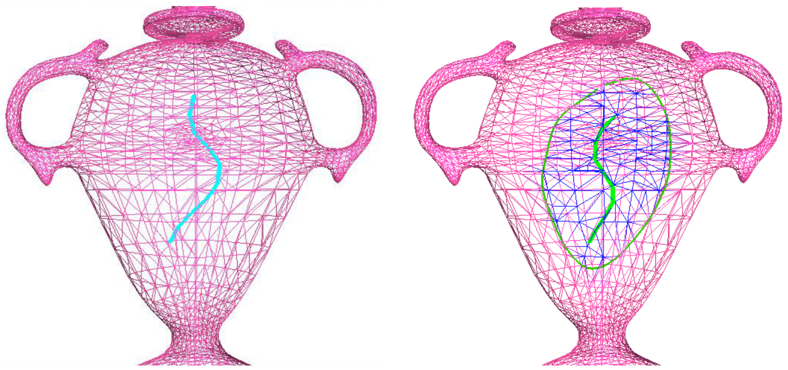
Chapter 4

System Overview

In this chapter, we describe the basic fundamentals of using our system. Prior to using either of the editors, users must decide what area of the mesh they would like to edit, and from what vantage point they want to view the area. There are camera controls, curve drawing tools, and deformation handles available to the user.

When our system starts up, the camera is always located in a default position. When we load a mesh, the camera automatically centers itself on it. A user may want to edit part of the mesh that is not directly in view, or a part that may be viewed better from a different angle. Hence, manipulating the camera easily is an important prerequisite for mesh editing. Our system offers users several different ways to perform the same type of camera change. Some users may prefer using keyboard shortcuts over the virtual trackball, while others may prefer using a combination of both. Having multiple ways to perform the same type of edit allows users to pick which ones they find the most speedy and intuitive. A description of the specific camera controls implemented in our system is provided in [Ap. A].

Before opening either of the editors, users must first specify what part of the mesh they would like to edit. We will refer to the area of the mesh that is selected



(a) Reference curve (b) Boundary curve (in green)

Figure 4.1: Reference curve

as the *region of interest* (ROI). Before either editor can be used, users must set the reference and boundary curves. The *reference* (figure 4.1(a)) curve forms the backbone of the feature we are trying to create or edit. The reference curve will lie at the center of any deformations that occur later on. It also helps define the general shape of the feature. After marking the reference curve, users must decide how much of the mesh around it to influence. For example, we may only want to extrude a small bump on the mesh or we may want to deform a large section. The area to be deformed is sectioned off with a *boundary* curve (figure 4.1(b). The reference curve, together with the boundary curve, completely define the ROI. Once the reference and boundary curves are set, no more input is required from the user. Users can choose to refine their input by *re-sketching* either of the curves. Re-sketching modifies a pre-existing curve by blending it with a newly drawn curve. Users can choose to replace the entire curve or just redraw a portion of it.

To perform deformations, users click on the reference curve and drag it around. More detail on how exactly deformations are performed will be provided in chapter 8. Vertices outside the ROI do not get affected by any deformations, while vertices inside the ROI are affected at varying levels. When the reference curve is deformed, we want the vertices that lay along it to deform completely, but we want the vertices laying along the boundary curve to remain stationary. All vertices in between move at some percentage of the total deformation. Exactly how much they deform is determined by using the shading editor and the deformation handles.

4.1 Implementation Details for Specifying a Region of Interest

In order to specify a region for editing, a user must first draw a reference curve. The new reference curve is represented as a spline. We build the curve by using the input points generated from the mouse as control points. The stroke is parameterized from 0 to 1, where a parameter value of 0 gives the first input point, and a value of 1 gives the last input point. Any value in between 0 and 1 can be used to find a point lying along the curve.

The stroke is then filtered, so that no control points on the curve are located too close together. To filter the curve, we perform the following steps. First, we take the mouse generated input points and project them back into screen space. We discard any points that are too close together in screen space. For our purposes, a thresholding distance of 10 pixels gives a nicely spaced distribution. Second, a NURBS curve is fitted to the remaining points. Third, the spline is resampled so that control points are spread out evenly over the length of the curve (see figure 4.2). We sample the curve at a high enough rate for there to be a control point every few pixels. The resulting spline is what we save as the final reference curve. Mesh faces located underneath the spline's control points are subdivided so that the spline's control points lie directly on mesh vertices.

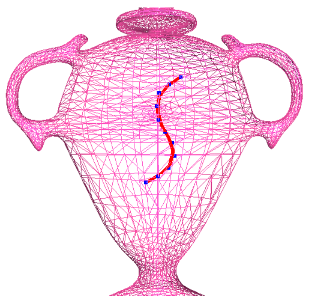


Figure 4.2: Evenly spaced control points

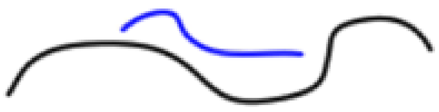
The boundary curve is represented in a similar manner. However the boundary is always a closed curve. Hence we automatically set the beginning and ending control points on the boundary curve to be the same point. It is then thresholded and resampled, just like the reference curve.

4.2 Implementation Details for Curve Re-Sketching

To re-sketch a curve, the user simply draws another curve over the old one. Depending on the new curve's distance to the old one, part or all of the old curve is replaced. If only part of the old curve is replaced, we want to blend smoothly between the old parts and new parts of the curve.

Our algorithm used to implement re-sketching (see figure 4.3) , based off of [3]'s technique, works as follows. Let us refer to the original curve as f and the re-sketched curve as m. When m is first drawn, it is input as series of points generated from the mouse's movements. The points are used to build a NURBS spline, and the NURBS spline is then resampled evenly. After m and f are merged, we are left with a final re-sketched curve, called r. How "smooth" a transition there is from f to m is controlled by a "smoothing" coefficient, k. Although Baudel's work states that a value of $k = 3$ works best, in our research best results were achieved when $k = 6$. There are three main parts to the curve merging/re-sketching process; one

(a) Originally sketched curve


(b) User draws amended version of curve area

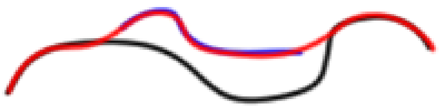

(c) Resulting merged curve

Figure 4.3: The resketching process

must analyze the starting transition from f to m, the main body of m, and the final transition from m back to f. Throughout the following explanation, refer to figure 4.4 for a visual illustration.

The beginning and ending transitions are identical to each other, but work in reversed directions. Let us define point S as the point on f that is closest to $m(0)$, the starting point of m, and E as the point on f that is closest to $m(1)$, the ending point of m. If $S < E$, then we need to calculate an initial blending point, $S' = S - k * d(f(S), m(0))$, from which we will begin blending between f and m. $d(f(S), m(0))$ is a term that measures the distance between the points $f(S)$ and $m(0)$. The location of $f(S)$, and hence the accuracy of the distance metric, depends on how finely divided into intervals the spline is. If we evaluate our NURBS splines at smaller intervals, we will achieve more accurate results. If $S' < 0$, that means that m extends the old curve; we can replace the beginning of f with the beginning of m, and not create a smooth transition between the two (see figure 4.4(b)). In other words, r will begin at $m(0)$. Otherwise, we must define a transition spline m_s going from S' to $m(0)$. To calculate m_s, we create a hermite spline with two points, S' and $m(0)$. The tangent at S' is calculated by finding the control point on f that precedes

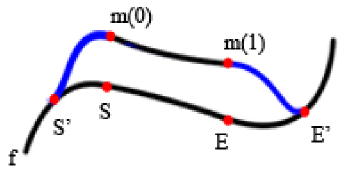
$S^{'}$ and subtracting its position from $S^{'}$. The tangent at $m(0)$ is found similarly by subtracting $m(0)$ from m's following control point. Then, for all the control points on f that fall between $S^{'}$ and S, we create new corresponding points on $r(t)$ as follows:

```
for each control point $t$ between $S^{'}$ and $S$:
$c = f((t - S^{'})/(S - S^{'}))$
$r(t) = (1 - c) * f(t) + c * m_s((t - S^{'})/(S - S^{'}))$
```

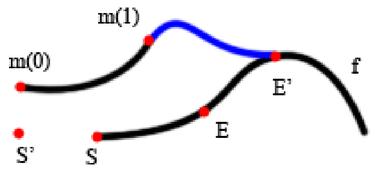
The same procedure is used to create an ending transition from m to f. We first calculate $E^{'} = E + k * d(f(E), m(1))$. If $E^{'} > 1$, we know that m is an extending stroke, and no transition back to f is necessary (see figure 4.4(c)). Instead, r will end at $m(1)$. Otherwise, we build a transition spline m_e that goes from the last point on m, referred to as $m(1)$, to $f(E^{'})$. Just like m_s, we build a hermite spline and filter it. For each control point on f between E and $E^{'}$, we build a corresponding point on $r(t)$ as follows:

```
for each control point $t$ between $S^{'}$ and $S$:
$c = f((t - E^{'})/(E - E^{'}))$
$r(t) = (1 - c) * f(t) + c * m_s((t - E^{'})/(E - E^{'}))$
```

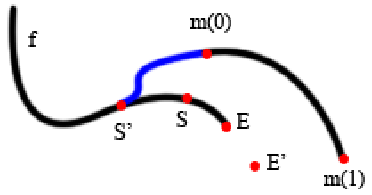
For all control points t along f between S and E, we create a point on r located at the closest point on m to $f(t)$. After performing these steps, we have completely integrated the new sketching stroke. A user has the freedom to re-sketch any curve drawn in our system.



(a) Start and end transitions



(b) No starting transition



(c) No ending transition

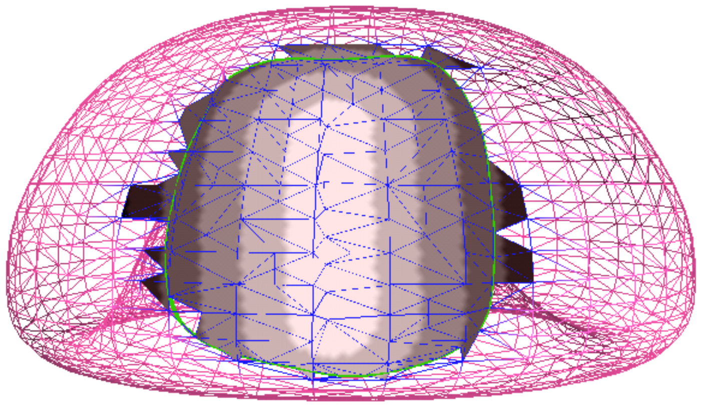
Figure 4.4: Re-sketching the curve

Chapter 5

Shading Editor

The shading editor (figure 5.1) is useful because it allows users control over how different portions of the mesh will deform. Alone, a reference and boundary curve are not enough to specify how we want the ROI to deform. The same reference and boundary curves can be used to specify completely different types of deformations (see figure 5.2). The shading editor is used to specify how much the vertices within the ROI should be deformed.

The editor opens in its own viewing window and lets the user view the ROI as a flattened region. Once users mark the area of the mesh they want to edit, the system projects a gray scale texture over the area. The system creates a square texture and then deforms it so that it lies over the entire region. In order to make the application of the texture to the mesh easier, we first flatten the selected area of the mesh into a disc. Then, users must decide how they want the gray levels to be set. Ideally, in a fully sketch-based system, our system would have a user completely draw the gray scale texture from scratch. This would be a very tedious process however, so instead we present the user with a default texture that can be edited. Users can edit the texture until they are satisfied with it, and then we copy the resulting texture


(a) Main window

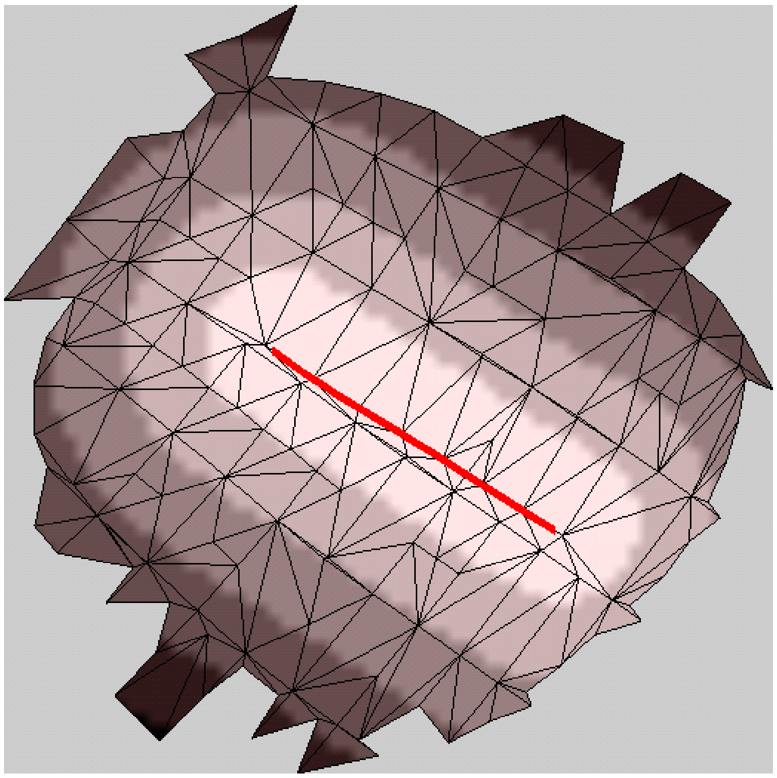
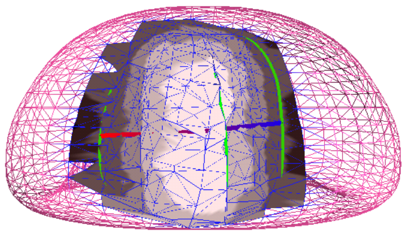
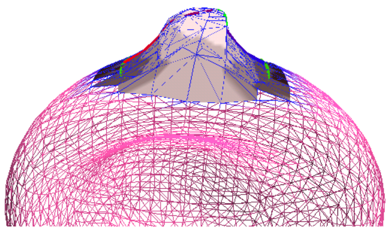

(b) Shading Editor

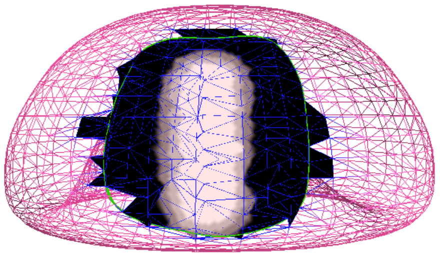
Figure 5.1: Layout of the shading editor



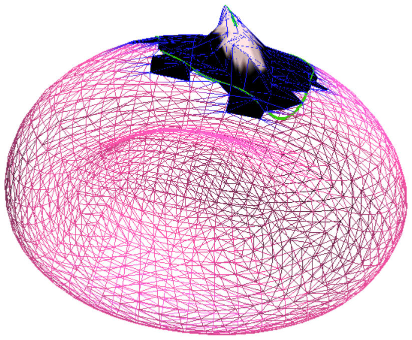
(a) Graduated gray scale fall-off



(b) side view



(c) Sharp gray scale fall-off



(d) side view

Figure 5.2: Different types of surfaces can be made using the same reference and boundary curves

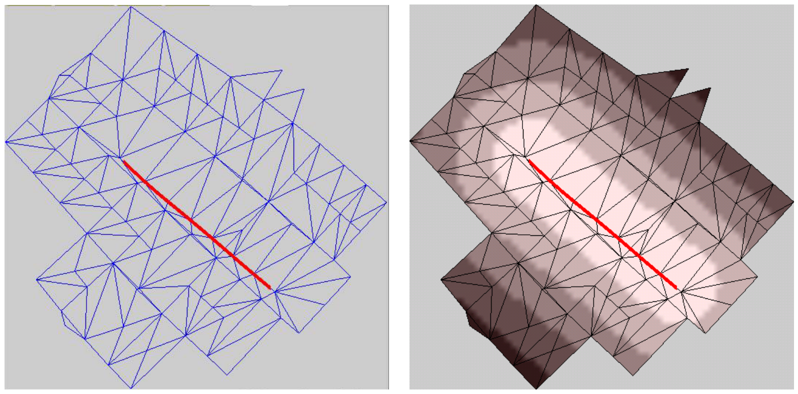
back onto the original mesh. The entire texture may be edited all at once, or if edits sectioned off with the area selection tool, changes can be limited to a specific region.

The gray level tells the user by how much a vertex will deform. For example, a white area of the texture indicates that vertices located in the area will move 100 percent with any specified deformation, whereas a black region indicates that vertices in that area will remain stationary. Gray values in between the two extremes signify varying levels of movement.

5.1 Computing the Region of Interest

In order to apply a texture to the ROI, we transform the selected area of the mesh into a form that is amenable to texture mapping. But first, we must compute what part of the mesh is included in the ROI. After the user has selected a part of the mesh to edit, we see what face lies underneath the point that lies halfway along the reference curve. From this face, we grow outwards, checking each neighboring face to see if it is also in the ROI. We continue doing this until we can no longer find any neighboring faces that are still within the ROI. Then we add in one extra ring of faces, just to make sure we have the entire area within the ROI accounted for. Only vertices strictly within the area marked off by the boundary spline will be affected by mesh deformations.

Having computed the ROI, we create a copy of the area and "flatten" it out into a disc. The vertices are rescaled so that they are all located within the range of -1 to 1 on the x and y axes. All vertices have a z coordinate of 0 after the flattening operation. Implementation details on how the rescaling is performed are provided in appendix [Ap. B].



(a) Flat disc (b) With texture applied

Figure 5.3: ROI as displayed in the shading editor

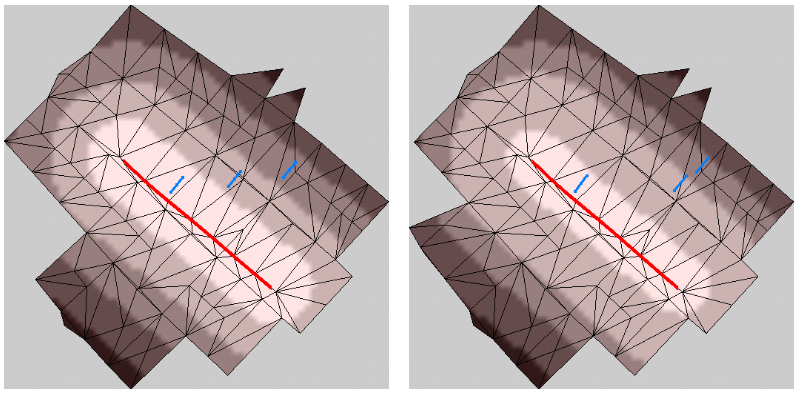
5.2 Generating a Texture

The default texture is generated based on a vertex's distance to the closest spline point on reference curve R. For each vertex in the ROI, we set a corresponding texture coordinate, (u, v). Trying to edit a continuous gray scale texture would not be very useful, however. Instead we discretize the distances into levels for display purposes. The number of levels is user controllable, but we find that a default number of 5 levels works well. The gray levels vary from white to black in even intervals. Figure 5.3(b) shows what the flattened mesh looks like with the default texture applied.

Although the texture is discretized into gray levels for viewing, the function it represents is still continuous. Within each level, vertices will still linearly interpolate between their level's boundaries in order to determine how much they should deform.

5.3 Editing the Texture Globally

Our system allows a user complete control over the texture's appearance. To edit the texture, the user places a handle on a particular layer (as in figure 5.4). A handle anchors the layer it is placed on. If only one handle is placed, then all layers will



(a) Handles (b) After dragging

Figure 5.4: Handles used for texture editing

change when it is dragged. However, if more than one handle is placed, only the layers between handles will move. For example, if three handles are placed and the middle one is dragged, only the layers between the outer handles will be altered. If an outer handle is dragged, any layers between its neighboring handle and the outermost layer will be changed. If the handles are dragged sufficiently inwards, it is possible to make a layer disappear. Not to worry - dragging the layers outwards again will make the missing layer reappear. However, no amount of dragging outwards can introduce a new layer that was not already pre-existing.

5.4 Editing the Texture Locally

Handle dragging affects the entire texture, but a user may only want to edit a section instead. For this case we introduce the area selection tool (see figure 5.5). After sectioning off an area, the user can place and drag handles, and only this area will be affected. While an area is selected, the user can also change the number of gray levels and have it only affect that region.

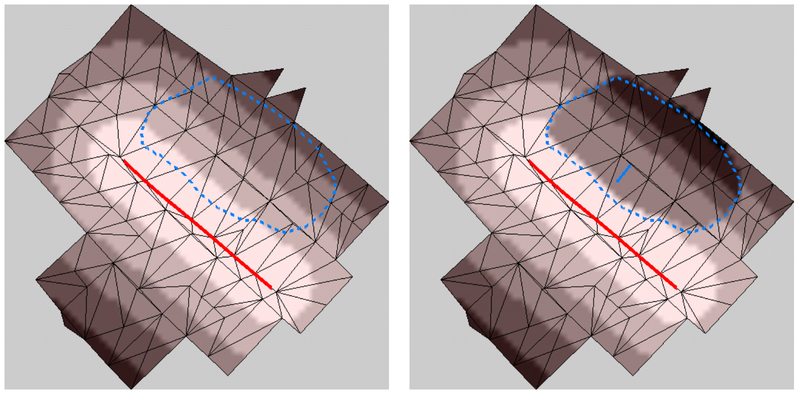

(a) Area Selection (b) Dragging within selected area

Figure 5.5: Editing within a selected region

5.4.1 Area Selection Algorithm

The area selection algorithm keeps track of what pixels are in the selected region. It works the same as a scan-line polygon fill algorithm mentioned in [6]. The algorithm is robust enough to handle an area selection that self-intersects. It takes as input points generated from the mouse, and outputs a set of pixels p that are strictly within the selected area. The basic outline of the algorithm is as follows. From the set of input pixels, we draw edges connecting them together. The edges will form the boundary of the area selected. Our algorithm scans lines from left to right, bottom to top, seeking to identify exactly what pixels within this area have been chosen by the user. We sort the edges according to their x coordinate. Finally, for each scan-line, we add to p any pixels between edges on that line, as long as they are considered 'inside' the region.

To start, we check each scan-line for intersections against edges. Since not all edges will intersect a given scan-line, we take advantage of *edge coherence*. Edge coherence states that most of the edges that intersect a particular scan line are likely to intersect the next scan line as well. Hence, we can update our intersection information based on the previous scan line, instead of calculating it from scratch each time. Using

Table 5.1: Area Selection Data Structures

Edge	(y_{max}, $p1$, $p2$, x, m) connecting 2 pixels
$p1$	Pixel
$p2$	Pixel
y_{max}	Max(p1.y, p2.y)
x	X coordinate of intersection between edge and scan line
m	Edge slope

Edge Table	list of (y_{min}, *edge*) pairs
y_{min}	Min(p1.y, p2.y)
edge	(y_{max}, x, m)

this property, we create an *active edge table (AET)* which keeps track of what edges the current scan line intersects. We also keep a complete list of all edges in a *global edge table (GET)*. A complete description of the data structures used by this algorithm is presented in table 5.1. All edges listed in a table are sorted in order of increasing y coordinates.

Maintaining the AET is done by running a scan-line algorithm. We look at the GET and find the minimum and maximum y coordinates. Our scan-line algorithm will examine all pixel rows in between and including these two. Let y be the row we are currently on. Starting with $y = y_{min}$, we scan the pixels from left to right. Although initially the AET starts out empty, it gets populated with edges as we scan a row. As we scan y, we move any edges with $y_{min} = y$ from the GET to the AET, and sort them on their x coordinate. Any edges already in AET that have a $y_{max} = y$ are removed. Pixels between pairs of x coordinates are then added to p. Finally, we update the x coordinates of non-vertical edges in AET and move to row $y = y + 1$.

As we move from one scan line to the next, for all edges in the AET, we update the intersection x coordinate by setting $x_{y+1} = x_y + \frac{1}{m}$, where m is the slope of the edge. m is calculated as $m = (y_{max} - y_{min})/(x_{max} - x_{min})$. Clearly this update rule can result in fractional x coordinates. We handle this by storing x as a double, and rounding up to get an integer pixel coordinate when necessary.

Whether a pixel is considered 'inside' the selected region is determined by the *odd-parity rule.* The odd-parity rule states a parity bit can be used to determine belonging; if the bit is even, the pixel is considered inside the region, otherwise it is outside. Each time a scan line crosses an edge, the bit is flipped. When we move to a new scan line, the bit is initially set to 'inside', since we start on the intersection between an edge and a scan line.

While the algorithm so far works fairly well, there are a few special cases worth noting. As we update an edge's x coordinate, we will be left with either an integer or fractional intersection. If there is an edge has an integer intersection, it is handled differently depending on what side of the scan line it bounds. If it falls on the leftmost side, we consider it to be inside the area. However if it is the rightmost, we consider it to be outside. Furthermore, if the intersection at an integer pixel coordinate occurs on a shared vertex, we have to handle it slightly differently. In this case, the y_{min} pixel of an edge affects the parity bit, but not y_{max} pixel. In other words, a y_{max} pixel only gets included if it doubles as the y_{min} for a neighboring edge. Lastly, horizontal edges are treated specially depending on whether they form a bottom or top row. Bottom horizontal edges are drawn, but top ones are not.

5.5 Applying Texture onto Original Mesh

Once users have edited the gray scale texture satisfactorily, the new texture will be mapped back onto the original mesh. Each vertex in the ROI determines how much it is able to deform according to what gray level was associated with it. Texture data is copied from the disc over to the original mesh. No new texture coordinates or colors need to be calculated.

Chapter 6

Profile Editor

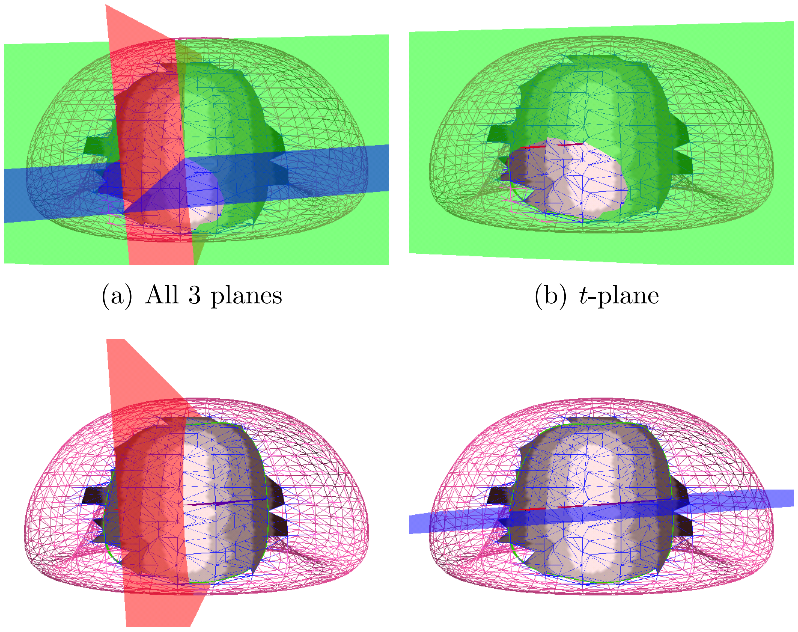
The profile editor is useful for re-sketching how a cross section of the ROI looks. The profile editor opens in a separate window and presents the user with a profile going horizontally across the center of the ROI. On the original mesh, we draw three planes based off of the reference curve that define a local coordinate system for the profile. The planes can be thought of as approximating a *frenet frame.* A frenet frame is defined by three orthonormal vectors along a point on a 3D curve in space. One vector is the tangent vector t for the curve at that point. The second vector is the curve's normal vector n. The third vector, the binormal vector b, is defined by taking the cross product of the tangent and normal vectors. For each vector we have a corresponding plane. The user can slide the plane corresponding to b along the reference curve. We present a profile of all the faces that cut through this plane in the editing window. The user can then re-sketch what the profile looks like, and have their changes applied back onto the mesh.

6.1 Calculating the Profile Planes

Based off the reference curve, we build three planes that define a local coordinate system for the profile, much like a frenet frame. The first plane, which approximates n, runs along the reference curve and cuts through the center of the mesh. The second plane, which approximates t, is perpendicular to the first and runs lengthwise along the curve. The third plane, which approximates b, is also perpendicular to the first and, by default, cuts along the center of the curve. The third plane can be moved so that it lies at any point along the reference curve. For clarity, let us refer these planes as the n-plane, t-plane, and b-plane, respectively. See figure 6.1(a) for an illustration.

To calculate the n-plane we find a plane of best fit for the reference curve's control points. We can do this easily using a least squares solver. Given n control points, we set up the equation $Ax = B$ such that A is a 3 by n matrix, x is a 1 by 3 matrix, and B is a 1 by n matrix. The solver finds the normal for a plane of best fit. A holds the x and y coordinates for all the points, and a 1 in place of the z coordinate. The z coordinate is placed in B. After the solver runs, x holds the new normal (a, b, c). We now have a normal but still need to find a point on the plane in order to completely describe it. Hence we choose the point that lies halfway along the reference curve. For drawing purposes, the planes are scaled so that they are large enough to cut through all points on the mesh.

Calculating the first plane is the most difficult; the other two planes can be derived from the first. The second plane is defined by point along the curve a and the vector $(a.z, a.y + s, a.z - s) - (a.x, a.y + s, a.z + s)$. Similarly, the third plane is defined by a and the vector $(a.x, a.y - s, a.z - s) - (a.z, a.y + s, a.z - s)$.



(a) All 3 planes (b) t-plane

(c) n-plane (d) b-plane

Figure 6.1: The profile coordinate system

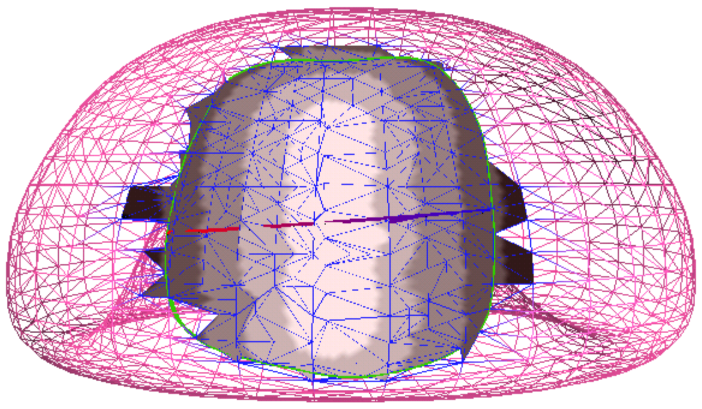
6.2 Calculating the Profile Cross-section

The profile editor displays a profile of the mesh along the second plane, cutting horizontally across a midsection of the ROI. To present the user with a profile, we calculate where the mesh intersects the b-plane. We start on one point of the boundary curve and move across the mesh faces until we intersect the plane again on the opposing side. To find the starting and ending profile points, we check to see where the boundary curve intersects the cross section plane. The boundary will intersect it at least twice. If there are more than two intersections, we only keep the leftmost and rightmost ones. To determine how far left a point is, we check to see how close it is to the average of the profile plane's two leftmost corner points. The closer it is to this point, the farther left it is. The averaged point is guaranteed to be farther left than any point on the boundary because of our choice of s. s is dependent on the mesh size. Similarly, to find the rightmost point, we choose the intersection that is farthest from the averaged point. Our cross-section building algorithm always scans from left to right, so we sort the final two boundary intersection points into this order.

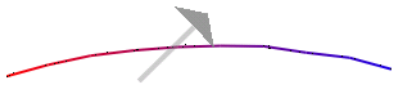
After finding the starting and ending boundary points, b_s and b_e, we are ready to traverse the mesh faces. Initially, we make note of what faces b_s and b_e lie on. We build a plane that holds points b_s, b_e, and the mesh center. Starting with the face under b_s, f_s, we move towards f_e along the plane, checking to see what mesh edges it intersects. We first check to see what edges of f_s intersect with the plane. If more than one intersects, we choose the one that is closer to b_e. Then we move onto that edge's opposite face and repeat the process until we reach f_e. We will be left with a series of points that form a nice profile across the ROI. The points are fitted to a NURBS curve, rescaled to be between the ranges of 0 and 1, and sent to the profile editor window.

6.3 Profile Editor User Interface

The profile editor window presents the user with an uncluttered view of the mesh's cross-section (figure 6.2). At this point, the user can re-sketch the curve to be a different shape. The same curve re-sketching technique available for editing reference and boundary curves can also be used here. Once changes are accepted, the mesh vertices are deformed to match the new profile. Exactly how the mesh is deformed will be discussed in chapter 8. The profile's orientation is indicated through use of a color gradient and an orientation arrow. The arrow points from the center of the curve, through the center of the profile, and towards the outside of the mesh. The color gradient is applied to the profile curve in the editor as well as on the original



(a) Profile presented on mesh



(b) Profile presented in editor

Figure 6.2: Profile Editor UI

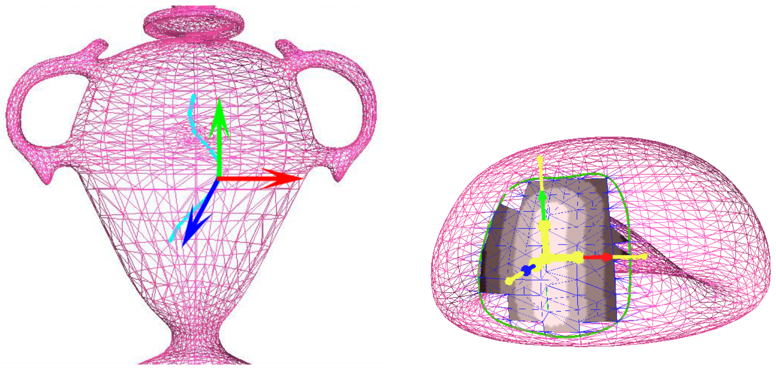
mesh (see figure 6.2). It helps the user correlate which part of the profile belongs on the left as opposed to the right.

Chapter 7

Direct Editing

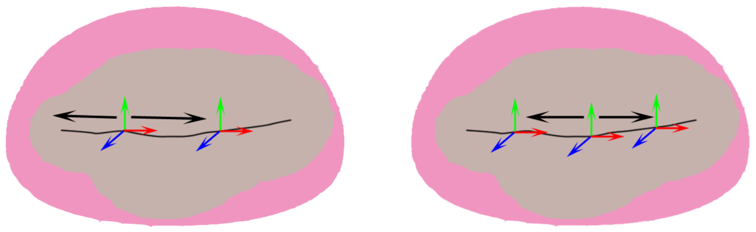
While users can use the two sketch-based editors to do change the mesh, we also present them with the option of directly editing the mesh by manipulating *handles*. Editing handles are analogous to the shading editor's handles; they are used to both introduce changes as well as limit the extent of deformations.

Figure 7.1 shows an image of an editing handle. It is composed of three orthonormal axis, defining a local coordinate system at that point. If a user clicks on an axis, the mesh will deform only along that dimension. Clicking on the inner third of the axis lets the user translate, clicking on the middle third lets the user scale, and



(a) Stylized handle (b) Actual handle

Figure 7.1: Handles



(a) 2 Handles (b) 3 Handles

Figure 7.2: Arrows indicate a particular handle's range of influence

clicking on the outer third lets the user rotate. After placing a handle, users can drag it around its axes.

To directly edit the mesh, the user sets handles down along the reference curve. When a handle is dragged, the changes in its position and orientation are used to create a geometric deformation matrix. The matrix is applied to all the vertices within the ROI, with varying levels.

Similarly to the shading editor, if users only want to deform parts of the ROI at a time they can set multiple handles. If a handle is dragged, it only updates the region of the mesh that is located between its neighboring handles. If a handle lacks a neighbor on one side, its influence extends in that direction until the end of the reference curve 7.2. If no handles have been placed along the reference curve, then the re-sketching deformation will extend throughout the entire ROI.

Chapter 8

Mesh Deformation

We have covered how to edit the mesh using both sketch-based techniques as well as a direct editing technique. Now that we have discussed how the editors are used to introduce deformations, let us discuss how the deformations are performed mathematically.

Once users have defined a ROI, our system automatically pops up a default texture. This is the same default texture that is displayed the first time the shading editor is opened. We simply display it initially so that users can skip directly to deforming the mesh if choose.

Once users specify a deformation, it is applied using Marc Alexa's matrix operators [1]. For each vertex, the percentage of the deformation that gets applied is determined by the grayscale texture value for that vertex. As discussed, the texture is set with the shading editor. Vertices that lie in a white texture area will move along completely with any deformation made by with a handle. Vertices in darker areas will move less. A vertex in a black region will not move at all, no matter what deformation is specified. After users finish specifying a deformation with one of the editors, the ROI is retriangulated and the texture is updated.

8.1 Mesh Retriangulation

Deformations can radically change vertex positions and introduce oddly-shaped triangles. By retriangulating we keep our triangles about the same size on average, and as close to equilateral as possible. A deformation might introduce an area of high curvature. Such an area needs more vertices in order to capture the curvature fully. Our retriangulation algorithm is sensitive to changes in curvature and will strive to preserve more detail in high curvature areas, but less in almost planar areas.

When we first load a mesh, our system calculates the average edge length, e_{avg}. e_{avg} is only calculated once. When we are retriangulating, we use then e_{avg} as a metric against which to evaluate edges for either swapping, collapsing, or splitting operations. If an edge is much shorter than e_{avg} is will be collapsed, where as if it is much longer than it will be split. The exact factor by which it must differ from e_{avg} depends upon the curvature of the neighboring area.

8.2 Matrix Operators

To apply deformation matrices, we use the $\bigodot$ and $\bigoplus$ operator as defined by Marc Alexa [1].

As explained by Amy Hawkins in [8]:

> The $\bigodot$ operator implements scalar multiplication of a transformation matrix. Example: Given a scalar s and matrices A and B (which may be any combination of rotation, translation or scale matrices), if $B = 2 \bigodot A$, then applying B gives the same result as applying A twice.
>
> The $\bigoplus$ operator is similar to matrix multiplication, with the exception that the $\bigoplus$ operator is commutative. The operator is defined such

that, if $AB = BA$, then $A \bigoplus B$ gives the same result as regular matrix multiplication. In the case where $AB \neq BA$, then $\bigoplus$ can be understood as applying A and B simultaneously. The operators are implemented using the matrix logarithm and exponential, as follows:

$$s \bigodot A = e^{s \log A}, \tag{8.1}$$

$$A \bigoplus B = e^{\log A + \log B}. \tag{8.2}$$

Once these two operators are defined, we can write the usual linear interpolation equation using Alexas operators:

$$[(1-t) \bigodot A] \bigoplus [t \bigodot B] t \in [0,1]. \tag{8.3}$$

To find percentages of the deformation matrices that satisfy the grayscale texture's constraints, we linearly interpolate according to equation 8.3.

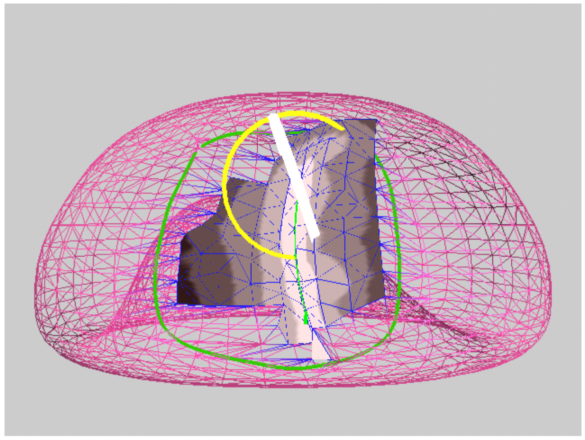
Chapter 9

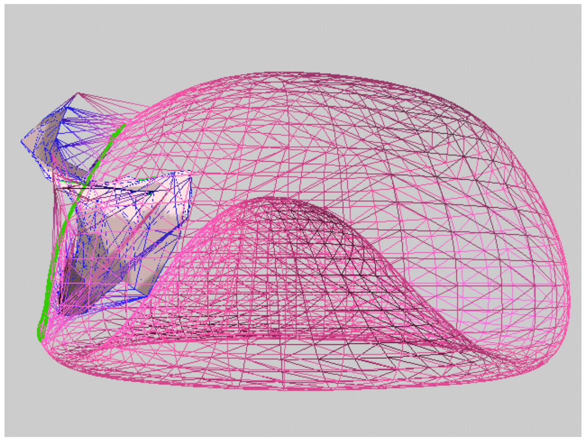
Results

We conclude by showing examples of different types of edits that can easily be performed with each one.

9.1 Editing Examples

The following figures show the results of deforming a mesh with our shading and profile editors.


(a) Rotation


(b) Successive rotation

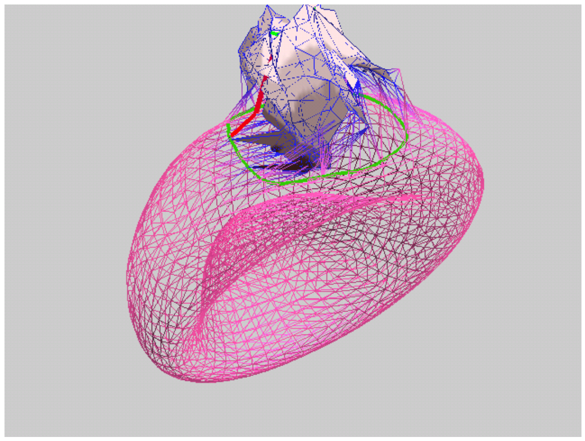

(c) Resketching of profile

Figure 9.1: Rotations and resketching

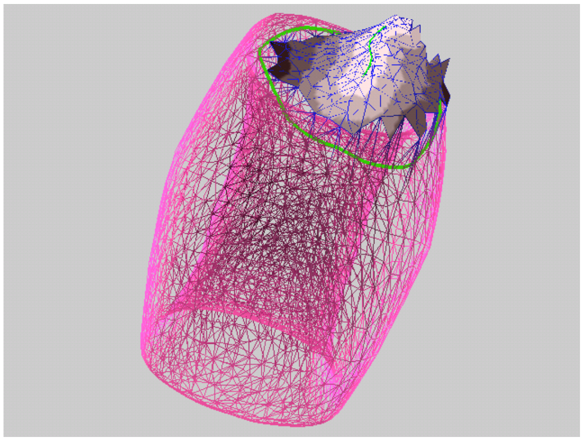
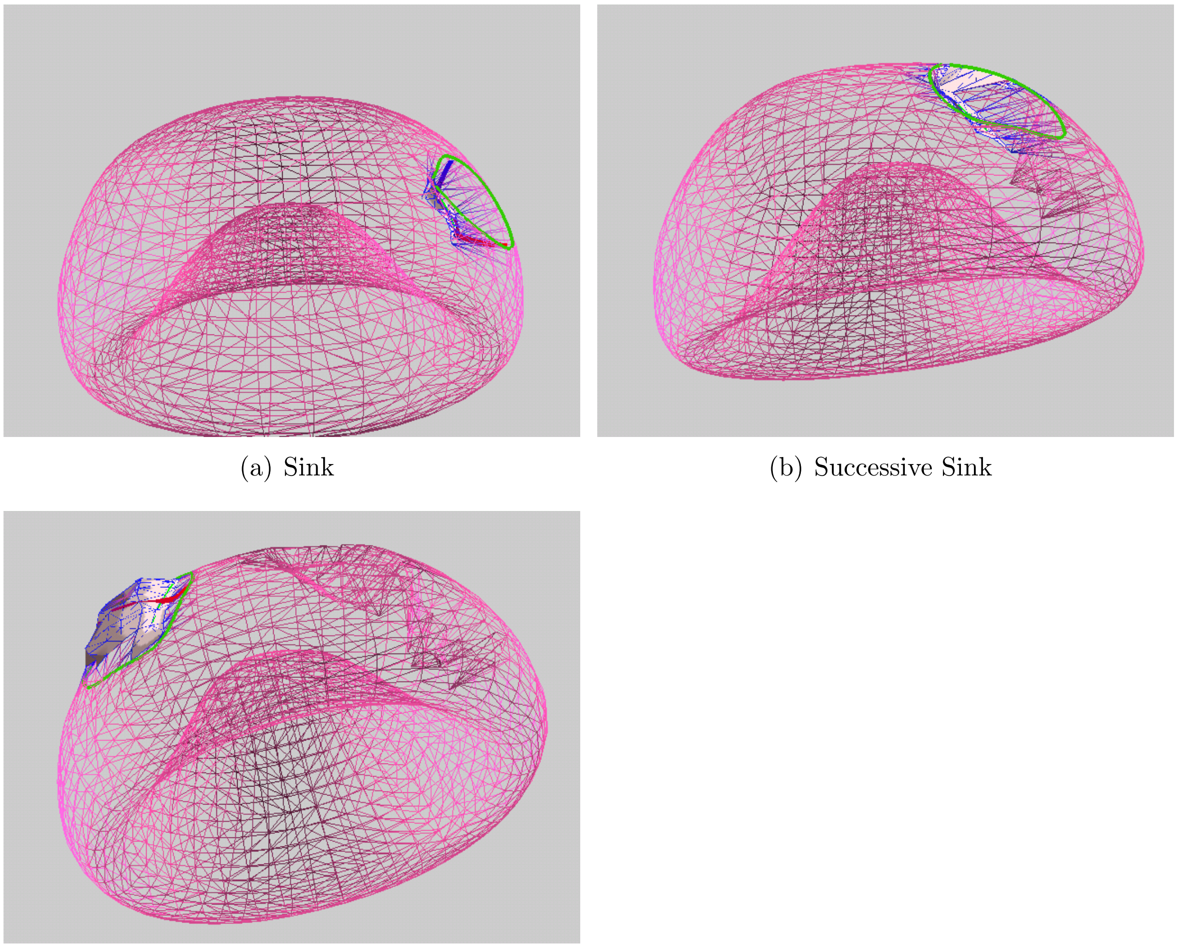


Figure 9.2: Rotate and extrude



(a) Sink

(b) Successive Sink

(c) Successive extrude

Figure 9.3: Sinking and extruding

Chapter 10

Future Work

Due to time constraints, we did not have time to evaluate our system against other standard graphics packages. Doing a proper comparison is necessary in order to see where our system excels. Feedback from a comparison analysis can also be used to help us decide what improvements need to be made. Evaluations can be conducted by performing a user study as well as running time analysis.

10.1 User Study

A user study should be conducted in order to compare our system against other commonly available modeling tools. Ideally, the study should be conducted on artists who are familiar with digital 3D modeling. A participant would be asked to complete the same task in our system and 2 other systems, such as Maya or 3DS Max. The time elapsed until completion should be noted, as well as any other particular difficulties the participant had. After performing several rounds of tasks, participants can describe their experience by filling out a short questionnaire. Some questions should aim to quantify the experience (using a multiple choice answer of "agree very strongly", "agree", "neutral", "disagree", or "disagree very strongly"). Others should present an area for open-ended comments. Good multiple choice questions to ask would be:

- Were tasks more difficult to complete in our system than in other graphics packages?
- Did the user interface feel more intuitive than in other graphics packages?
- Did you feel our system let you complete tasks more quickly?

Some good open-ended questions to ask would be:

- Were some tasks more difficult to complete than others? Why?
- Were some editors more difficult/unintuitive to use than others? Why?
- Are there any changes to UI that could be made to improve its effectiveness/ease of use?
- Would you consider using this application in your daily work? Why or why not?

To fully realize the usefulness of our tools, we need to evaluate how much our system increases productivity over other existing tools, and what advantages users perceive in our user interface. Constructive feedback from a user study could be used to increase the effectiveness of our UI.

10.2 Implementation Changes

Our implementation as it stands has several areas in which it could be improved. The most pressing issue is that it needs to run faster. While it does work at an interactive rate, a user's operations still take several seconds to update. The largest bottleneck occurs with the number of calculations we perform when evaluating splines. Distance calculations involving splines are performed often, and require a high degree of accuracy. To improve speed, we could lower the accuracy on our distance measures. Another bottleneck occurs when intersecting rays with the mesh. When users draw a spline, we project their strokes onto the surface of the mesh. In order to find which face a spline's control point lies in, we project a ray from the camera's eye point, through the control point, and see where it intersects the mesh. To improve speed, we can partition the search space using a data structure such as oct-trees. However, since our mesh geometry changes dynamically, any space partitioning data structure would have to be recalculated often. A better solution would be to work on optimizing our intersection code as it stands.

Appendix A: Camera Control Implementation

Keyboard shortcuts can be used to alter the camera in any direction. See table 1 for a complete listing of shortcuts. Pressing the ‘r’ key spins the mesh counter-clockwise, similarly to the spin dial. Pressing ‘x’, ‘y’, or ‘z’ will orient the camera so that it is looking down that axis. Pressing ‘1’, ‘2’, or ‘3’ will rotate the camera around the x, y, or z axis respectively, by a small amount. For all shortcuts mentioned so far, pressing ‘shift’ at the same time will reverse the direction of their movement. Hitting the ‘w’ resets the camera to the default position, and hitting ’c’ centers the camera on the mesh. Pressing ‘h’ rotates the camera clockwise. However, if ‘h’ and ‘shift’ are pressed together, the camera will pan in. Similarly, pressing ‘l’ spins the camera counter-clockwise, but if pressed in conjunction with ‘shift’ it will pan the camera out. Pressing ‘i’, ‘m’, ‘j’, or ‘k’ rotates the camera up, down, left, or right respectively. However, if pressed in conjunction with the ‘shift’ key, the camera is panned up, down, left, or right respectively. Pressing ‘p’ pans the camera in, while ‘n’ pans the camera out. If pressed in conjunction with ‘shift’, these keys zoom instead of pan. Pressing the numbers ’7’ through ’9’ will toggle between the three different profile planes that appear when the profile editor is open. Pressing ’6’ will display all the planes at once, while pressing ’0’ will display none. Lastly, although not a camera control, the keyboard shortcut ‘q’ will quit the application.

In our system we present the user with four main camera controls, which together allow for complete control of the camera in all dimensionsThe first controller, a zoom slider, controls how big or small the scene is that a user sees. It multiplies the objects in the scene by a scaling factor which controls how large they appear. The second controller, a spin dial, allows a user to rotate the mesh clockwise or counter-clockwise up to 360 degrees. Spinning is useful for turning the mesh upside down quickly, while not changing which side of the mesh is facing forward.

Table 1: Camera Control Shortcut Keys

Key	Action	With 'shift' modifier
r	spins counter-clockwise	spins clockwise
x	looks down x axis	looks down negative x-axis
y	looks down y axis	looks down negative y-axis
z	looks down z axis	looks down negative z-axis
1	rotates clockwise around x axis	rotates counter-clockwise around x axis
2	rotates clockwise around y axis	rotates counter-clockwise around y axis
3	rotates clockwise around z axis	rotates counter-clockwise around z axis
w	resets camera to default position	-
c	centers camera on mesh	-
h	spins clockwise	pans in
l	spins counter-clockwise	pans out
i	rotates up	pans up
m	rotates down	pans down
j	rotates left	pans left
k	rotates right	pans right
p	zooms out	pans in
n	zooms in	pans out
6	toggles all profile planes on	-
7,8,9	toggles between profile planes	-
0	toggles all profile planes off	-

Lastly, the user can control the camera with a virtual trackball. With a virtual trackball, users can click anywhere on the mesh and drag their mouse. As the mouse moves, the mesh will rotate in the same direction, such that the point of the mesh on which the user clicked remains underneath the mouse. Using a virtual trackball is very intuitive and easy for novice users to learn.

Appendix B:
Mesh Rescaling Implementation

Rescaling a mesh's vertices is simple. We iterate through all the vertices in the region of interest, and keep track of the minimum and maximum x and y coordinates we see. If one of the minimum x coordinates is less than 0, we add its absolute value to the x coordinates for every vertex. The same is done if the minimum y value is negative. We also add the minimum x and y values to the maximum x and y values, respectively, if they are negative. For each vertex, we divide its x coordinate by the the maximum x coordinate (and likewise for y). At this point, all vertices will have x and y coordinates ranging between 0 and 1. For display purposes, we want all vertices to be in the range of -1 to 1. As our last step, we multiply all the vertices' locations by 2 and subtract 1. Note that since we flattened the mesh beforehand, for all vertices $z = 0$, and none of the z coordinates must be rescaled. The reference curve is also rescaled so that it retains the same relative position to the faces it lies on.

References

[1] Marc Alexa. Linear combination of transformations. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 380–387, New York, NY, USA, 2002. ACM Press.

[2] L. Balmelli, T. Liebling, and M. Vetterli. Mesh optimization using global error with application to geometry simplification. *Journal of Graphic Models, special issue on processing of large polygonal meshes*, 2003. Preprint available at www.balmelli.net.

[3] Thomas Baudel. A mark-based interaction paradigm for free-hand drawing. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 185–192, New York, NY, USA, 1994. ACM Press.

[4] Jonathan M. Cohen, Lee Markosian, Robert C. Zeleznik, John F. Hughes, and Ronen Barzel. An interface for sketching 3d curves. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 17–21, New York, NY, USA, 1999. ACM Press.

[5] T. Corman, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[6] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice*. The Systems Programming Series. Addison-Wesley, second edition in c edition, 1996.

[7] Cindy Grimm and Matthew Ayers. A framework for synchronized editing of multiple curve representations. *Comput. Graph. Forum*, 17(3):31–40, 1998.

[8] Amy Hawkins and Cindy Grimm. Keyframing using linear interpolation of matrices. In *Journal of Graphics Tools*, 2006.

[9] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, New York, NY, USA, 1996. ACM Press.

[10] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 19–26, New York, NY, USA, 1993. ACM Press.

[11] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: a sketching interface for 3d freeform design. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 409–416, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[12] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. As-rigid-as-possible shape manipulation. *ACM Trans. Graph.*, 24(3):1134–1141, 2005.

[13] Holger Diehl Josef Ponn, Udo Lindemann and Franz Mller. Sketching in early conceptual phases of product design: Guidelines and tools. In *Eurographics '04: Proceedings of the Eurographics Workshop on Sketch-based Interfaces and Modeling*, pages 9–18, 2004.

[14] Tao Ju, Scott Schaefer, and Joe Warren. Mean value coordinates for closed triangular meshes. *ACM Trans. Graph.*, 24(3):561–566, 2005.

[15] Youngihn Kho and Michael Garland. Sketching mesh deformations. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 147–154, New York, NY, USA, 2005. ACM Press.

[16] Peter Lindstrom and Greg Turk. Image-driven simplification. *ACM Trans. Graph.*, 19(3):204–241, 2000.

[17] Yaron Lipman, Olga Sorkine, David Levin, and Daniel Cohen-Or. Linear rotation-invariant coordinates for meshes. *ACM Trans. Graph.*, 24(3):479–487, 2005.

[18] Andrew Nealen, Olga Sorkine, Marc Alexa, and Daniel Cohen-Or. A sketch-based interface for detail-preserving mesh editing. *ACM Trans. Graph.*, 24(3):1142–1147, 2005.

[19] Kenneth P. Camilleri Christopher Spiteri Philip J. Farrugia, Jonathan C. Borg and Alexandra Bartolo. A cameraphone-based approach for the generation of 3d models from paper sketches. In *Eurographics '04: Proceedings of the Eurographics Workshop on Sketch-based Interfaces and Modeling*, pages 32–42, Grenoble, France, 2004.

[20] Pierre Leclercq Roland Juchmes. A multi-agent system for the interpretation of architectural sketches. In *Eurographics '04: Proceedings of the Eurographics Workshop on Sketch-based Interfaces and Modeling*, pages 53 – 61, 2004.

[21] Karan Singh and Eugene Fiume. Wires: a geometric deformation technique. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 405–414, New York, NY, USA, 1998. ACM Press.

[22] Jean-Philippe Pernot Bianca Falcidieno Franca Giannini Jean-Claude Lon Vincent Cheutet, Chiara E. Catalano. 3d sketching with fully free form deformation features (?-f4) for aesthetic design. In *Eurographics '04: Proceedings of the Eurographics Workshop on Sketch-based Interfaces and Modeling*, pages 9–18, 2004.

[23] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. Sketch: An interface for sketching 3D scenes. In Holly Rushmeier, editor, *SIGGRAPH '96 Conference Proceedings*, pages 163–170. Addison Wesley, 1996.

[24] Kun Zhou, Jin Huang, John Snyder, Xinguo Liu, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Large mesh deformation using the volumetric graph laplacian. *ACM Trans. Graph.*, 24(3):496–503, 2005.

Vita

Raquel A. Bujans

Date of Birth	May 1, 1982
Place of Birth	New York, NY
Degrees	B.S. Computer Science, December 2004

August 2006