A Manual Editor for Model Creation from Medical Images

Chand T. John
Scott L. Delp

December 3, 2005

Abstract

Advances in biomedical imaging and in the speed and use of computational methods in attacking biological problems have led to a proliferation of image data across an enormous range of physical scales. Processing of data is a necessary step in every field of biology and medicine in which images are a source of information. A plethora of automated and semi-automated registration and segmentation techniques have been developed by researchers for biomedical image processing. However, these techniques are often developed to work in one specialized context, and no general automated techniques work well in all fields that require the use of biomedical images. Thus, adequate tools for manually processing image data are an absolute must for any software package that attempts to provide image processing tools to researchers across a wide range of disciplines. I will describe one such software package, the 3D Slicer, and the manual segmentation and registration tools it contains to complement its automated algorithms. The 3D Slicer is free, open-source, cross-platform, and easily extensible and customizable to any developer's or user community's needs. In particular I will describe the features I added to its manual image editor that provide basic manual segmentation capabilities on par with existing commercial software catering to the same user community.

1 Introduction

Biomedical images have proliferated throughout many fields of biological research, ranging from molecular dynamics to musculoskeletal modeling to virtual endoscopy. Magnetic resonance imaging (MRI) and computerized tomography (CT) are used to produce volumetric images of macroscale anatomical features such as musculoskeletal tissue, brain tumors, and whole organs and organ systems. Cryo-electron microscopy (cryo-EM) is used to produce 3D shape descriptions of molecules. A variety of imaging techniques are used to produce geometric data describing biological structures.

Following acquisition, many fields have similar needs with regard to processing this geometric data. Common operations in many fields include *registration* of multiple image data sets to form a single, correctly aligned data set; *segmentation* of structures of interest from within the data; *visualization* of the data; *measurements* within the data; and construction of *3D models* of structures extracted from the data. A number of context-specific computer implementations have been produced to facilitate the extraction of 3D geometric models from volumetric biomedical images. Image segmentation has received extensive attention for the last three decades by computer vision, computer graphics, and medical researchers, but even the best techniques often serve only as a starting point for developing a proper segmentation technique for a given type of image data, and extensive manual assistance is still needed to augment an automatically generated segmentation, and most techniques work well only in very limited contexts. Model construction methods also have received some attention, and need to be fast and accurate to be useful to non-programmer users. Software is also needed to manipulate data volumes and models: modification of meshes and Boolean operations are common volume manipulation tasks.

However, the differences in needs for different biomedical research groups has led to the formation of unique, specialized image-to-model pipelines. No single software package exists that is used in all areas of biomedical image processing, even though so many vastly different fields share similar needs. For example, the Surgical Planning Lab (SPL) in Brigham and Women's Hospital (BWH) at Harvard University uses semi-automatic segmentation with pixel-level accuracy and the marching cube algorithm to construct 3D meshes

from segmentation, while the Neuromuscular Biomechanics Lab (NMBL) at Stanford University uses manual segmentation, subpixel accuracy, and a Delaunay-based lofting algorithm from sampled cardinal splines to construct 3D meshes. Even within a single research lab, one can find individual researchers who piece together their own software pipeline using whatever tools they happen to find available. Individuals will often use commercial packages such as 3D DOCTOR from Able Software, BodyBuilder from Vicon Motion Systems, Matlab from MathWorks, or Geomagic Studio from Raindrop Geomagic to put together a software pipeline from individual pieces. This fragmented use of image-to-model pipeline software has led to difficulties in sharing data between researchers, and requires new researchers to have to spend extensive time and effort in constructing their own image-to-model pipeline to conduct their own research. There is a great need for a single software pipeline to unify the needs of researchers across many fields of biomedical image processing, so that researchers can spend less energy developing robust software, can share data with ease, and can port their data to other applications tailored to their own specific needs.

However, biomedical researchers and computer scientists are now attempting to solve this problem. Spearheaded by the National Alliance for Medical Image Computing (NA-MIC), the 3D Slicer is being introduced as a freely available, cross-platform, open-source, extensible and customizable software package for use by researchers across a wide range of fields. The package is currently in use primarily by brain researchers, but users from a vast array of other fields also use and develop modules for Slicer. The 3D Slicer was originally developed in 1997 by David Gering at MIT as a unification of the visualization and image-guided surgery software existing in SPL. Since then its use, development, and robustness has grown tremendously. Individual developers can develop and add their own modules to Slicer. However, much of the work done since its original implementation has been on semi-automatic and automatic segmentation of data. Little effort was put into developing its manual segmentation tools, even though a large majority of the segmentation done today requires substantial manual input. From January to August of 2005, I developed a new manual image editor that has made Slicer a more practical tool for clinical researchers. I will describe the features I added to create this new image editor. All of my software changes have been incorporated into NA-MIC's upcoming release, Slicer 2.6.

2 Basic Slicer Architecture

Slicer is organized as a combination of C++, used for the computation and graphics, and Tcl/Tk, used for controlling C++ objects and managing windows. Slicer consists of its base code and its modules. The base code cannot be removed from Slicer, but any module can be added or removed simply by adding or removing a directory containing the Tcl and C++ code for the module, as long as one pays attention to any dependencies between the module and other modules. The C++ code of Slicer uses VTK, a freely available open-source library of graphics functions and structures, built on top of OpenGL.

Slicer consists of three main windows: the Menu window, the Viewer window, and the Tk console. The Menu window contains six *panels*: Data, Volumes, Models, Alignment, Editor, and ModelMaker. Each panel contains one or more *tabs*. The image editor I will describe is part of the Editor panel's Details tab. The editor is activated through the Editor panel's Effects tab.

3 Drawing Program Functionality

Slicer's original manual image editor contained a basic functionality for drawing and modifying control polygons. The user could click or drag the mouse to plot control points over an image slice. The user could also select and move any number of existing points. However, the drawing, selecting, and moving of points was a hassle, since switching between the Draw, Select, and Move modes required the user to manually click the button for the desired mode. Hence, to draw a polygon, select a few control points, and move those points, the user would have to click Draw, plot the points, click Select, select the desired points, click Move, and then move the desired points. The speed and comfort with which these basic operations can be performed increased greatly when this functionality was replaced with a behavior more like that of commercial drawing programs.

I implemented an automatic proximity detection algorithm for real-time switching between the Select and Move modes, so that the user can avoid having to repeatedly click on the Select and Move buttons to

switch modes during editing. If the current mode is Draw, then no automatic switching occurs because the user must be allowed to plot points at any location on an image at any time. But if the current mode is Select or Move, the automatic proximity detection algorithm will switch from Select to Move mode when the mouse pointer is "near" a set of selected control points. If the control key is pressed on the keyboard, Slicer will stay in Select mode at all times, thus allowing the user to select groups of points with ease, as in any standard drawing program. The proximity detection algorthm works as follows. The input parameters are the mouse pointer position (X, Y); the coordinates (x_i, y_i) of the N selected points; and a nearness threshold r, which is set equal to 3 in the current implementation of Slicer. Assuming the current mode is either Select or Move, the algorithm updates the mode based on the input parameters.

```
SetMode (X, Y, x[1..N], y[1..N], r)
 IF N < 1 THEN
    Mode := Select
 IF N = 1 THEN
    IF |X - x[1]| <= r AND |Y - y[1]| <= r THEN
       Mode := Move
    ELSE
       Mode := Select
 IF N > 1 THEN
    r2 := r * r
    FOR i := 1 TO N - 1
       L := PointSegDistSq (X, Y, x[i], y[i], x[i + 1], y[i + 1])
       IF L <= r2 THEN
          Mode := Move
          Exit this procedure
 Mode := Select
```

The PointSegDistSq function call computes the squared distance between a point (X, Y) and the line segment with endpoints $(x[i], y[i])$ and $(x[i+1], y[i+1])$. The algorithm is described in the "Distance to Ray or Segment" section of [4].

```
PointSegDistSq (X, Y, xi, yi, xf, yf)
 DistSq := Infinity
 (vx, vy) := (xf - xi, yf - yi)
 (wx, wy) := (X - xi, Y - yi)
 c1 := (vx, vy) dot (wx, wy)
 IF c1 <= 0 THEN
    DistSq := |(wx, wy)|^2
    RETURN DistSq
 c2 := |(vx, vy)|^2
 IF c2 <= c1 THEN
    (ux, uy) := (X - xf, Y - yf)
    DistSq := |(ux, uy)|^2
    RETURN DistSq
 b := (float)c1 / (float)c2
 (pbx, pby) := (int)((xi, yi) + b * (vx, vy) + (0.5, 0.5))
 (dx, dy) := (X - pbx, Y - pby)
 DistSq = |(dx, dy)|^2
 RETURN DistSq
```

One important note is that all of the points above have integer screen coordinates. The use of square roots when computing distances is avoided by squaring quantities such as r. Floating point values are only used in the two lines where the variable b appears. The overall use of integers and simple arithmetic operations wherever possible makes this proximity detection algorithm perform very well in real-time, where these computations must be repeated each time the user moves the mouse pointer. In Slicer, changing the

mode variable's value automatically deactivates the previous mode's button and activates the new mode's button on the effects tab, so that the user knows when it is okay to move or select points based on which button is activated.

Another simple but useful drawing program-like functionality is that clicking the left mouse button while in Select (and not Move) mode will deselect all control points. This eliminates the need for the user to explicitly click the Deselect All button in the Menu window. The Deselect All operation is now in the Edit: menu in the Menu window.

4 Inserting Intermediate Points

BodyBuilder [1] contains one important polygon editing operation that Slicer lacked: the ability to insert points between existing control points to allow the user to manually refine a polygon. The implementation of this feature posed a challenge: when the user clicks in an arbitrary location on an image, how do we know where in the polygon to insert this point? One solution would be to ask the user to select the two points in between which the new point will be inserted, but this would make the task of simply inserting a point too laborious for the user. After all, BodyBuilder does not require any extra input from the user.

I implemented a proximity-based insertion algorithm as follows. The inputs are the mouse click location M and the current, potentially empty control polygon being drawn by the user. Note that one property of the control polygon is whether it is open (like a polyline) or closed (like a polygon).

1. If there are less than two control points in the polygon, add the new point M to the end of the control polygon.
2. If there are two points in the polygon, add M between the two points.
3. Now assume there are at least three points in the control polygon.
4. Find the nearest control point Q to the mouse click location M.
5. Let L and R be the control points preceding and succeeding Q in the polygon. If Q is the first point in the polygon, L will be set equal to the last point in the polygon, and if Q is the last point in the polygon, R will be set equal to the first point in the polygon.
6. Compute the (integer) squared distances D_L and D_R from Q to L and R respectively.
7. If Q is the first point in the polygon, then if $D_L < D_R$, then if the polygon is closed, add M to the end of the polygon (because it is easier to add to the end of the polygon than to the beginning in the implementation), or if the polygon is open, add M to the beginning of the polygon since the first point is closer to M than the last point. If Q is the first point, but $D_L \geq D_R$, then add M as the point immediately after Q in the polygon.
8. If Q is not the first point in the polygon, then regardless of whether the curve is open or closed, simply add M as the point preceding Q if $D_L < D_R$, or add M as the point after Q if $D_L \geq D_R$.

This algorithm uses only integers and no floating point numbers throughout. This insert operation works well when the control polygon does not contain large fluctuations in curvature between adjacent control points. Typically, this poses no problems since most polygons are drawn with control points that closely approximate smooth shape boundaries without sharp pronounced edges. But when the user inserts points somewhat carelessly, the location of point insertion in the polygon can be counterintuitive. More intelligent ways of inserting points based on a point-to-curve distance calculation, or based on an assumption about the overall round nature of a polygon, could yield better results. See Figure 1 for an example polygon which can produce counterintuitive behavior during point insertion.

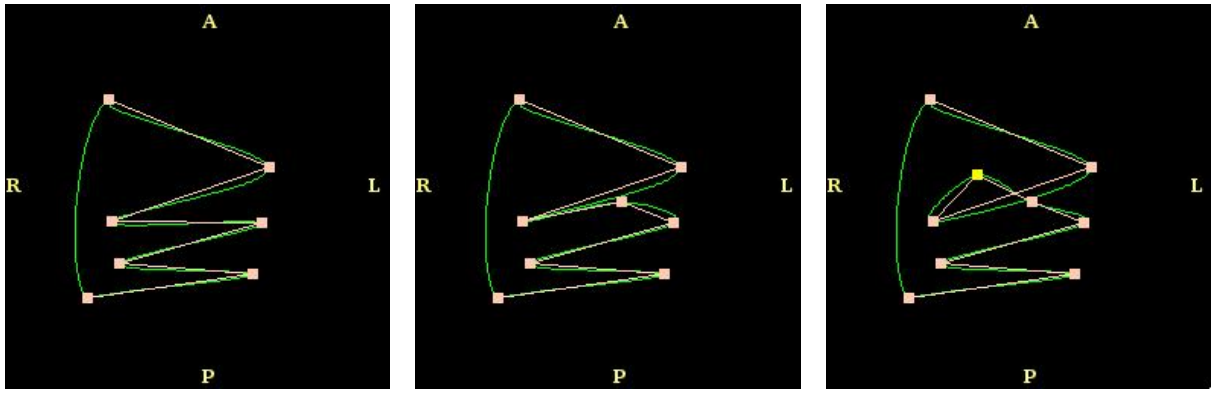

Figure 1: (Left) A control polygon drawn in Slicer. (Center) The result of inserting a point into the polygon. (Right) The result of inserting another point, colored yellow, which caused the polygon to have a self-intersection; we know the algorithm placed this point between the wrong pair of points since such a topological change to the polygon should never occur during segmentation.

5 Cardinal Splines and Sampling

BodyBuilder interpolates every control polygon with a cardinal spline in real time [1]. Sampling such a spline with arbitrary precision allows the user to send as much or as little geometric detail about each contour to the model construction algorithm in BodyBuilder. I implemented real-time interpolation of control polygons with cardinal splines and sampling of cardinal splines with arbitrary precision (up to pixel-level accuracy) into Slicer's manual image editor.

A cubic Bézier curve is a smooth polynomial curve determined by four points B_0, B_1, B_2, and B_3. Each point on the curve has a unique parameter value t between 0 and 1, inclusive. The point with parameter value t is computed using a recursive interpolation algorithm called the *de Casteljau algorithm*, which works by computing the first-order intermediate points

$$\begin{aligned} B_0^1(t) &= (1-t)B_0 + tB_1 \\ B_1^1(t) &= (1-t)B_1 + tB_2 \\ B_2^1(t) &= (1-t)B_2 + tB_3, \end{aligned}$$

the second-order intermediate points

$$\begin{aligned} B_0^2(t) &= (1-t)^2 B_0 + 2t(1-t)B_1 + t^2 B_2 \\ B_1^2(t) &= (1-t)^2 B_1 + 2t(1-t)B_2 + t^2 B_3, \end{aligned}$$

and finally the desired point, which is third-order (hence the name *cubic* Bézier curve):

$$B(t) = (1-t)^3 B_0 + 3t(1-t)^2 B_1 + 3t^2(1-t)B_2 + t^3 B_3. \tag{1}$$

Suppose we are given a control point P_i which is neither the first nor the last point on a control polygon with at least three control points. Then P_i is attached to two points P_{i-1} and P_{i+1} by edges of the polygon. To interpolate a cardinal spline through these points, we treat P_{i-1} and P_i as endpoints of one cubic Bézier curve, and treat P_i and P_{i+1} as endpoints of another cubic Bézier curve. But this only gives us two points for each of the two Bézier curves–recall that we need four points to fully determine a cubic Bézier curve. So, we impose an additional constraint on the derivative of the curve at each control point P_i:

$$P_i' = (P_{i+1} - P_{i-1})/2. \tag{2}$$

Now by differentiating equation 1, we obtain the equations

$$\begin{aligned} B'(0) &= 3(B_1 - B_0) \\ B'(1) &= 3(B_3 - B_2) \end{aligned}$$

for a particular cubic Bézier curve with control points B_0, B_1, B_2, and B_3. Thus if we have the derivative of a curve at B_0 and at B_3, then we can determine the other two control points B_1 and B_2. Equation 2 gives us the derivative at each curve's endpoints. We could actually pick any constant in the denominator in equation 2. We picked 2 to be the denominator, which makes our cardinal splines a popular type of curve known as a *Catmull-Rom spline*.

This gives us a way to compute pieces of the interior of a cardinal spline. But what about the curve joining the first and second control points, or the curve joining the last and second-to-last control points? Suppose we are interpolating a cubic Bézier curve through the first and second control points of the polygon. Then we want to compute points B_0, B_1, B_2, and B_3 such that $B(t)$ represents this cubic curve. The problem is, if we let the first control point be P_0 and the second control point be P_1, there is no control point preceding P_0 from which to obtain constraints like in equation 2. We propose the following solution: let the derivative P'_0 at P_0 be the reflection of the derivative vector P'_1 at P_1 over the edge $\overline{P_0P_1}$:

$$P'_0 = 2\left(\frac{P'_1 \cdot (P_0 - P_1)}{\|P_0 - P_1\|^2}\right)(P_0 - P_1) - P'_1.$$

We resolve the analogous problem for the last two control points P_{N-1} and P_{N-2} in the same way, so

$$P'_{N-1} = 2\left(\frac{P'_{N-2} \cdot (P_{N-1} - P_{N-2})}{\|P_{N-1} - P_{N-2}\|}\right)(P_{N-1} - P_{N-2}) - P'_{N-2}.$$

Finally, if the curve is closed, then we must draw another cubic Bézier curve between P_{N-1} and P_0. But here we can compute P'_{N-1} using P_{N-2} and P_0 as the previous and next control points; and we can compute P'_0 using P_{N-1} and P_1 as the previous and next control points, thereby completely determining a cubic Bézier curve connecting P_{N-1} and P_0.

For polygons with two control points, the entire cardinal spline we interpolate through the points will just be the line segment joining the two points. For polygons with one or zero control points, we do not interpolate anything.

Once a cardinal spline interpolating the control polygon is computed, it is rendered by rendering its individual cubic Bézier curves. Each cubic Bézier curve is rendered by repeated subdivision into small enough subcurves, each of which in turn can be approximated by line segments. The inputs into the algorithm are the four control points of a cubic curve, all with floating point precision (not integers). The recursive subdivision is essentially the de Casteljau algorithm with parameter value $t = 0.5$. A curve is considered to be approximately a line segment when the squared distance between the end control points of the curve is less than or equal to 4.

```
DrawCubicCurve (B0, B1, B2, B3)
 D03 := |B3 - B0|^2
 IF D03 <= 4.0 THEN
    DrawLineSegment ((int)B0, (int)B3)
 ELSE
    B01 := 0.5 * (B0 + B1)
    B11 := 0.5 * (B1 + B2)
    B21 := 0.5 * (B2 + B3)
    B02 := 0.5 * (B01 + B11)
    B12 := 0.5 * (B11 + B21)
    B03 := 0.5 * (B02 + B12)
    DrawCubicCurve (B0, B01, B02, B03)
    DrawCubicCurve (B03, B12, B21, B3)
```

The DrawLineSegment procedure rasterizes a line segment between two points on the image with integer screen coordinates. We stress that a cardinal spline is modified exactly corresponding to how its underlying control polygon is modified: if a point is added to the polygon, we recompute the whole spline. If a point is deleted, we again update the whole spline. If a point is moved, we still update the whole spline in real time. The algorithm has posed no speed issues. However, as is the case with many polynomial curves, fluctuations in curvature between adjacent control points can lead to wild behavior in the shape of the curve, and even

mild curvature fluctuations can lead to undesirable behavior at the ends of a cardinal spline. A simple solution to these endpoint issues may be to simply compute derivatives P'_0 and P'_{N-1} using the technique described above for closed curves, and to use these derivatives in computing the curve connecting P_0 and P_1 and the curve connecting P_{N-2} and P_{N-1}. A more fundamentally sound improvement to these issues may be to use implicit curves (level sets) to model these contours instead of parametric curves. Implicit curves yield far greater flexibility and generality of shape with simple equational representation. Methods for creating 3D finite element meshes from imaging data using implicit surfaces have already been developed [5].

The user must specify an integer sampling density $d \geq 0$ for the cardinal spline. The algorithm for sampling points from a cardinal spline works by sampling d points from each cubic Bézier curve segment of the spline, in addition to the already existing control points in the control polygon. So if $d = 0$, the only sampled points are the original control points themselves. Given a value for d, for each cubic curve segment described by a parametric function $B(t)$, the algorithm uses the de Casteljau algorithm to compute the points $B(1/(d+1)), B(2/(d+1)), \ldots, B(d/(d+1))$. This sampling algorithm works well in practice. The main advantages of the algorithm are simplicity and speed, while the main disadvantage is that high-curvature areas will be undersampled and low-curvature areas will be oversampled. An adaptive curvature-based sampling algorithm would yield more balanced results.

One controversy among image processing researchers is whether to allow polygons and splines to have subpixel accuracy, or to simply label structures by labeling pixels. NA-MIC's Slicer users typically label structures at the pixel level and may have no need for spline contours at all. However, NMBL researchers typically plot spline contours with subpixel-level accuracy. Making software to optimally satisfy both types of manual segmentation remains a challenge.

6 Unapply and Reapply Logic

A major feature missing in the original implementation of Slicer's manual editor was the ability to go back and re-edit contours that have already been applied. It is common for researchers to draw approximate contours on all slices of data quickly, and then go back and refine each contour. Although Slicer already had the ability to allow users to mark or unmark individual pixels or groups of pixels as part of a labelmap, it did not store the control points of the original polygons drawn by the user to generate the labelmap in the first place. Thus, there was a strong need for an undo feature, or rather, an "unapply" feature.

I implemented the unapply feature to have the following functionality: the user draws up to twenty polygons manually on each slice, and when the user clicks the Unapply button in the Menu window (without moving to a different image slice), one of the polygons on that slice is made editable (i.e., is unapplied). Slicer will in fact scroll through the polygons one at a time, each time Unapply is clicked on that image slice. Currently, up to 200 slices of image data may be segmented. Soon these limitations will be removed by reimplementing the stack data with STL vectors instead of fixed-size C-style arrays.

The data structure for managing these polygons is called vtkStackOfPolygons. A vtkStackOfPolygons object contains a fixed-size array of 200 vtkPolygonList objects and 200 Boolean variables indicating whether each of these vtkPolygonList objects has ever had a polygon stored in it. Each vtkPolygonList contains a fixed-size array of 20 vtkPoints (polygon) objects, along with the following integer parameters for each of these 20 polygons: sampling density, closedness, preshape, label, and order.

When the Unapply button is clicked, the following algorithm is executed.

```
Unapply()
 Delete all points in PolyDraw
 s := CurrentSliceNumber()
 IF CurrentSlice != s THEN
    poly := Stack.GetFirstRetrievablePolygon(s)
    Set current draw color to poly's color
    n := poly.GetNumberOfPoints()
    FOR i := 0 TO n - 1
       Add ith point in poly to PolyDraw
    CurrentSlice := s
```

```
    PolyNum := Stack.GetFirstRetrievePosition(s)
    UnapplyNum := PolyNum
 ELSE
    PolyNum := Stack.GetNextRetrievePosition(s, PolyNum)
    IF PolyNum != -1 THEN
       UnapplyNum := PolyNum
       poly := Stack.GetPolygon(s, PolyNum)
       Set current draw color to poly's color
       n := poly.GetNumberOfPoints()
       FOR i := 0 TO n - 1
          Add ith point in poly to PolyDraw
```

The algorithm first checks whether the user has moved to a new slice since the last time Apply or Unapply was clicked. If the user is on a new slice, or if the user just started segmenting and is clicking Unapply for the first time (even before Apply was clicked), then the algorithm will retrieve the first nonempty polygon in the vtkPolygonList array for that slice and make that polygon editable by setting PolyDraw to equal it. If the user has already clicked Apply or Unapply on this slice without doing so on any other slice since then, the algorithm will retrieve the *next* polygon in the vtkPolygonList array for that slice (allowing the user to scroll through each of the applied polygons on a slice by repeatedly clicking Unapply). If there is only one polygon applied on that slice, then that polygon will remain selected no matter how many times Unapply is clicked. If no polygons exist on the slice, nothing will be retrieved and PolyDraw will remain unchanged. Some important variables are:

1. Stack: the vtkStackOfPolygons object used to store all polygons drawn by the user
2. PolyNum: initial value -1; the index of the polygon on which the Unapply and Apply algorithms are currently focusing
3. CurrentSlice: initial value -1; number of the slice on which the Unapply and Apply algorithms are currently focusing
4. UnapplyNum: initial value -1; the index of the polygon to be replaced when Apply is clicked next

When the Apply button is clicked, the following algorithm is executed.

```
Apply()
 s := CurrentSliceNumber()
 IF CurrentSlice != s THEN
    CurrentSlice := s
    PolyNum := -1
    UnapplyNum := -1
 IF UnapplyNum != -1 THEN
    Stack.RemovePolygon(s, UnapplyNum)
 PolyNum := Stack.GetNextInsertPosition(s, PolyNum)
 IF PolyNum = -1 THEN
    Exit this procedure since slice s polygon list is full
 Stack.SetPolygon(s, PolyNum, density, closed, preshape, label)
 UnapplyNum := -1
 IF ClearPointsBeforeApply == "Yes" THEN
    Clear current slice's labelmap
 NumApply := Stack.GetNumApplyable(s)
 FOR q := 0 TO NumApply - 1
    p := index of qth applyable polygon in Stack on slice s
    poly := qth applyable polygon in Stack on slice s
    n := poly.GetNumberOfPoints()
    IF n > 0 THEN
       Add pixels for poly to slice s of current labelmap
```

```
 IF DeletePointsAfterApply == "Yes" THEN
    Delete all points in PolyDraw
 ELSE
    Deselect all points in PolyDraw
```

If the last time Unapply or Apply was clicked was on a different slice, then we simply add the PolyDraw polygon to the polygon list for the current slice. If the 20 slots in the polygon list for the current slice are already full, we give up trying to add this polygon to the list and the algorithm halts. If we were successful, then we update the rendering of the labelmap for the current slice by including the newly applied polygon into the labelmap computation. If we already clicked Unapply on the current slice, we actually remove the unapplied polygon and replace it with the newly applied polygon. This allows the user to unapply, edit, and reapply any polygon on any slice at any time, since the user can already scroll through the already-applied polygons on any slice at any time by repeatedly clicking Unapply.

7 Edit Operations

Most Microsoft Windows programs contain three editing features upon which users rely to correct editing mistakes or efficiently repeat certain editing operations: cut, copy, and paste. The situation is no different for Slicer users. A user may want to make multiple copies of one polygon and paste them onto multiple slices instead of hand-drawing contours on each individual slice. A user may want to remove a polygon from one slice and paste it on another. Some users may even be accustomed to using the cut operation in lieu of the delete all operation as a safe way of being able to recover any information that was mistakenly deleted.

Initially Slicer contained a single polygon, PolyDraw, which stored the control points of the current (potentially empty) polygon being drawn by the user. To implement the cut, copy, and paste operations, I created another polygon object, CopyPoly, that would temporarily store a control polygon. The basic algorithms for the cut, copy, and paste algorithms are given below.

```
Copy()
 Delete all points in CopyPoly
 Add all points in PolyDraw to CopyPoly

Cut()
 Delete all points in CopyPoly
 Add all points in PolyDraw to CopyPoly
 Delete all points in PolyDraw

Paste()
 Delete all points in PolyDraw
 Add all points in CopyPoly to PolyDraw
```

These operations can always be performed by clicking on the appropriate buttons in the Effects tab's Edit: menu. Shortcut keys have also been implemented for these operations: Control+X for cut, Control+C for copy, and Control+V for paste, consistent with typical applications. Unfortunately, Slicer only responds to these shortcut keys when the Viewer window is the currently active window, and so its operation can be a bit confusing to users who are not aware of this fact. This problem applies to other features in Slicer as well and needs to be corrected.

The four other operations in the Edit: menu already existed in Slicer as individual buttons: select all, deselect all, delete selected, and delete all.

8 Other Additions

When exporting segmentation data to other applications for smooths surface construction, finite element meshing, and simulation, it is useful to have the points sampled from all splines drawn by the user on all slices, combined into a single point cloud. Programs such as Geomagic Studio can take this data and

construct a NURBS surface interpolating the points. I have implemented a feature to allow exporting of such data after the user has constructed a labelmap. The output file name has an extension ".pts" and simply lists every point plotted by the user on every slice, plus all of the additional points sampled from the spline contours, one point per slice. The points are listed in RAS coordinates.

9 Power User Functionality

The new image editor provides enough shortcut keys so that "power users" can segment large amounts of data with relative ease. The user can typically keep one hand on the keyboard and one hand on the mouse. The following shortcut keys have been implemented:

1. UP ARROW: Apply
2. DOWN ARROW: Switch between Draw, Select/Move, and Insert modes
3. LEFT ARROW: Scroll to previous slice
4. RIGHT ARROW: Scroll to next slice
5. CONTROL+A: Select all points
6. DELETE: Delete selected points
7. CONTROL+D: Delete all points
8. CONTROL+X: Cut polygon
9. CONTROL+C: Copy polygon
10. CONTROL+V: Paste polygon

In normal circumstances, then, the user can start on the first slice, draw and edit a contour, press the up arrow key to apply the contour, and then press the right arrow key to scroll to the next slice, and repeat. The left arrow key can be used to come back to earlier slices for re-editing.

10 Problems with Current Slicer Architecture

As mentioned before, one implementation challenge is to optimally accommodate the software needs for both pixel-level accuracy and subpixel accuracy for segmentation. There are also other issues we are still only becoming aware of regarding different practices among different research groups. Also, as users request additional features, the user interface becomes increasingly more complex. At the same time, users want simplicity of use. It takes some compromises and some cleverness of design to reconcile these two opposing demands. In addition, the diverse needs of Slicer's large user community lead to the addition of different types of features for different groups that can conflict with other groups' needs or desires.

There are also problems with Slicer's current organization on the developer side. For one, the C++ code is inherently difficult to debug in the conventional ways. The most effective way is to wrap output statements with Debug and GlobalWarningDisplay statements:

```
DebugOn();
GlobalWarningDisplayOn();
.
.
vtkDebugMacro( << outputString );
.
.
GlobalWarningDisplayOff();
DebugOff();
```

Slicer's Tcl code is considerably easier to debug since one can always run Slicer and step through hypothetical Tcl code inside the Tk console window itself. However, some programmers feel that Tcl is a difficult language to debug in since the most effective method is to step through the code manually.

11 Conclusion

To satisfy the growing needs of the user base and improve the developer experience, Slicer needs a major architectural reorganization. Some expected improvements coming to Slicer 3.0 include the use of VTK widgets, which will greatly expand the basic graphical tools and capabilities of Slicer. These changes will not appear in Slicer 2.6, however. Overall, Slicer is expected to be the tool of choice for image processing needs across all biomedical fields. Its open distribution and large user and developer base, combined with the ease of learning to use it and develop with it, make it a powerful tool for researchers across all image processing fields. The ability to easily plug in new ITK-based image processing algorithms using the vtkITK module allows researchers to readily test any algorithms they have implemented in ITK. The image editor itself needs several improvements as well, as mentioned in the individual sections earlier.

12 Acknowledgements

Thanks to Allison Arnold, Thor Besier, Silvia Blemker, Christie Draper, and Kate Holzbaur in the Neuromuscular Biomechanics Lab at Stanford who provided valuable input on the essential ingredients needed for an image-to-model software pipeline. Nathan Wilson provided detailed information about his own Tcl-wrapped VTK-based modeling software for cardiovascular biomechanics. Thanks to Steve Pieper for the help he continues to provide in all aspects of Slicer development. Ron Kikinis provided important information and advice on the needs of the Slicer user community. Bill Lorensen, Will Schroeder, and Luis Ibáñez provided further suggestions and encouragement for this work.

This work was supported by an NIGMS biocomputation predoctoral training fellowship. Stanford's NIH-funded Center for Biomedical Computation (Simbios) supported travel to UCSD and MIT for NA-MIC meetings.

References

[1] BodyBuilder manual.

[2] G. Farin. *Curves and Surfaces for CAGD*, 5th ed., Academic Press, 2002.

[3] http://slicer.org

[4] http://softsurfer.com/Archive/algorithm_0102/algorithm_0102.htm

[5] Y. Zhang, C. Bajaj, B. Sohn. 3D Finite Element Meshing from Imaging Data, *Computer Methods in Applied Mechanics and Engineering (CMAME) on Unstructured Mesh Generation*, 194, 48-49, 5083-5106, 2005.