Three-dimensional Reconstruction of Intravascular Ultrasound Data

K.R. Subramanian Michael Funk

26 October 1998

Acknowledgements

We would like to thank the staff of the Heineman Medical Research Laboratory at Carolinas Medical Center, without whom this project would not have been possible. They provided us with access to lab facilities for data acquisition, and the expert assistance of Brett Fowler. The contributions of Dr. Mano Thubrikar are also greatly appreciated; Dr. Thubrikar kindly volunteered his time in evaluating the reconstructions our software produced, and gave helpful professional advice throughout the development process. This project was supported, in part, by the National Science Foundation, grant number DUE-9651080.

Chapter 1

Introduction

Visualization is rapidly becoming a popular field of research within the medical community. A number of technologies widely used in medicine (such as ultrasound) lend themselves well to visualization techniques. One area in which this is particularly true is in intravascular applications. Physicians need to be able to locate and identify potentially dangerous structures found on the arterial wall, such as atherosclerotic plaque. This is important both for diagnosis, and in preparation for intervention.

Current research in medical volume visualization is for the most part centered around three technologies for data acquisition: CT (computed tomography), MRI (magnetic resonance imaging), and ultrasound. Each of these technologies has its own characteristic strengths and weaknesses. MRI produces very high quality data. However, it takes a relatively long time to gather the data, ruling it out as an option for real-time visualization applications. CT suffers from many of the same problems.

Ultrasound, while plagued by often "noisy" and inconsistent data, is nonetheless quite useful because of its relative lack of intrusiveness, and the rapidity and efficiency of the data-gathering process. In particular, for cardiovascular applications, ultrasound provides higher resolution data than the alternatives. Angiography, for example, only provides a silhouette of the vessel, making it difficult to produce useful reconstructions. Ultrasound is thus best suited for visualization systems in which interactivity and real-time responsiveness are highly valued.

This paper documents the development of a prototype of one such system, specifically geared towards intravascular ultrasound (IVUS) visualization. It is our belief that this system offers several distinct advantages over traditional means of analysis. In particular, great care was taken to accurately reconstruct curved blood vessels, by incorporating data from a secondary source (biplane xray fluoroscopy). Because of this, much of the work implies new methodology for the gathering of the data, in addition to providing new visualization tools with which to interpret it.

Accurate visualization of IVUS data offers a number of unique challenges to overcome. It is by nature extremely noisy; also, the data produced is in the form of a sequence of two-dimensional images. Three-dimensional reconstruction of these images takes place only in the mind's eye of the specialist who analyzes the data. The visualization software developed by this project attempts to ameriolate both of these areas of difficulty, first by applying filtering and image processing techniques to the original IVUS data, and secondly by using the filtered data to generate a three-dimensional reconstruction of the blood vessel itself. The applications software developed by this project allows viewing and analysis of both the original (2D) data, as well as the 3D reconstruction, which may reveal structures only implicitly defined by the original IVUS data. This software is designed not to replace traditional means of analysis, but rather to provide additional tools with which a diagnosis could be made.

An additional difficulty associated with 3D reconstruction of the IVUS data is the fact that all of the data is relative to the position of the catheter tip at a given time. A trivial solution is to ignore the problem and assume that the catheter tip is moving along one axis only, from one frame to the next. Prior research projects [10] have made this assumption for the sake of expediency, directing their intellectual focus to other, more pressing areas (such as filtering). Unfortunately, it is simply not possible to determine the path of the catheter (and, by extension, the blood vessel) from the IVUS data alone. It is necessary to use another means to track the position of the cathether tip in 3D space, while the IVUS data is being gathered. For this purpose we have employed biplane xray fluoroscopy; the exact mechanism used (described in detail later in this document) was rather cumbersome, but was suitable for our purposes. The focus here was more on what to do with the data once it was acquired, rather than on streamlining the data acquisition process itself. Nonetheless, if this sort of visualization is ever to become a useful and widespread tool for clinical analysis, much more work will need to be done in the area of data acquisition.

My own involvement in this project was as an applications programmer, both in development of the user interface and the onscreen rendering engine, and in the underlying visualization pipeline, which transformed the raw ultrasound data into polygonal surfaces suitable for rendering on a graphics workstation. In addition, I wrote a helper application which automated much of the image-processing work that needed to be done prior to the actual visualization. The main visualization application and the helper application are fully documented in appendices A and B, respectively.

St. Goar [6] and Yock [9] offer additional information on IVUS technology and its applications, although neither work is related to computer graphics.

Chapter 2

Data Acquisition and Preparation

2.1 Ultrasound and X-rays

One of the major goals of this project was accurate reconstruction of curvatures in the path of the blood vessel being examined. A trivial analysis of the ultrasound data does not take this into account; it seems very natural to assume that the catheter moves only along one axis, and that each frame of cross-sectional data is parallel to the next. In reality, this is rarely the case.

First, a note on the technologies and procedures involved. The IVUS data is generated by a transducer attached to the tip of a catheter, which is inserted into one of the patient's blood vessels, near the point of interest. The catheter is withdrawn at a constant speed, while data from the transducer is directed to a video output. The output is stored on videotape for later analysis. Due to the nature of the technology, the IVUS probe only generates two-dimensional data, along a plane perpendicular to its spatial orientation. Therefore, each frame of the videotape represents a cross-section of the vessel, at a given point along a segment of the vessel. When taken as a whole, the videotape's contents can be thought of as describing a volume of space containing the vessel. One is left with a sequence of cross-sections, one per frame.

Figures 2.1 and 2.2 should clarify this situation. The line in figure 2.2 represents the actual path of the catheter, in a hypothetical experiment. The squares arranged along this path represent the frames of data gathered by the probe. Recall that the probe generates a continuous video signal, and that at any given time the probe has a particular spatial orientation. Because of the way the probes are designed, the probe only “sees” in two dimensions, along a plane perpendicular to the orientation of the probe. Each frame of video data generated by the probe corresponds to what the probe “sees” at a given point in time and space, with the probe itself at the center of the frame. Each point in each frame will have a particular grayscale value; higher values represent a greater density at that point in space, as determined by the probe. Figure 2.3 is a typical example of the IVUS probe’s output.

The data is gathered during “pullback”; the term refers to the fact that the catheter has already been inserted into the patient’s vascular system, and is slowly being withdrawn, at a constant rate (typically on the order of a millimeter per second). The equipment used in our experiments generated a PAL video signal, resulting in 30 frames per second of video data.

Consider a volume of space, which is large enough to contain every point which was ever visible to the IVUS probe during pullback. Each point in this volume has a single scalar value associated with it, corresponding to the density of the material at that point. Each point on each frame of the IVUS data corresponds to a point in our volume. In order to do a proper 3D reconstruction of a curved vessel, it is necessary to map each frame onto a position within the volume. Since the catheter will generally follow a curved path during pullback, the positions of each frame within our volume will not necessarily be coaxial. This is the situation illustrated in figure 2.2. Figure 2.1 shows a far simpler situation, in which the catheter follows a straight path. Much of the previous work done in IVUS visualization has assumed a straight path for simplicity; one of the goals of this project was to generate accurate reconstructions which took into account the curvature of the vessel.

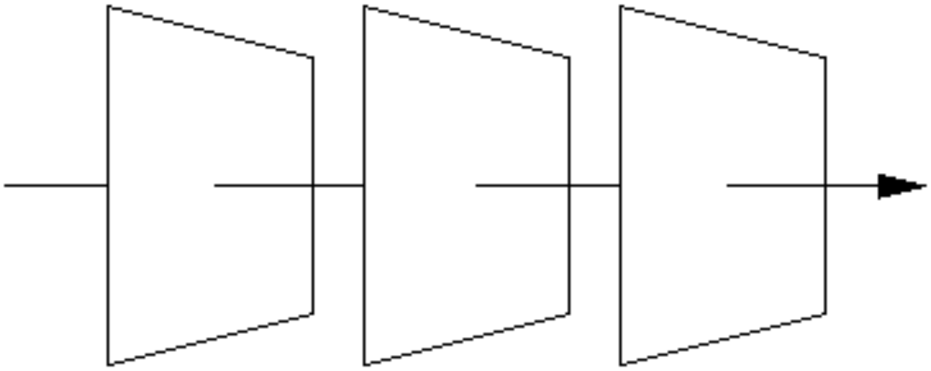


Figure 2.1: Mapping IVUS data to 3D space, assuming a straight path

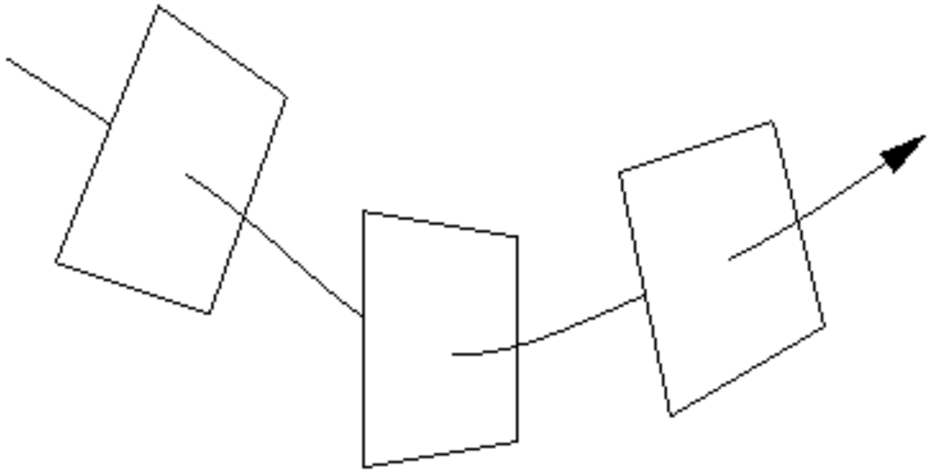


Figure 2.2: Mapping IVUS data to 3D space, curved path

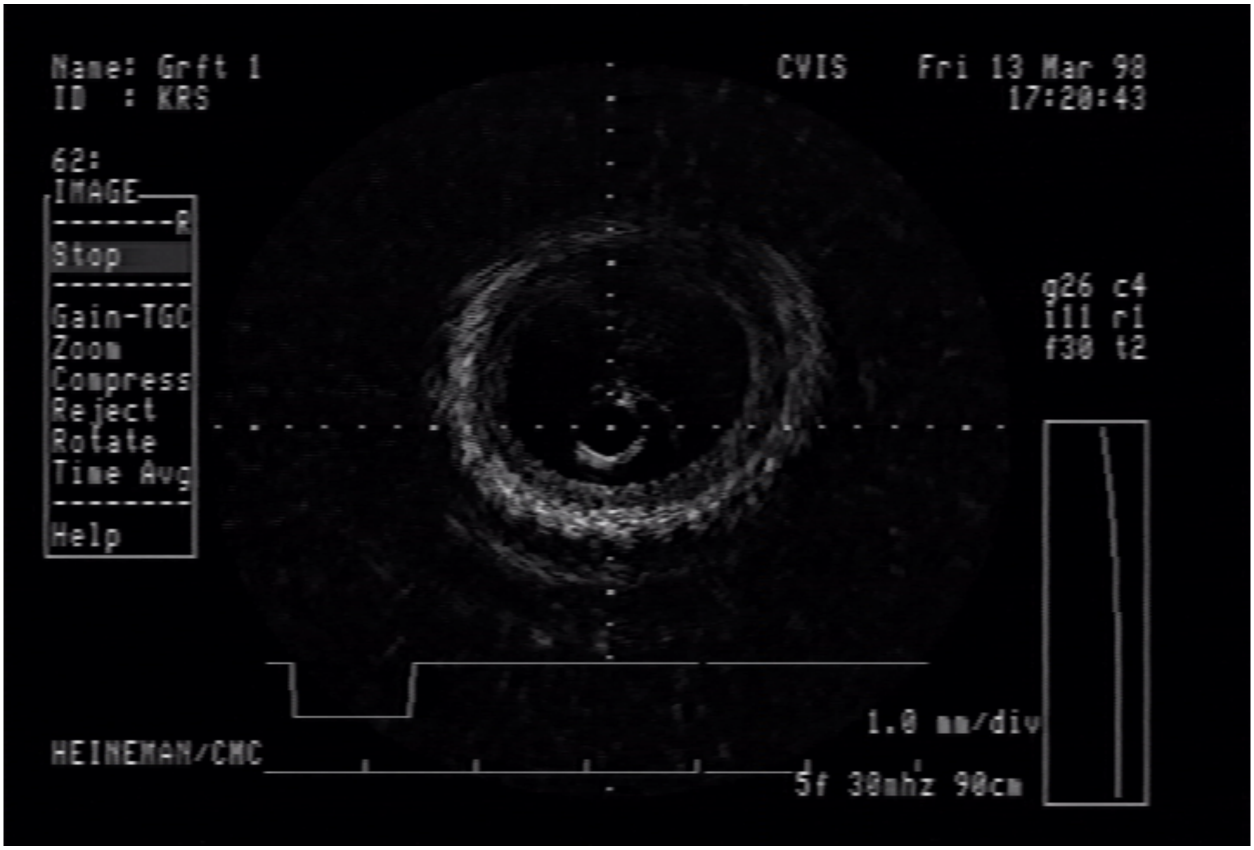


Figure 2.3: Typical IVUS probe output

In order to accurately model the curvature, it is necessary to keep a record of the path of the catheter tip itself. This is only made possible by introducing a second form of data acquisition. In this case, biplane X-ray flouroscopy was used. Two X-ray machines were set up in such a way that they were directed towards the catheter tip, along directions perpendicular to one another, as well as perpendicular to the general direction of motion of the catheter. Figure 2.6 shows the arrangement of the X-ray machines and the IVUS apparatus. To avoid confusion, it should be stated that neither live human nor animal subjects were used; in all of our experiments the catheter was drawn through either a length of tubing or a sample artery which had been removed from a subject. The output from the X-ray devices and the ultrasound probe were both recorded on SVHS for later analysis. By having two views perpendicular to the general direction of motion (as well as each other),

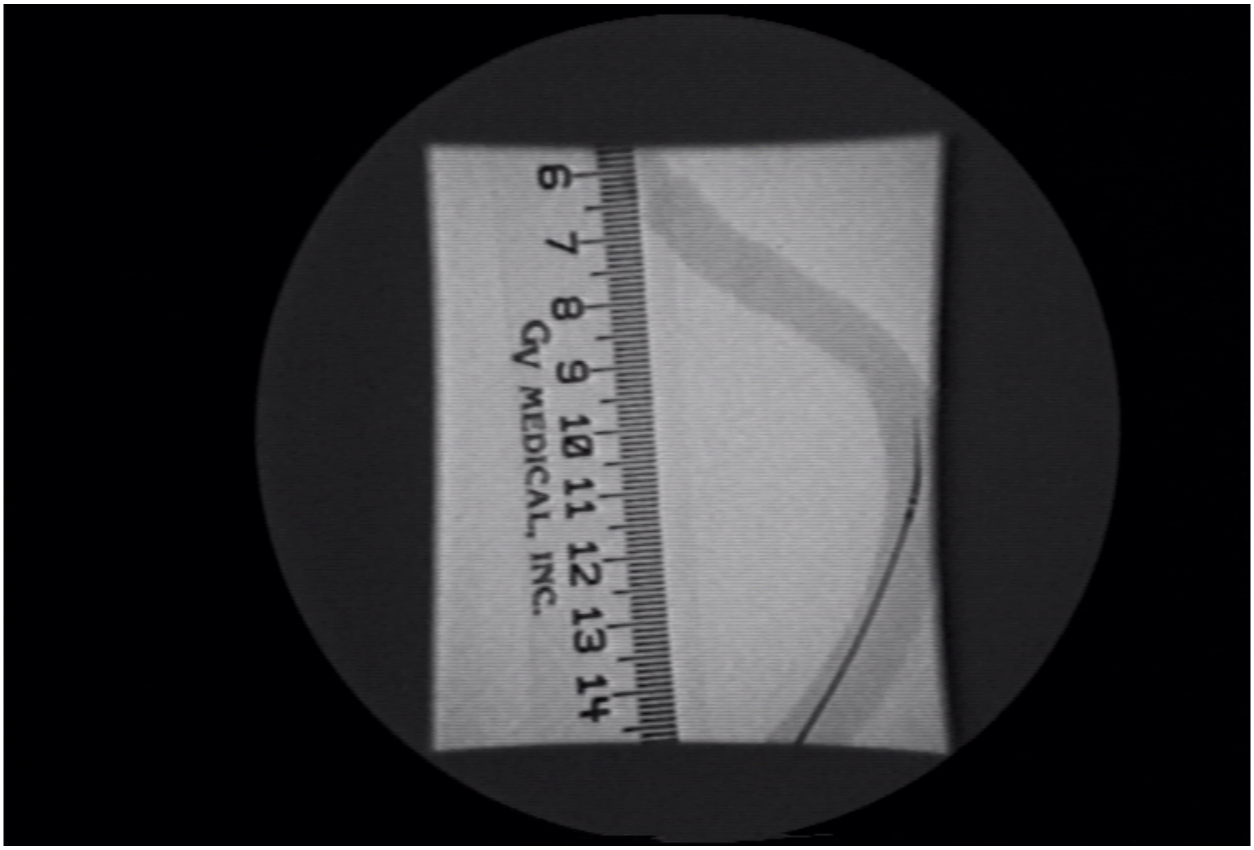


Figure 2.4: X-ray image, x view

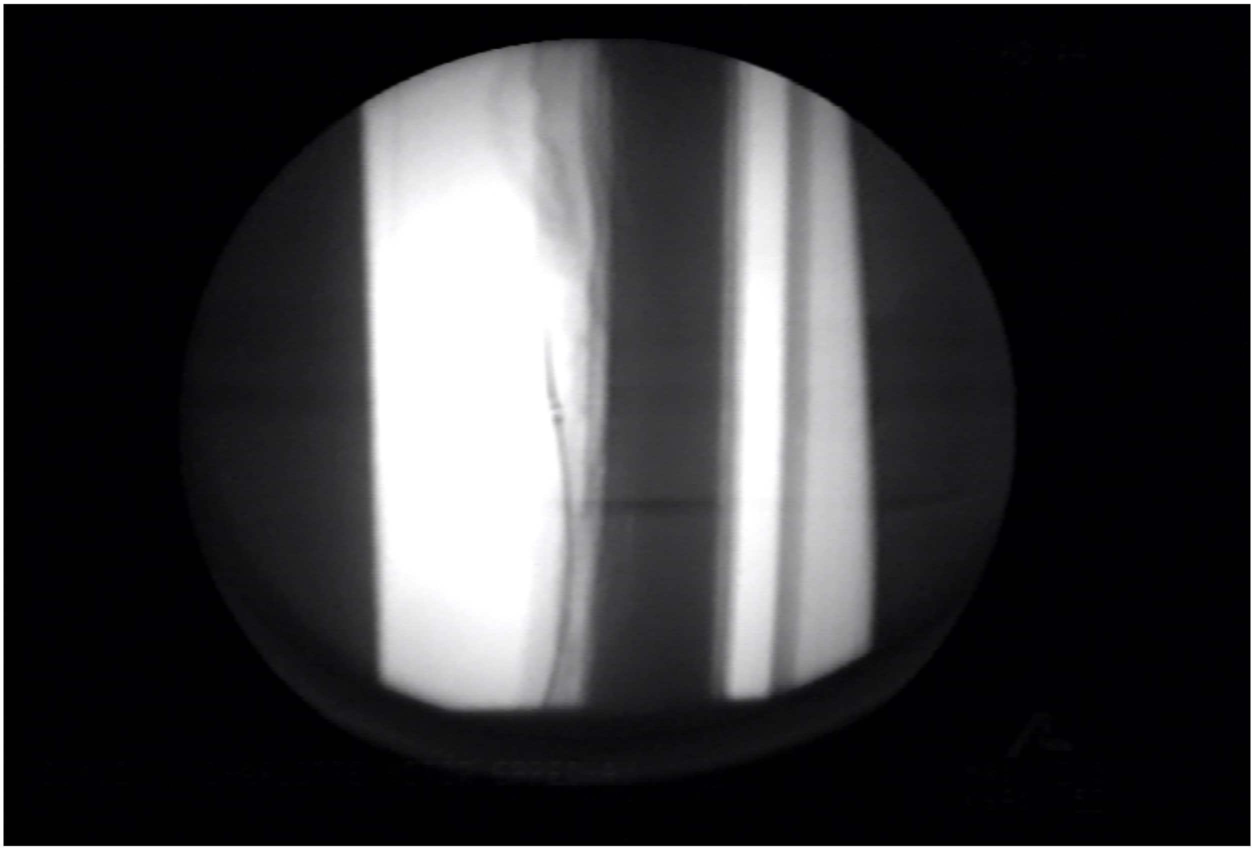


Figure 2.5: X-ray image, y view

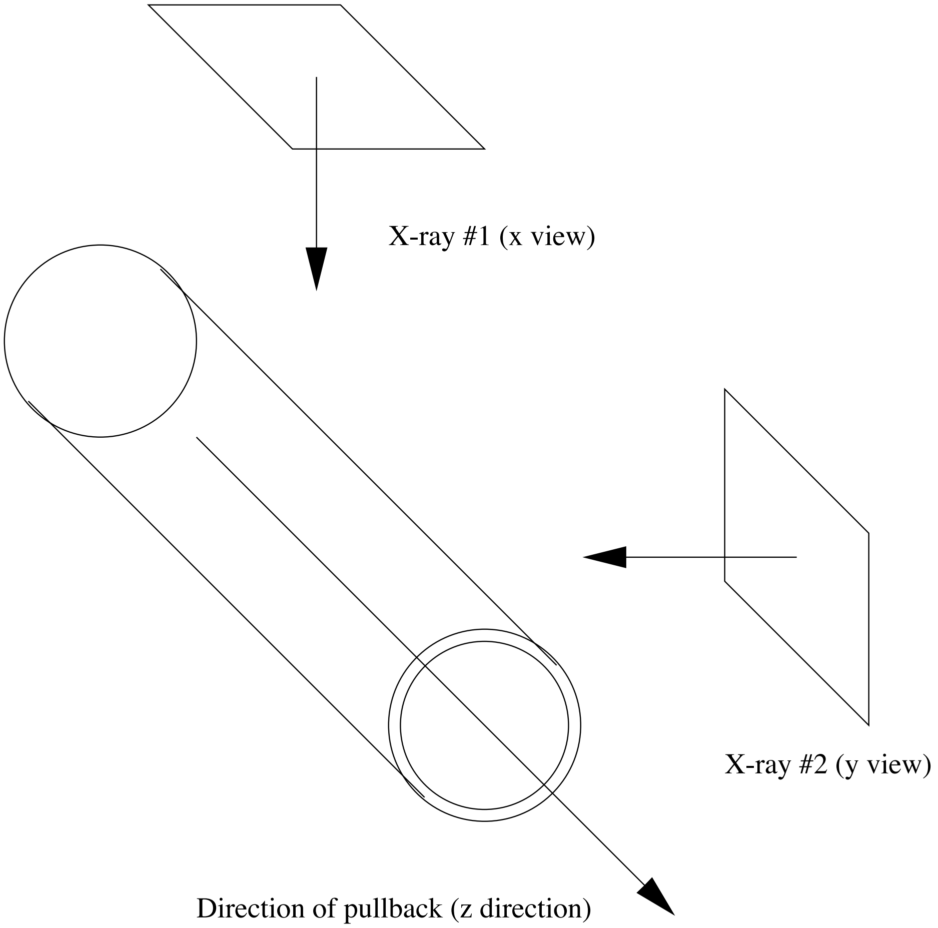


Figure 2.6: Configuration of X-ray machines

it was possible to determine the catheter tip's position in 3D space at each point along its journey, relative to its starting position. Figures 2.4 and 2.5 are examples of images taken from the two X-ray machines during pullback; the catheter is visible, as is the vessel itself, and a ruler placed nearby as a measurement aid.

While gathering the ultrasound data, the motion of the catheter was periodically stopped, and the X-ray machines were turned on to provide a snapshot of the catheter tip's current position. The mathematical procedure used to generate a spline corresponding to the curved path of the catheter (and the blood vessel) only required a sampling of points along the path in question. Various techniques were used to synchronize the X-ray images with the record from the ultrasound probe; in all cases the X-ray images were taken at discrete intervals, rather than continuously. On average, X-ray images were taken at 6-7 points during the motion of the catheter.

2.2 Digitization

After the data had been recorded to SVHS, an Abekas Diskus recorder was used to digitize the images stored on tape. The X-ray images were examined immediately, and processed by hand. The end result was a set of 3D coordinates derived from each pair of X-ray images, corresponding to the catheter tip's location in 3D space at the time the images were recorded. The coordinates were all relative to the initial position of the catheter. The 3D coordinates were derived as follows: for each pair of digitized X-ray images associated with each point (such as those shown in figures 2.4 and 2.5), a pixel count was taken from the edge of the image to the tip of the catheter (the location of the transducer). For the x images, the count was from the left edge of the image; for the y images, the count was from the right edge of the image. Once all points were determined, there were recalculated relative to the first point, which became the origin of the coordinate system. Since the the X-ray machines are set up to take perpendicular views of the catheter, any observed lateral displacement from one x image to the next can be taken as motion in the x direction only; likewise for the y images.

If the z-axis is taken to correspond to the general direction of the catheter's motion, then the X-ray images provide information regarding motion along the x and y axes. The displacement along the z-axis from one frame to the next was known from direct measurement at the time of data acquisition. Also, each pair of X-ray images was associated with a particular frame of IVUS data at the time of data acquisition; it was this knowledge which enabled us to associate the derived points in 3D space with their specific points in time in the IVUS data stream.

A final detail: the x and y coordinates for each point were in pixels, whereas the z

coordinate associate with each (x, y) pair was in millimeters. A small metal sphere was placed next to the catheter in some of the X-ray images; the sphere's diameter was known through direct measurement, and the image of the sphere was used to determine the pixels-to-millimeters conversion factor.

The end result of this procedure was, we had a sampling of 6-7 points in 3D space, representing points through which the catheter tip had traveled in its path. We also had a very large series of images from the ultrasound probe, frequently numbering in the hundreds (the SVHS recorder recorded 30 frames per second). We took a sampling of those images to construct a volumetric (3D) dataset corresponding to that particular ultrasound run, generally on the order of 100-150 images. The volume was defined as follows: each image represented a cross-section of the volume (an (x, y) plane), for a given z-value. The scalar value of a given point at (x, y, z) was obtained by looking at the grayscale value of the (x, y) pixel of image number z. Taken as a whole, the series of ultrasound images described a curvilinear volume of 3D space, in which each point in that space was described by a single scalar value. While that scalar value was obtained by examining the grayscale value of the corresponding pixel on the corresponding digital image, ultimately it represented the density of a point in space, as determined by the ultrasound probe. Note that at this point, the points derived from the X-ray images have not yet been used. Ultimately they will be used to derive a spline which closely approximates the path of the catheter, and that spline will be used to create a deformation of the volumetric data.

2.3 Filtering and Smoothing

Before 3D reconstruction could proceed, however, it was necessary to apply some image processing techniques to the volumetric data. As stated earlier, ultrasound can produce rather noisy data, but carefully applied filtering and smoothing can eliminate much of it. In addition, the ultrasound device itself had left several artifacts on the image that needed to be removed. The device superimposed tick marks over each image, presumably to facilitate measurement and analysis. These were easily removed by masking off the appropriate bits, as the pixels affected by the tick marks were consistent from one image to the next. Another artifact was produced by the transducer itself: a ghost-like shadow of the catheter, in the center of each image. This artifact was also consistent from one frame to the next, and thus it was also possible to remove it by masking off the appropriate pixels. Much of the image processing work described here follows the procedures and algorithms found in Zhou [10].

The first step in the image processing phase was to crop each image in order to remove the overlayed textual data which had been composited by the IVUS controller. Secondly, the tick marks produced by the device were removed by masking off the row and column of pixels where it was superimposed. While this had the negative side effect of damaging some of the original data, it was deemed a better alternative to simply allowing the tick marks to remain. Anecdotal evidence suggests that most IVUS equipment may be configured to not generate the tick marks in the first place; however, the equipment to which we had access did not have this capability. Thirdly, something had to be done about the catheter artifact. Since it was very nearly circular, and was consistent in form from one frame to the next (down to individual pixels), it was decided to simply mask off all pixels within a certain radius of the center of each image. Finally, we resized the resultant images to an acceptable width and height; generally, this meant scaling them down to 50-100 pixels in both dimensions, down from 300 or so, on average. This was done via simple point sampling; in the future, bicubic interpolation may be used for greater accuracy (at the expense of more processing time). The scaling was necessary because of the need to reduce the memory requirements of the program. In the final dataset, 100-150 of these images will be concatenated together to produce a volumetric dataset; in addition, the program which performs the actual 3D reconstruction and visualization needs to retain several copies of the dataset (and its derivatives) in memory at once, further reinforcing the need to keep the size of the datasets down to a minimum. Scaling the individual images down dramatically decreases the software's memory usage, and in our experience produces acceptable results.

Other problems were less predictable, however, and required more sophisticated filters. Seemingly random "speckles" cropped up on the boundaries of each image, a by-product of the noisiness inherent in ultrasound. Also, there was not necessarily a smooth progression of grayscale values across structural boundaries; rather, it was not uncommon to encounter drastic variations from one pixel to the next. Again, this could be chalked up to ultrasound noise. Both problems were resolved by applying a combination of Gaussian and/or median filtering to each image, which had the effect of eliminating the random "speckling", while also smoothing out the more dramatic shifts in value between adjacent pixels. In both cases, retaining the integrity of the data is of vital importance; the idea was to improve the signal to noise ratio of the data rather than to massage it into compliance with the software's needs. While loss of information is inevitable in such a situation, it is believed that the signal to noise ratio of the data was greatly improved, a more than satisfactory tradeoff. Figures 2.7, 2.8, 2.9, and 2.10 show the results of successive median filtering, a simple smoothing procedure whereby each pixel's value becomes the median of its original value and that of its 8 immediate neighbours. Figure 2.7 shows a typical IVUS image, after cropping, but prior to the application of the median filter. The remaining figures show the results of successive passes of the filter: 2.8 is after one pass, 2.9 is after two, and 2.10 is after three. Two passes seemed to produce the best results; after the second pass, successive passes appeared to "dim" the data to the point of obscurity. Neither the catheter artifact nor the tick marks have been masked off; as is evident from the filtered images, the artifacts are too strongly defined to simply rely on a smoothing filter to eliminate them. It is necessary to mask them off prior to the filtering process; in this case they were allowed to remain in order to demonstrate this very point.

As the image processing procedure was repetitive and easily automated, a small applica-

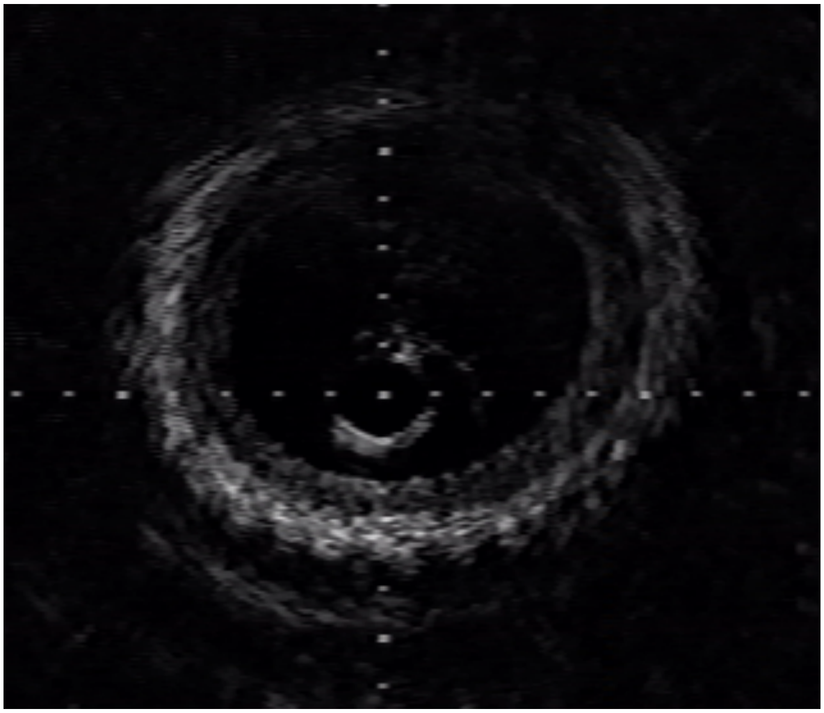


Figure 2.7: Image, prior to median filtering

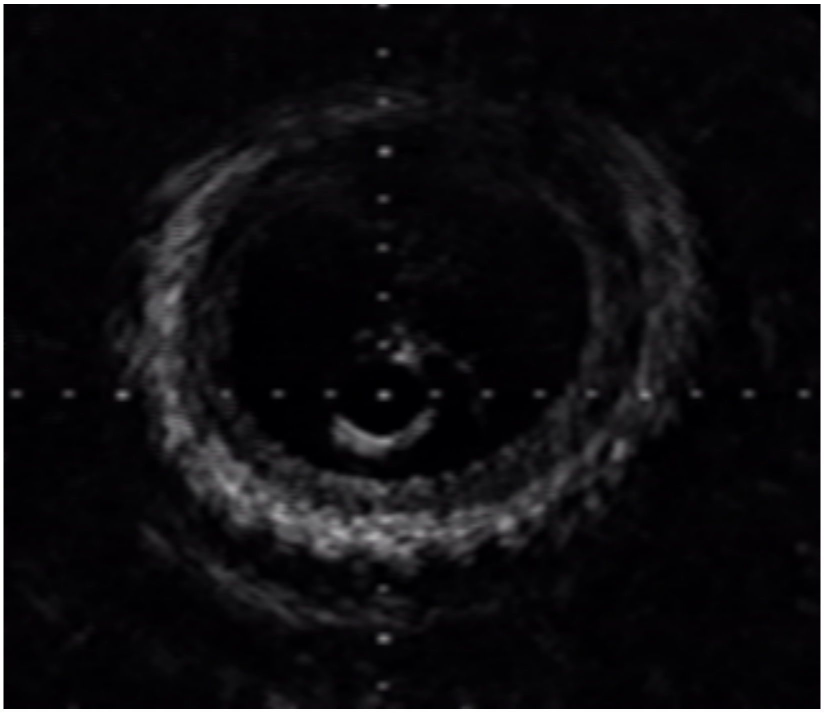


Figure 2.8: Image, after one filtering pass

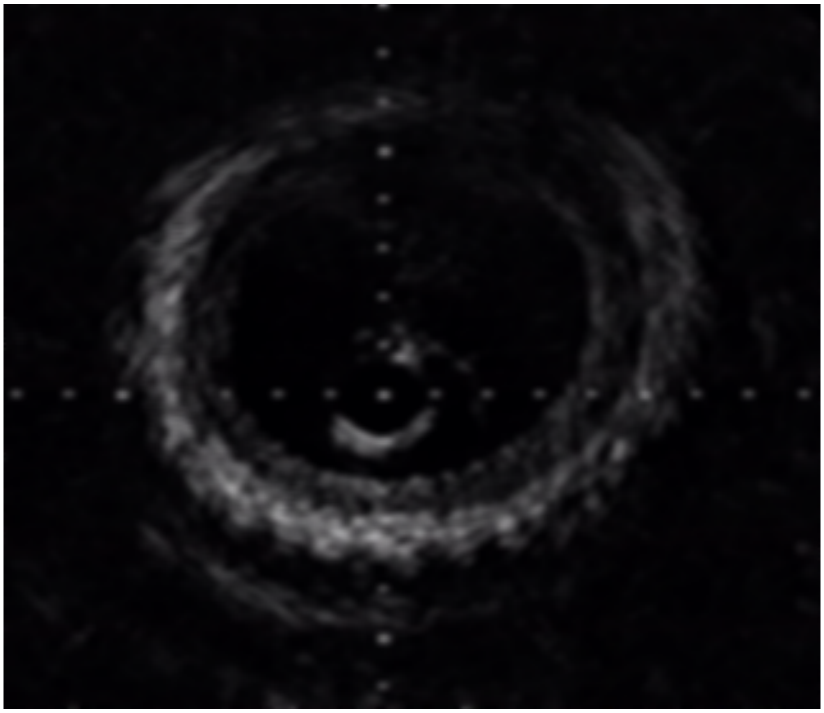


Figure 2.9: Image, after two passes

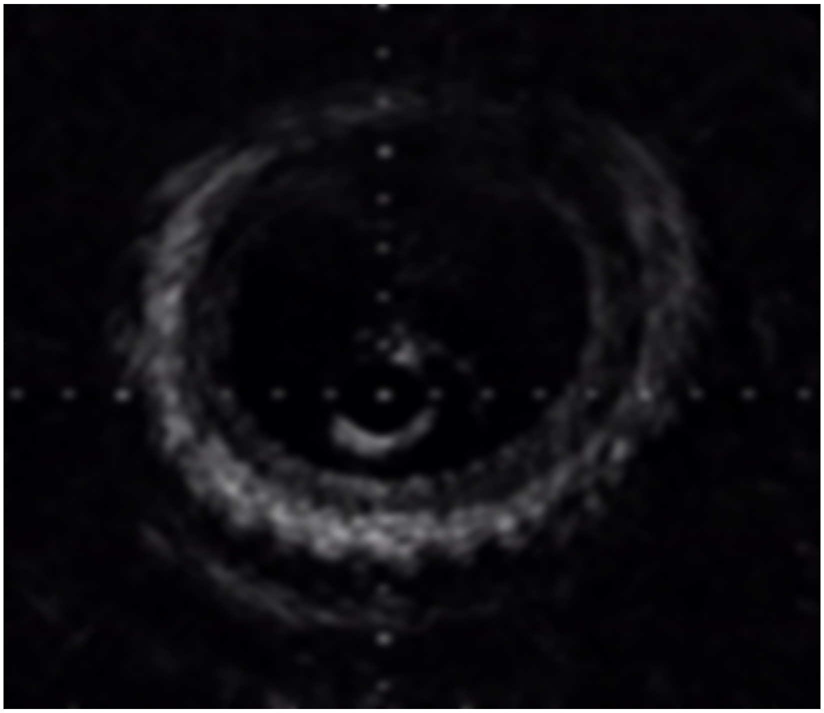
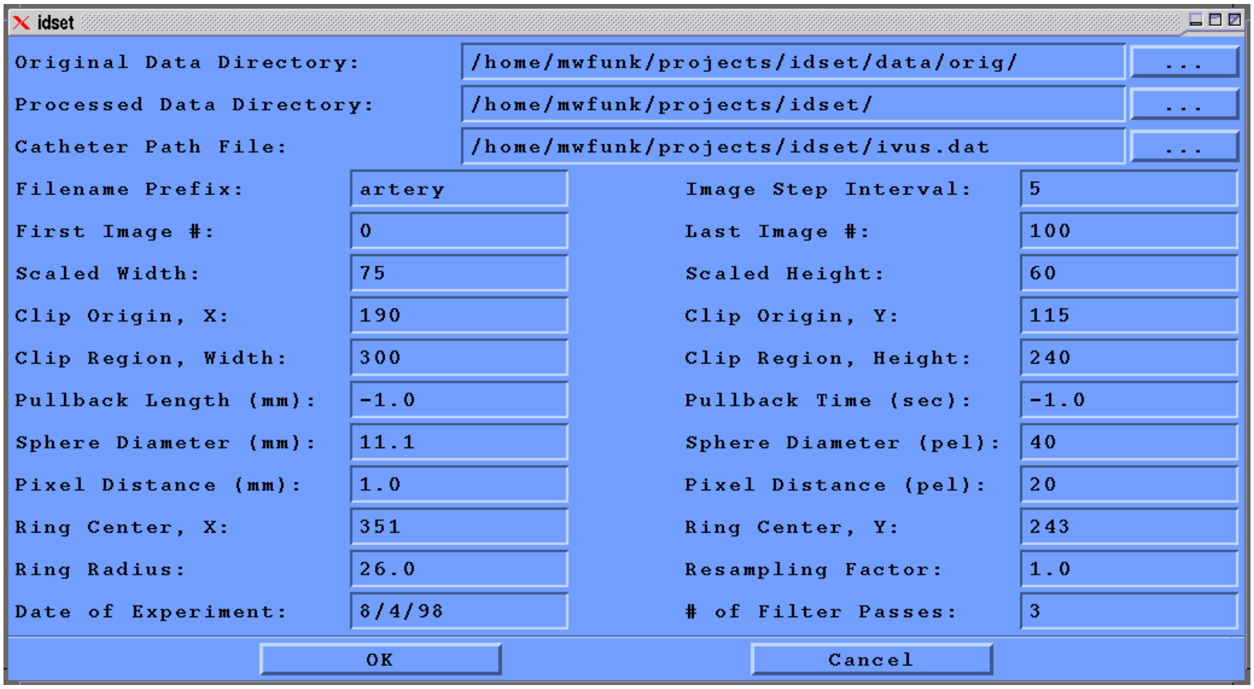


Figure 2.10: Image, after three passes



Original Data Directory: /home/mwfunk/projects/idset/data/orig/
Processed Data Directory: /home/mwfunk/projects/idset/
Catheter Path File: /home/mwfunk/projects/idset/ivus.dat
Filename Prefix: artery
Image Step Interval: 5
First Image #: 0
Last Image #: 100
Scaled Width: 75
Scaled Height: 60
Clip Origin, X: 190
Clip Origin, Y: 115
Clip Region, Width: 300
Clip Region, Height: 240
Pullback Length (mm): -1.0
Pullback Time (sec): -1.0
Sphere Diameter (mm): 11.1
Sphere Diameter (pel): 40
Pixel Distance (mm): 1.0
Pixel Distance (pel): 20
Ring Center, X: 351
Ring Center, Y: 243
Ring Radius: 26.0
Resampling Factor: 1.0
Date of Experiment: 8/4/98
# of Filter Passes: 3

Figure 2.11: Volumetric dataset creation

tion was written which performed almost all of it. The applet, seen in figure 2.11, picks up after the videotape has been digitized (and after the 3D coordinates have been obtained from the X-ray images), but prior to any image processing or manipulation. The applet needs only to be told where to find the digitized images, and where to store its output. It ultimately produces a single "idset" file (for "IVUS discrete set"), which is a file format specifically designed as input for the main visualization application. It contains the volumetric data, as well as the 3D coordinates from the X-ray images, and other miscellaneous fields regarding the data acquisition procedure, such as the date of the experiment. The various fields in the applet each serve one of 3 purposes: they control selection of input files, they set parameters for the image processing pipeline, or they fill in informational fields in the header of the output file. In view of the sheer number of fields available for the user to manipulate, the applet generates reasonable default values for the vast majority of them. In most cases the user only has to manually enter a half-dozen or so values in order to generate an idset. The only other file which must be provided by the user is a simple text file which contains the 3D points obtained from the X-ray images. The points are listed in a simple text file. It is not necessary for the user to normalize the points with respect to the first one (the origin), nor is it necessary for the user to convert the pixel values to millimeters; the applet will do this automatically. The points are merely inserted as is into the idset file; they do not affect the image processing procedure in any way, and in fact are not used until later, when the main visualization application loads up the idset file and performs the 3D reconstruction of the data.

Chapter 3

3D Reconstruction and Visualization

Once the data has been gathered, digitized, sampled, and filtered, 3D reconstruction can begin in earnest. The visualization pipeline for the 3D reconstruction consists of 3 major segments. The first computes a spline (a cubic curve in 3 dimensions) from the points sampled by the X-ray images and applies it to a spatial deformation of the volume. The second involves generating polygonal data from the volumetric, scalar data. The third is really a group of procedures, included not out of necessity but convenience. The third segment consists of a number of filters which boost the performance of the rendering process later on, when interactivity and smooth animation are extremely important.

3.1 Modeling Curvatures

Recall that the location of the catheter tip in 3D space is known at several discrete points along its path, relative to its initial position. These points are sufficient to generate an interpolating spline, which should provide a reasonably accurate approximate of the location of the catheter tip at any point along its path. A Kochanek-Bartels [1] spline is used, to ensure that the generated spline passes through all of the points. Roelandt [3] and Lengyel [2] document much work in the area of mapping IVUS video data to specific points in (3D) space and time; the system used here is far simpler but in many ways functionally equivalent, and sufficient for the task at hand.

Once the spline has been generated, each ultrasound image is placed along it. Each image is placed in such a way that the spline passes through its center, and that the tangent to the spline is coincident with a line passing through the center of the image and perpendicular to its plane. In this way, each pixel in each image now has a definite location in a 3D volume defined by the spline (representing the path of the catheter tip) and the images which are "hung" on it. The position of each image along the length of the spline is given by the image's z-value, that is, its position within the series of images. An image that comes in the middle of the series would be assigned a position halfway along the length of the spline. As far as how this changes the volumetric data is concerned, prior to this operation the data could only be thought of in terms of figure 2.1; that is, a volume defined in terms of a sequence of parallel, coaxial planes. Afterwards, the volumetric data assumes the more realistic form depicted in figure 2.2, in which each point of known data is mapped into our volume on the basis of not just its position within its "frame" and the number of the frame in the sequence, but also on where the catheter was at that time, as well as its orientation.

3.2 Isosurfaces

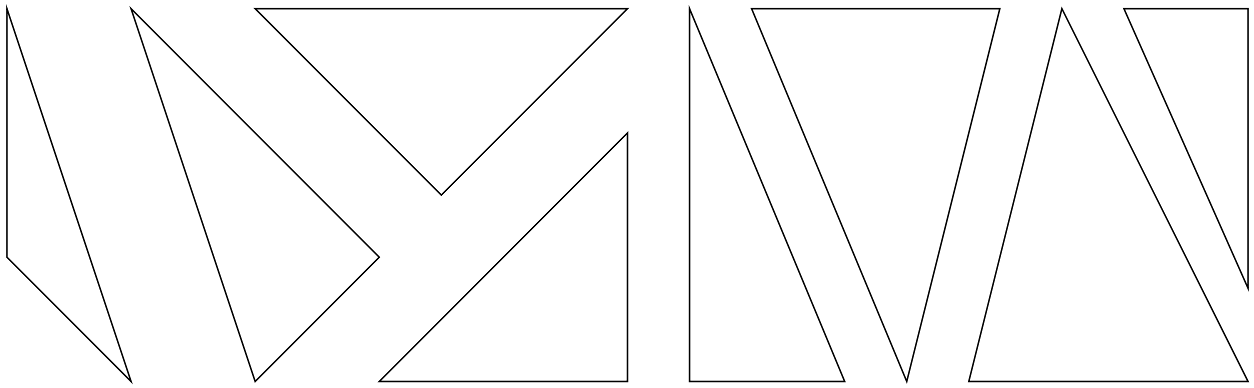
The next step in reconstruction involves transforming the volumetric, scalar data into a polygonal object defined in terms of a relatively small number of vertices. This is achieved by constructing an isosurface from the volumetric data. An isosurface is the 3-dimensional equivalent of a contour line on a topographical map; in the same way that each point on the contour line is at the same elevation, each point on an isosurface has the same scalar value. The simplest method for constructing isosurfaces is the well-known Marching Cubes algorithm [5]; however, given the deformed layout of the volumetric data, a more sophisticated variant is used. Marching Cubes is designed for a uniform grid, and is a special case of a more general family of algorithms. Nonetheless, generating the isosurface is a relatively fast and painless proces. The most difficult part is selecting the value to base the isosurface on; this is a judgement call on the part of the user, and cannot be automated. This value should reflect the threshold between the interior of a blood vessel and the inner wall. While certain values seem to produce better results than others (and are provided as intelligent defaults), in general the ideal value is going to depend on the patient's condition and the ultrasound equipment itself. Under certain circumstances, it may be desirable to specify multiple values and generate multiple surfaces in order to better model a volume, but as of yet this capability does not exist in the application. There is a slider with which the user may control the isosurface value, however.

3.3 Optimization

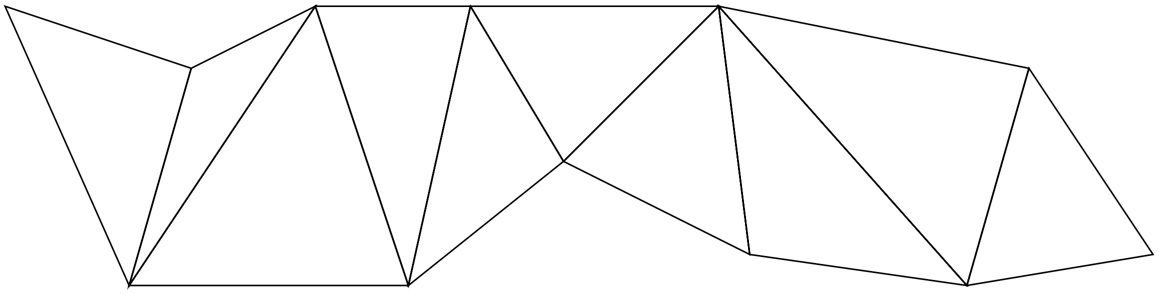
In essence, the process of reconstruction is complete at this point. However, more work needs to be done in order to make the data suitable for interactive viewing and exploration. Specifically, the complexity of the geometry needs to be reduced; the number of polygons used to represent the reconstruction can be brought down, and the format used to store the geometric data can be made more efficient.

First, a process known as decimation is applied to the polygonal data. Decimation looks at groups of triangles; if it finds adjacent triangles within a few degrees of lying on a plane, it replaces the group with a single triangle. The number of polygons eliminated (approximated, really) in this way increases as a function of the tolerance level (the degree to which the orientation of adjacent triangles may differ) of the algorithm. This is a very computationally expensive process, and time consuming compared to the previous filters, but it improves the rendering speed of the application significantly. The algorithm used to construct the isosurface in the first place is not very intelligent, and as a result doesn't check to see if a particular group of triangles it just created don't lie in a plane, and can thus be substituted by a larger (and less complex) polygon. Decimation can reduce the number of vertices needed to specify the isosurface by up to 85%, at least in our own experience, given the datasets we used for testing.

Second, as many triangles as possible are converted into a single triangle strip. A triangle strip is a graphics primitive composed of groups of triangles which share vertices. Figure 3.1 illustrates the difference between ordinary triangles and triangle strips. Ordinarily, specifying the location of three triangles involves specifying 9 vertices, even if some of those vertices are shared. If some of these vertices are shared, then obviously some of this data is redundant and can be expressed in a more compact form, such as a triangle strip. Converting as many triangles as possible into triangle strips reduces the amount of redundant data stored in memory. This improves the rendering speed of the application because fewer vertices means fewer equations for the processor (or graphics hardware) to solve in order to render the isosurface. A number of standard procedures in computer graphics involve per-vertex operations; for example, shading calculations, and transformations (scaling, translation, etc.) By minimizing the number of vertices used to render the surface, we also minimize the number of equations which must be evaluated in order to do the rendering. The final result is rendered by the main visualization application developed for this project, as seen in figure 3.2. Both the decimation algorithm and the algorithm used for generation of triangle strips are described in detail in the VTK documentation [5]; sample implementations of both are freely available in the VTK source distribution.

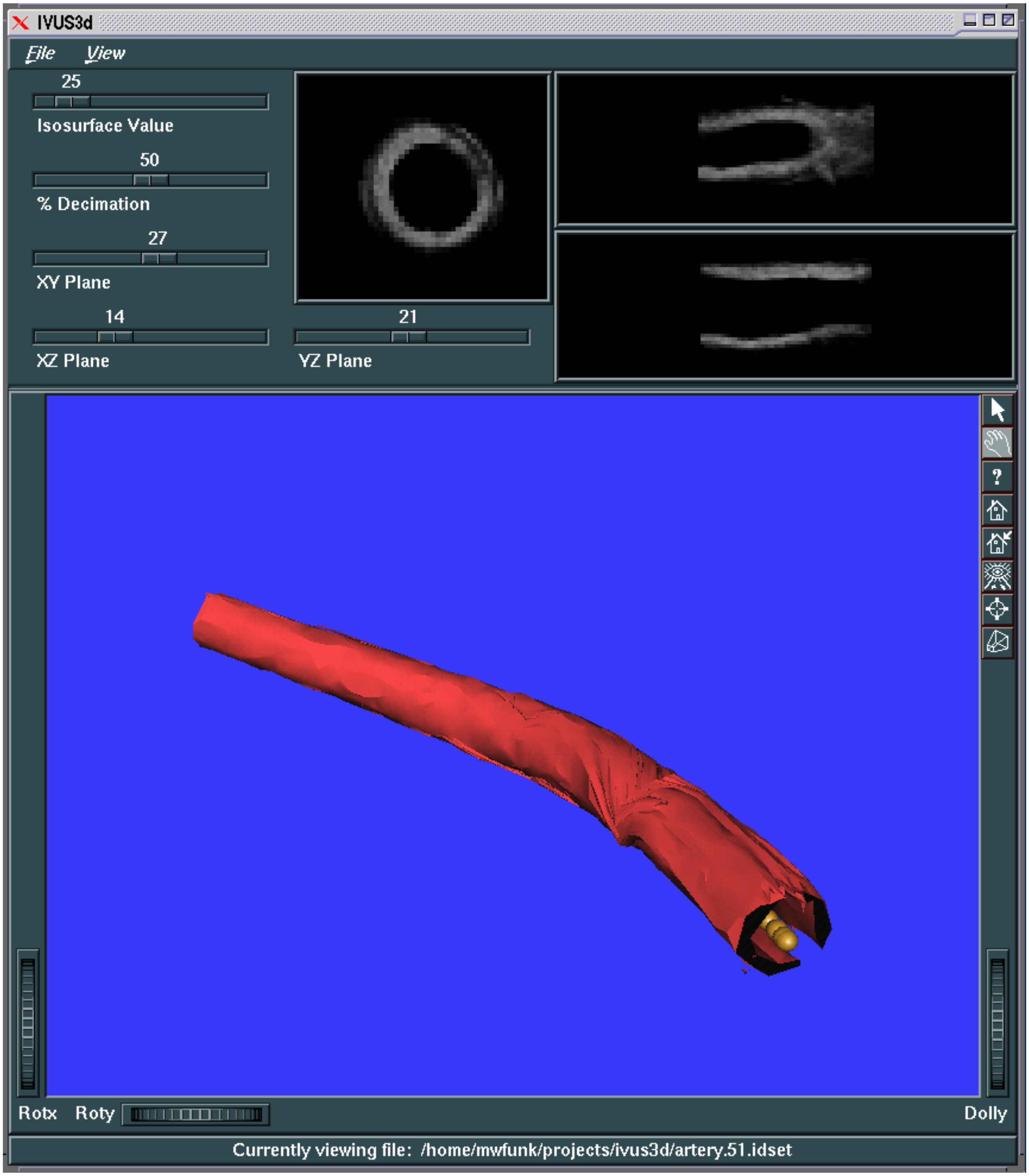


Triangles



A triangle strip

Figure 3.1: Triangles and triangle strips



25
Isosurface Value

50
% Decimation

27
XY Plane

14
XZ Plane

21
YZ Plane

Currently viewing file: /home/mwfunk/projects/ivus3d/artery.51.idset

Figure 3.2: 3D IVUS visualization

Chapter 4

3D IVUS System Description

Having described the procedures involved in data acquisition, and the algorithms used to create the 3D reconstructions of that data, we turn our attention to the details of the development process itself: what hardware and software was used in development and what demands this sort of software places on the underlying platform; also, details regarding the actual interactive visualization application that was created, and what external tools it relies on. The reader is referred to Subramanian [7] for more theoretical background; the remainder of this paper focuses on more implementation-specific issues.

4.1 Hardware

All of the software was developed and run on graphics workstations from Silicon Graphics, Inc. (SGI): primarily R5000-based O2's, as well as an R10000-based Indigo2. For any future incarnation of this program or this visualization system, it is absolutely necessary to have 3D graphics hardware. A large part of the potential utility of this type of system revolves around its interactive use; lack of decent 3D hardware acceleration greatly reduces the level of interactivity of the software, to the point of unusability. If the user has to wait for several minutes just to see the reconstruction from a different angle, then it would be wiser to invest in a system based more on a batch-processing paradigm, with less computationally expensive means for specifiying the desired viewing angle. Another system requirement is sufficient memory...the more the better. It was not unusual for even a small dataset to require 50MB of memory; unfortunately there is no way to reduce this requirement short of decreasing the granularity or the size of the data.

One of the more useful features of the application is the ability to assign an arbitrary alpha value (for transparency) to the material used for the 3D reconstruction. This allows the interior of the reconstruction to be seen, as well as providing a clear view of the plot of the path of the catheter. Having direct support for alpha blending in the graphics hardware is therefore particularly desirable, if this feature is to be commonly used. We had the luxury of having access to an SGI Indigo2 with the High Impact graphics option, which offered excellent support for alpha blending in hardware, to the point of there being no perceptible drop in performance when the transparency feature was activated. This was in stark contrast to the other machines used (mostly O2's), in which the interactivity of the application dropped dramatically due to the machine being forced to perform alpha blending in software.

4.2 Software

The main application was written in C++, chosen both for the power of the language as well as for compatibility with certain libraries. Two libraries in particular were indispensable: Open Inventor [8] and VTK (the Visualization Toolkit) [5]. Both are implemented as C++ class libraries.

VTK was used for number-crunching and "behind the scenes" data processing. VTK includes 3 general types of classes: sources, filters, and sinks. Instances of these classes are strung together in a list to form a visualization pipeline. Sources are objects which produce output, but take no input. They are always at the beginning of the pipeline. They are either going to correspond to things such as an input file, or perhaps one of VTK's advanced data types, created by the program. VTK's built-in data types include classes for volumetric data, and for large polygonal datasets. The initialization of a source is unique to each class (reflecting the wide variety of sources available), but all sources produce a specific type of data as output. All sources have a getOutput() method which return a pointer to their output buffer.

Filters have both input and output. Filters take a specific type of data as input, perform some sort of transformation on that data, and yield the transformed data as output. The output type is not necessarily the same as the input type. An arbitrary number of filters may be strung together in the visualization pipeline, the only requirement being that the data types are consistent with one another at all junctures. Nearly all of the algorithms required by the software were already implemented as VTK filters, freeing up the authors' time to concentrate on 3D reconstruction. In particular, the algorithms used to construct isosurfaces of the data, as well as optimizing the data for fast rendering (decimation, triangle stripping, and so on) were already implemented and readily availabe.

At the end of the visualization pipeline lies a sink object. A sink, as the name suggests, takes an input but produces no output. A sink always terminates a pipeline, and has no getOutput() method. Sinks are generally going to be either writers or renderers. Writers convert their input into files of some appropriate format (for example, VRML or AutoCAD). Renderers use available graphics facilities to render their input onscreen, and optionally provide a user interface with which the user can interact with the rendered objects.

Our software used Open Inventor for all onscreen rendering; the final node in the VTK pipeline was thus neither a writer nor a renderer. It was written specifically for this project; it took polygonal data as input and generated the equivalent data structure for Inventor to render.

Open Inventor was used for the actual onscreen rendering, and much of the user interface. Inventor provides an object-oriented layer of abstraction on top of the native 3D rendering library, OpenGL. OpenGL is a low-level, immediate-mode graphics API, whereas Inventor is very high-level, and is retained-mode. The distinction between immediate and retained graphics API's is this: functions in immediate-mode libraries are generally commands to draw graphics primitives on the screen immediately. It is up to the programmer to keep track of which objects are currently being rendered, what properties are associated with each object, and what the geometric relationships between the various objects are. An immediate-mode API thus provides little more than a thin layer of abstraction on top of the graphics hardware. By contrast, in a retained-mode API, the programmer rarely (if ever) explicitly causes rendering to take place. Rather, the programmer manipulates a database of objects to be drawn. The library itself determines when rendering must be done, and relieves the programmer from having to write the underlying database code himself.

In addition to being an abstraction layer on top of OpenGL, Open Inventor provides a number of useful user-interface components, including viewers for rendered objects. In this respect it is also a layer of abstraction on top of Xt/Motif and the X Window system, in which it is implemented. While finer-grained control over the application could have been achieved by using Xt/Motif and OpenGL directly, Open Inventor saved a considerable amount of time, and no doubt resulted in a much higher-quality package than would have been developed otherwise, due to the quality and utility of its prebuilt components.

Despite Inventor's interface components, there were a number of interface elements that had to be coded from scratch. Fortunately, it proved trivial to include Inventor components within a larger Xt/Motif-based application. Essentially, Inventor provided the rendering areas (as well as some basic mechanisms for user interaction with the rendered objects), while most of the other interface elements (the menubar, sliders, etc.) were hand-coded in Xt-Motif. There were also a small number of rendering areas devoted to displaying the original, 2D data; Inventor was used for creation of the rendering widgets, but the routines which drew into these widgets were coded in straight OpenGL (unfortunately, Inventor has no facilities for dealing directly with two-dimensional data). Nonetheless, Inventor provides hooks into OpenGL in all of its interface components, so integration of Inventor and straight OpenGL was quite straightforward.

4.3 Development Issues

The most pressing issue was the need for optimized performance, both in terms of rendering speed (which is bound by the CPU and the graphics hardware) and the memory footprint. One approach that alleviated both areas involved reducing the amount of geometry that the application had to deal with. The decimation and triangle stripping processes (both available as VTK filters) helped a great deal. While they may have added several seconds to the initial processing time it takes to generate a 3D reconstruction, they improved the interactive performance a great deal. It was judged to be far more important to improve the interactive performance than the startup time, within reason. The user can no doubt tolerate waiting a few extra seconds before being able to view the reconstruction, but if the viewing process itself is sluggish, then the usefulness of the application suffers.

In addition, steps were taken to reduce the memory footprint. By default, each VTK filter retains a copy of its version of the dataset in memory, so that if the visualization pipeline has to rerender, only those filters whose input or parameters have been modified need be run again. Since this was much less of an issue than memory consumption, VTK was configured so that the various nodes in the visualization pipeline did not cache a copy of the data upon execution; rather, the data was passed along to the next filter in the pipeline, then discarded.

4.4 The Application

The application itself consists of two parts, a 3D viewing area, and a collection of 2D viewing widgets. The 3D viewing area is where the 3D reconstruction gets rendered, and contains controls allowing the user to change the camera's location and angle, as well as modify the rendering process itself (designating a default viewing angle, switching to wireframe mode, among many other useful settings). A new dataset is opened via a file selection dialog box reached through the menubar. Two sliders exist for altering the parameters to the isosurfacing algorithm and the decimation filter; they change the isovalue and the target percentage for decimation, respectively. Figures 4.1 through 4.3 show reconstructions of the same dataset using different isovalues.

Once a dataset has been opened, additional dialogs become available (again, through the menubar), allowing the user to modify such things as the material properties of the reconstruction (most notably the degree of transparency) and the specification of a clipping volume, so that only a subset of the geometry may be viewed. Figure 4.4 shows the application with the material editor dialog open; figure 4.5 shows the results of setting a higher alpha value in order to obtain a transparent reconstruction. The complete plot of the path is clearly visible through the transparent material. Figure 4.6 shows the dialog box used to specify the clip volume; through this dialog box it is possible to define a subset of the volumetric data to be used for generation of the reconstruction. Figure 4.7 shows one application of this technique: the top half of the artery appears to have been removed, revealing much more of the inner structure (and providing a clear view of the catheter path, without having to resort to transparency).

In addition to the 3D canvas and its associated controls, there are a group of 3 2D drawing widgets, each of which show a different cross-section of the volumetric data (they show the (x, y), (x, z), and (y, z) planes). There is an independant variable associated with

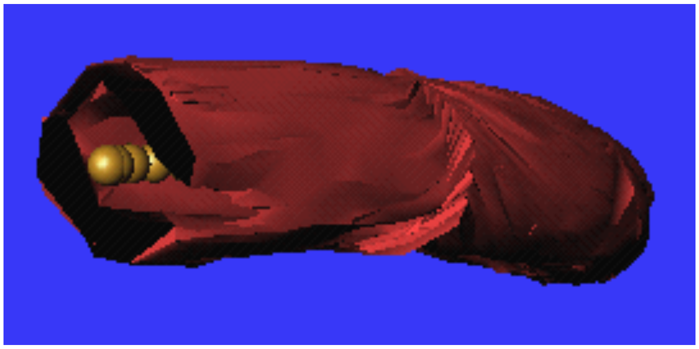


Figure 4.1: IVUS 3D reconstruction, isovalue 9

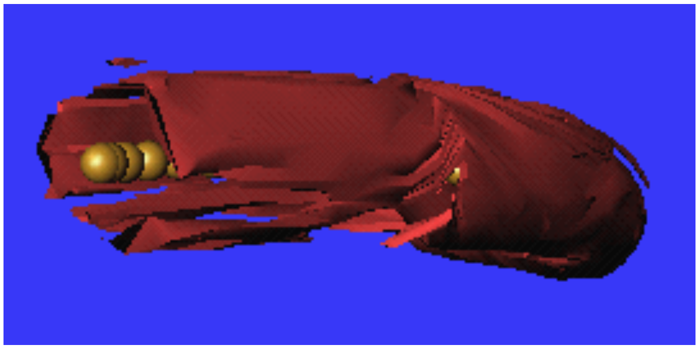


Figure 4.2: IVUS 3D reconstruction, isovalue 45

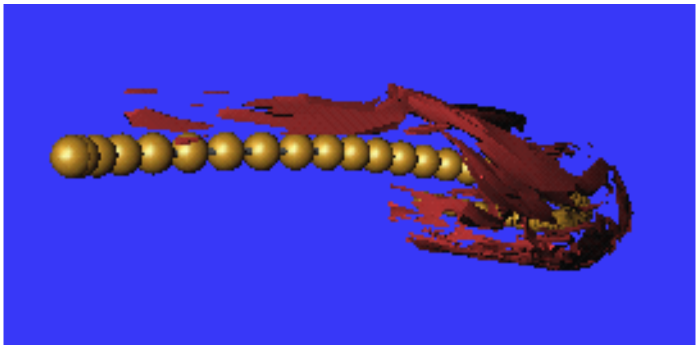
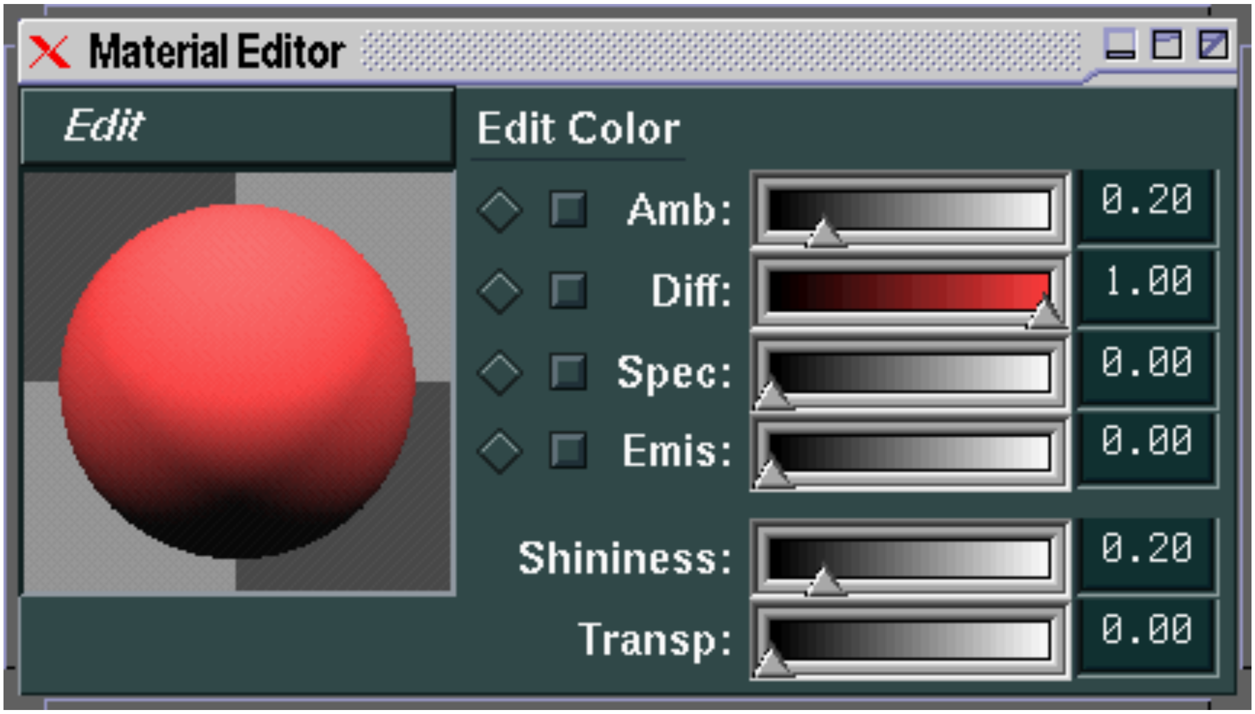


Figure 4.3: IVUS 3D reconstruction, isovalue 117



Amb: 0.20
Diff: 1.00
Spec: 0.00
Emis: 0.00
Shininess: 0.20
Transp: 0.00

Figure 4.4: The material properties editor dialog

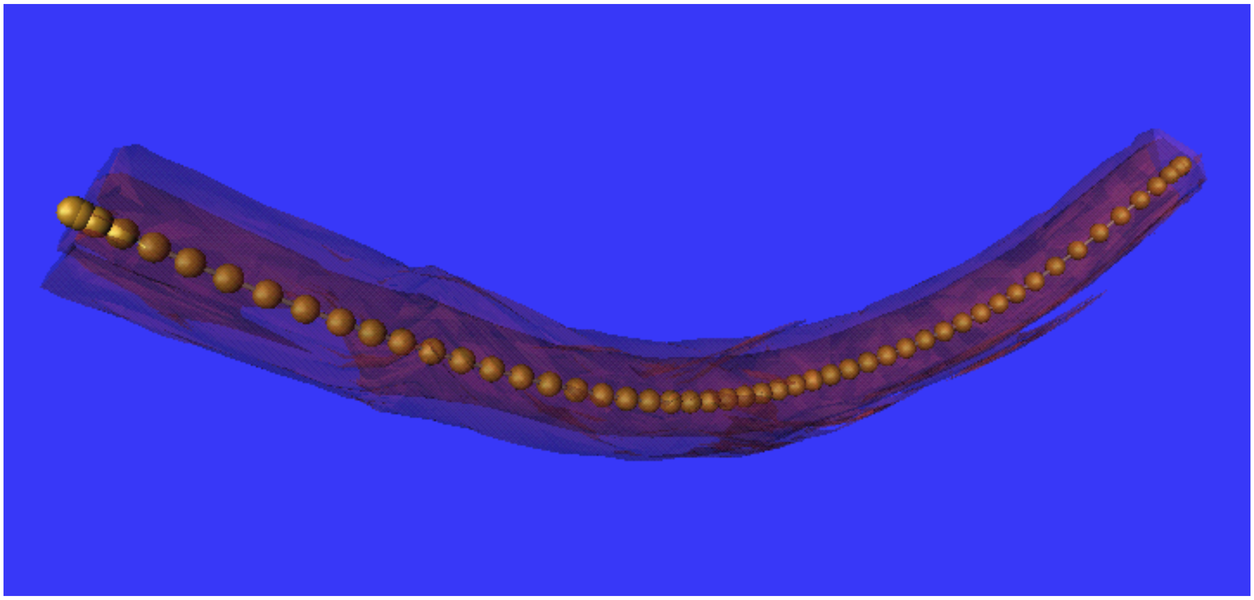
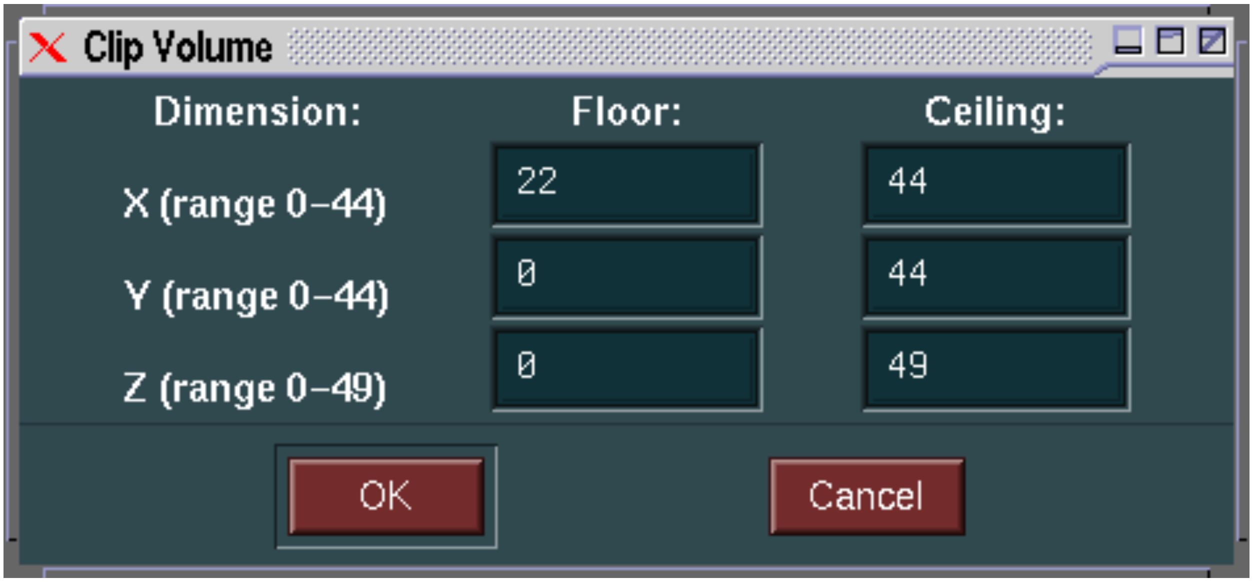


Figure 4.5: Using a transparent material for reconstruction

each one, which is set using another slider. For example, for the (x, y) view there is a slider which allows the user to set the z value of the plane being viewed. The data displayed in these widgets has not been modified by the curvature deformations, and as a result the (x, y) view directly corresponds to the sequence of images generated by the ultrasound probe, after clipping and smoothing, but prior to any other processing. The (x, z) and (y, z) views offer perspectives on the original data not previously available. The reasoning for providing both 3D and 2D views is based on the belief that no one form of visualization is ideal, and that perhaps the best solution in the face of uncertainty is to place as many tools as possible in the hands of the analyst. Certain features of the data might not be obvious given the 3D reconstruction alone, whereas some observations might only be made given exposure to both formats together.



Clip Volume

Dimension:	Floor:	Ceiling:
X (range 0–44)	22	44
Y (range 0–44)	0	44
Z (range 0–49)	0	49

Figure 4.6: The clip volume dialog

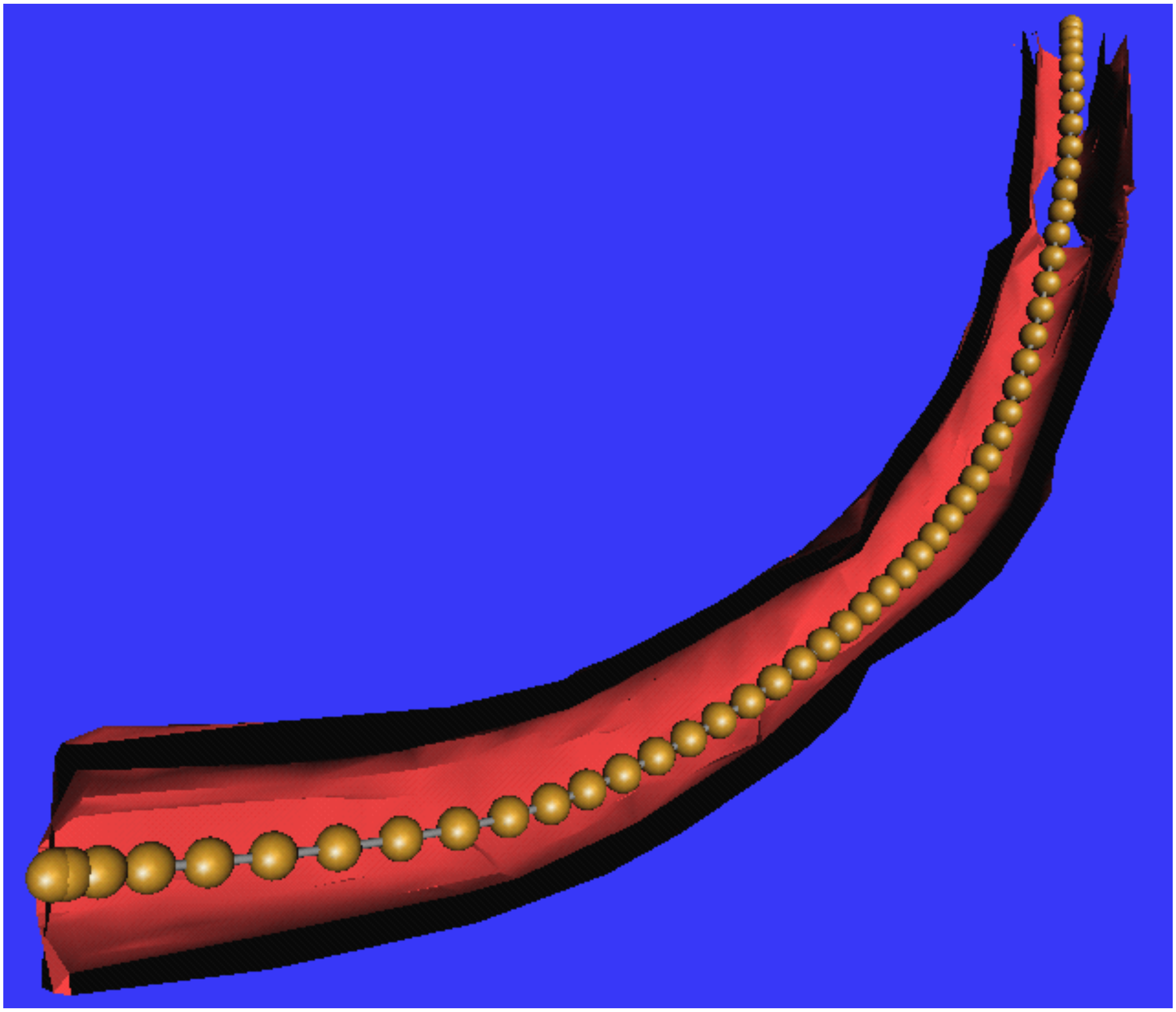


Figure 4.7: Clipping the top half of the volumetric data

Chapter 5

Future Directions of Development

Work is currently underway to minimize the amount of time spent in the digitization process, by eliminating the digitization phase of the data acquisition procedure; it should be possible to directly connect the ultrasound equipment to a digital storage device. This would eliminate what is perhaps the most time-consuming step of the process, digitization of the images from SVHS. In addition, some preliminary research is being done on the viability of performing much of the image processing on a video input in realtime. If this is found to be feasible, it may be possible to eliminate all intermediate phases of data processing between the gathering of IVUS data and the generation of the 3D reconstruction, as a program could be written which would take its input directly from the probe itself rather than from an intermediate, static form of stored data (such as the idset files currently used). Even if this proves to be impossible with current technology, ultimately this is the direction such systems will most likely go in the future.

Another area of potential research is in proper segmentation and analysis of the reconstruction; in the context of intravascular ultrasound, this means being able to identify key coronary structures automatically. In particular, having the ability to automatically identify aberrations and unusual conditions could greatly assist medical specialists in both diagnosis of problems and in the planning of direct interventions. The work of Rosenfield, et al [4] contains some preliminary thoughts on segmentation, as well as possible applications of the technology.

Also under consideration is the eventual usage of the Java language and environment, as a means of providing portability and greater maintainability and extensibility for the code itself. Any decision to go in this direction is entirely dependant on available 3D API's, and their performance and availability on target platforms. Also important is the performance of current and future JVM's, particularly with regard to JIT compilation. The application developed for this project is quite resource-hungry and bound by CPU speed; a transition to Java will only be worthwhile when we are certain that performance hits in this area can be minimized. For the time being, any transition to a Java-based environment is on hold until these issues can be addressed.

Chapter 6

Conclusion

In terms of providing a complete system, the single greatest hurdle to overcome is providing a more streamlined system for data acquisition. If such visualization packages are to become a standard tool in the future, the data acquisition procedure should take no more than an hour or two, and should be automated (where possible) to the point that most of the procedure can be done without human supervision. Ideally, the gathering of the data via an ultrasound probe (and a corresponding, integrated means of tracking the probe in 3D space) should take only a few seconds, with the bulk of the time spent performing the filtering and smoothing procedures on the data. Aside from initially setting up a small number of parameters, the latter process can be entirely automated and left to run unattended. The initial storage device should be digital; removing the need to digitize a videotape would remove one more link from the chain, and reduce the need for human labor and supervision, as well as improve the margin of error.

One area in which progress was made was in the image processing stage, after digitization and prior to reconstruction. Previously, this had to be done more or less by hand, using a number of scripts specifically written for the project. The procedure was insufficiently generalized and standardized, and as a result could take as long as an hour or more. The dataset creation program shown in figure 2.11 reduced this to a couple of minutes, at most. It also completely automated the procedure, so that once the initial parameters were set, no

Size of dataset, in frames:	32	43	51
Processing time:	1:04	1:22	1:43

Table 6.1: Timings for dataset creation

additional interaction was required. Timings for three sample datasets are given in table 6.1.

As far as the software itself is concerned, the three most pressing issues are interactivity, correctness, and analytical utility. Barring dramatic improvements in processing power and available memory, it will be highly desirable to speed up the interactive rendering engine and reduce the memory consumption even more than it already has been. For this sort of resource-intensive application, there really is no point at which "enough" optimization has been performed; even though acceptable performance has been achieved given the datasets used for testing, more optimization means that it becomes practical to use larger and more complicated datasets. There really is no ceiling on how large a dataset one could conceivably want to visualize, hence further optimization will always be a useful area to explore. Secondly, correctness. A significant amount of processing is performed on the original data; common sense dictates that the longer one's visualization pipeline is, the greater the margin of error becomes, and the less relevant (or even potentially confusing) the output becomes. Work needs to be done, starting with very simplistic, cut-and-dried test cases (fiberglass tubing, hoops, and so on), to determine how accurate the reconstruction is. Such work might also benefit those researching the instruments themselves, developing better catheters or transducers, or improving the methodologies involved with the usage of existing equipment. Finally, analytical utility. This means, increasing the amount of information that can be derived from the data. One area of discussion has revolved around giving the user the ability to perform measurements in physical units (such as millimeters) on arbitrary features found in the reconstruction. For example, the user may wish to measure the diameter of the blood vessel at various points, in order to help diagnose medical conditions, or weigh the benefits and dangers of various forms of intervention.

Appendix A

The Visualization Application

A.1 Usage

The primary focus of development was on the interactive visualization application, which was named “ivus3d”. This program takes as input a single file containing the filtered volumetric data from the ultrasound run. The file format was developed specifically for this application; the files typically are given an “idset” extension (for **i**vus **d**iscrete **set**). In addition to the volumetric data, the files contain the set of points corresponding to the catheter path for that particular dataset. There are also some miscellaneous fields in the file header containing useful information about the dataset, such as the dimensions (in voxels) of the volume, the date of the experiment, and so on.

The application can be seen in figure 3.2. When the program first starts up, there is no dataset loaded. To load an idset file, choose the “Open” option from the File menu. A file selection dialog box will pop up, allowing the user to select an idset file to load. After a file has been selected, the application will generate an isosurface from that dataset and display it in the main rendering area. By clicking and dragging on the rendering area, the user may move the camera to any arbitrary position and orientation around the reconstruction.

Right-clicking on the rendering area will bring up a menu which contains a number of options controlling how the isosurface is rendered, including wireframe and hidden-line modes, and switching between parallel and perspective viewing modes.

There are two user-configurable paramters which control the reconstruction process itself. These are the isovalue used to construct the isosurface, and the target percentage for the decimation procedure. Both may be set using slider widgets located in the upper left-hand corner of the application's interface. Changing the target percentage for decimation means changing what percentage of the the original polygons the program attempts to eliminate. This doesn't mean that that percentage of polygons will actually be eliminated; it only means that the decimation filter will relax its constraints until they reach a point where it estimates that that many polygons will be eliminated.

Changing either the isovalue or the target decimation percentage prior to the loading of a dataset will simply alter the initial values for these parameters when a dataset is finally loaded. If a dataset has already been loaded when the change is made, the reconstruction will be redone using the new parameters.

In addition to the main rendering area are three smaller rendering areas in the top half of the window. These areas are used to display two-dimensional data, namely orthogonal cross-sections of the volumetric data. For each of these 2D areas, there is a corresponding slider widget which allows the user to select which cross-section to view. For example, the leftmost rendering area is used to display cross-sections which are perpendicular to the z-axis. A slider allows the user to select a z value, and the corresponding 2D area will display the x-y cross-section located at that point on the z-axis. The other two areas display x-z and y-z planes, with sliders allowing the user to select y and x values, respectively. The x-y view is especially useful as it is corresponds to the original frames of video data from the ultrasound probe. Flipping through a sequence of x-y planes is analogous to flipping through the sequence of video frames generated by the probe. However, there is a difference: the images displayed in the 2D rendering areas have already had the various image processing filters applied to them, and will not be identical to the original video data. The 2D rendering areas are thus most useful for getting a sense of the structure of the volumetric data, and for evaluating the effectiveness of the image processing techniques used.

Finally, a number of features may be accessed through the two menus on the menubar. The first menu, the “File” menu, contains three entries: Open, Save, and Quit. Selecting Open brings up the file selection box, so that the user can select an idset to load. Selecting Quit exits the application, after requesting confirmation from the user via another dialog box. Selecting Save will write out a file in the Inventor format, corresponding to the current 3D reconstruction. If no dataset has been loaded (and hence no 3D reconstruction has yet been generated), the Save item is greyed-out and unavailable. After selecting Save, the user is presented with a dialog box, which allows the user to specify a filename and which directory to place it in. By default, the current working directory is used, and a default filename is constructed by replacing the extension of the idset file with “iv”, the standard extension for Open Inventor files. Inventor files may be viewed using a number of other applications, including the “ivview” program from SGI, and a Netscape plugin. In addition, it is trivial to write Inventor-based programs which read in data from a file in this format.

The second menu is the “View” menu, which contains two items. Both of these items are greyed-out if the user hasn’t already loaded up a dataset. This menu contains two entries, Material Editor and Clip Volume. The Material Editor entry will bring up a dialog box which allows the user to modify the properties of the material used for the reconstruction. This includes various color-related parameters, but the most useful function of this dialog is the ability to change the level of transparency of the material. When the reconstruction is initially rendered, it is fully opaque. However, by giving it a nonzero level of transparency, it becomes possible to examine the interior geometry of the reconstruction. In addition, the plot of the catheter path becomes fully visible. The tubes and cylinders which make up the plot of the path are not affected by the material editor, only the reconstruction itself. The material editor dialog may be seen in figure 4.4.

The other item on the “View” menu is Clip Volume. Selecting this item brings up yet another dialog box, which allows the user to select a subset of the volumetric data to use. Any combination of x, y, and z ranges may be used. This is useful in case one wants to examine the interior of the reconstruction without having to resort to transparency. The dialog box may be seen in figure 4.6; figure 4.7 is an example of a clipped reconstruction.

A.2 Tools, Languages, and Libraries

Aside from the application used to generate the idset files, no additional external tools are required to run the software. However, ivus3d does have the ability to save the 3D reconstructions in the Open Inventor 2.0 file format. This allows the reconstructions to be read in directly by any other tools which can read this format. This includes a number of programs which ship with Irix, including a Netscape plugin.

The implementation language for ivus3d was C++. C++ was chosen because we wanted to use certain class libraries for the project, namely VTK and Open Inventor. In addition, we felt that C++ offered better facilities for modular programming than the alternatives, without sacrificing performance.

The program relies on a number of external libraries for much of its functionality. As was noted earlier, the actual reconstruction is performed using classes from VTK. A custom VTK class was written which exported VTK’s internal format for storing geometric data to the functionally equivalent format used by Open Inventor. This allowed us to use Open Inventor to perform the actual rendering. This was desirable for three reasons. First of all, Open Inventor has a highly optimized rendering engine, which gave demonstrably better performance than the corresponding renderer that ships with VTK. Secondly, it is easier to integrate straight OpenGL code with Open Inventor than it is with VTK. Finally, Open Inventor included a large number of very high-level user interface components which proved useful.

In addition to Open Inventor and VTK, Motif was used to create much of the user interface. Open Inventor provided essential UI components such as the viewer widget, which displays the reconstruction, and allows the user to interact with it in a number of different ways. However, Open Inventor does not provide simpler, more general-purpose components like menubars or sliders. This was where Motif proved useful. Also, since Open Inventor's widgets themselves were implemented using Motif, it was very easy to integrate the two toolkits to provide a single, uniform UI.

A.3 Source Code Roadmap

The source code is composed of 5 source files and a number of headers. One of the files, `main.cc`, contains the `main()` function for the program. It is very small; it does nothing more than instantiate the Interface class (described in the next paragraph), and enter the main event loop of the program.

The program uses a very simple object model (if it could even be called that). One class, Interface, encapsulates all of the code related to the user interface and the onscreen rendering. To the greatest degree possible, all Motif and Open Inventor code is confined to this class, and no VTK code will be found within. The other class is the (perhaps poorly named) Data class, which contains the VTK pipeline. All VTK code is within this class. No UI code is contained in this class, with one exception. The final node in the VTK pipeline converts VTK's geometric data structures to the format used by Open Inventor. A cleaner implementation might place the VTK/Inventor conversion class outside of Data, and somehow place it in between Data and Interface. The Data and Interface classes are implemented in the files `data.cc` and `interface.cc`, respectively. Corresponding header files (`data.h` and `interface.h`) contain their class declarations.

When the program starts up, it creates one instance of the Interface class. When an idset file is loaded up through the interface, an instance of the Data class is created. The idea is that each dataset would have an associated instance of the Data class. In the current implementation, there is only one dataset available at time. However, it is written in such a way that the program could be extended to allow multiple datasets to be open at once, with one instance of Data for each. This would allow for a richer user environment; for example, multiple datasets could be open at once, each in a different top level window. Alternately, multiple instances of the same dataset could be open at once, each with different parameters for its visualization pipeline (such as different isovalues).

Manipulation of the visualization pipeline's parameters is done by calling public methods in the Data class, rather than by directly manipulating the VTK nodes. This allows us to have a clean seperation between the UI code and the visualization pipeline, to the degree that we could do away with VTK altogether, and reimplement its functionality using custom code or another toolkit, and not require a single modification to the UI code. The same applies to the interface; it could be completely rewritten, if need be, and we wouldn't have to worry about potential side effects within the visualization pipeline.

The Data class contains instances of the various classes that form the nodes of VTK's visualization pipeline. In addition the the VTK classes, two custom classes were written for use in the pipeline. The first class, IvusDiscreteSet, is an object-oriented wrapper around the idset files. This class is declared in the file `ivusDiscreteSet.h` and implemented in the file `ivusDiscreteSet.cc`. An instance of this class forms the beginning (or source, in VTK terminology) of the visualization pipeline. The constructor for the IvusDiscreteSet class takes a filename as a parameter; this should be the name of an idset file to read into memory. The class serves two functions, aside from being a reader for the idset file format. First, it marshals the data into a form that can be used by VTK. The VTK pipeline gets its initial data from this class. Secondly, it generates a spline from the points stored in the idset file (corresponding to the catheter path), and manipulates the volumetric data accordingly.

This class implements a number of other methods for examining and manipulating idset files and their data, but these two features are by far the most important for the program.

The visualization pipeline is implemented as a linked list of nodes, each node an instance of a VTK class. The pipeline contains three kinds of nodes: sources, filters, and sinks. A source is at the beginning of the pipeline; it generates an output, but does not accept another node as input. In our pipeline, the IvusDiscreteSet class is the source. Although it is not a proper VTK node, it creates one that can be used as a starting point.

Following the source are an arbitrary number of filters, chained together in sequence. A filter has both an input and an output; a pointer to a VTK node may be set as an input, and it exports a pointer to its output, which will be another VTK node (not necessarily of the same data type). In ivus3d, all of the required filters were available as part of VTK. It was not necessary to write additional filters. The filters include an instance of the vtkContourFilter class, which generates an isosurface from scalar, volumetric data. Three other filters were used. Two of them are for optimization: vtkDecimate and vtkStripper. These perform the decimation procedure and the triangle stripping procedure described in section 3.3. The third filter is an instance of vtkPolyDataNormals. This filter generates normals for polygonal data, enabling a much higher quality rendering of the reconstruction.

The final node in the pipeline is a sink object; a sink takes a node as input, but does not produce an output. A sink forms the end of the pipeline. A custom VTK node was written for use as a sink: vtkIVSink. This node takes as input polygonal data in VTK's format, and transforms it into the format used by Open Inventor. This is the only Open Inventor code to be found outside of the Interface class. When the application renders the reconstruction, it calls a public method in the Data class which returns a pointer to the Open Inventor data generated by the vtkIVSink node. This data is then passed to the rendering widget inside the Interface class. The vtkIVSink class is declared in the file `vtkIVSink.h` and implemented in the file `vtkIVSink.cc`.

A.4 Implementation Notes

The object model is rather minimal, and rather awkward. It does, however, perform the task for which it was created: to segregate the interface code and the rendering code from the visualization pipeline. While it may become desirable in the future to create a new object model from scratch, this fundamental relationship should be maintained. In the early stages of development, intermingling interface and pipeline code was the single greatest source of bugs, extensibility problems, and “spaghetti code”.

Appendix B

The Dataset Generator

B.1 Usage

In addition to the interactive visualization program, a helper program was written to automate the process of creating idset files from the raw ultrasound data. This program assumes that the video data has already been digitized, and that a simple text file has been created that contains the points on the catheter path. Both the digitized video data and the text file are needed to create an idset. The interface consists of a number of text entry widgets, almost all of which initially contain suitable default values. The program's interface may be seen in figure 2.11.

Prior to running the program, the video data should have already been digitized. The program expects that the digitized images are all in the same directory, and are named consistently. The filenames for the digitized images should start with a common prefix, followed by a number denoting the image's position in the sequence (with no leading zeroes), follwed by an *. rgb* extension. The files should all be in SGI's *rgb* format, although support for more file formats could easily be added in the future. For example, a subset of the files for a given dataset might be named `artery1.rgb`, `artery2.rgb`, `artery3.rgb`, and so on. Each file corresponds to a single frame of the original video data.

In addition, a text file should have been created containing the sequence of points in the catheter path derived from the X-ray data. The file should contain one line of text for each point; each point is described by 3 decimal integers, seperated by whitespace (spaces and/or tabs). The program takes care of translating each point relative to the first, so it is perfectly acceptable to use the raw pixel counts from the X-ray images.

Out of all of the text entry fields in the application, it is only required that the user fill in the first three. Reasonable default values are provided for the rest of the fields, which may of course be changed by the user. The first field specifies the directory in which to find the digitized images. The second field specifies the directory in which to place the generated idset. The third field specifies the location of the text file containing the catheter path information. In addition, the user will probably want to change the values for the fourth through seventh fields; these specify the image file prefix, the interval to skip between files, the number of the first file, and the number of the last file, respectively. For example, specifying “artery” for the prefix, “5” for the interval, “20” for the starting number, and “80” for the last number will result in an idset being created using every fifth file in the sequence from `artery20.rgb` through `artery80.rgb`.

The other fields control the image processing parameters and the contents of the data fields in the generated idset file’s header. Examples of image processing parameters include how many times (passes) should the median filtering be performed on each image, and what dimensions should each image be scaled to prior to inclusion in the idset. An example of an idset file header field is the date of the original experiment; the program uses the current date by default.

In addition, miscellaneous fields inform the program of such things as the total distance covered during pullback, and the duration of pullback in seconds. This enables the program to determine the speed at which the catheter was moving, and helps to impose physical units (such as millimeters) on the data. Without such information, all of the geometric data is relative, and is not directly related to standard units of measure.

After the required information is entered, the user simply clicks on the "OK" button, and the idset is created. For most of the datasets we used, the process of idset creation took no more than a minute or two (see table 6.1 for specific timings). The idset file may then be loaded into the main visualization application for 3D reconstruction and data exploration.

B.2 Tools, Languages, and Libraries

This program is dependant on one external program: sgitopnm, from the NetPBM package. NetPBM is a collection of standalone programs which operate on image files, doing both file format conversion and image processing. The sgitopnm program converts SGI `.rgb` files to the `.ppm` (Portable Pixmap) format native to NetPBM. The digitized images are initially available as `.rgb` files, and are converted to `.ppm` files because the file format is very simplistic and easy to manipulate. The dependancy on this executable should probably be removed in future development; it's functionality is easily duplicated within the program. Alternately, the program could be made to operate directly on `.rgb` files without too much difficulty.

The code was written in C. The application was small (it is easily contained within a single file) and it didn't require any libraries other than Motif, so it was decided that any sort of object model or class hierarchy would be overkill.

B.3 Source Code Roadmap

The source code for the program is contained within a single file, `idset.c`. The bulk of the code constructs the label and text entry widgets. The entire functionality of the program is contained within a single function, `ok_but_cb()`, which gets called when the user clicks on the "OK" button. This function contains the entire image processing pipeline, the parameters of which are set via the text entry widgets. The function obtains the values of these parameters by directly examining the strings contained within the entry widgets.

B.4 Implementation Notes

The current form of the application assumes that the images have already been digitized, and are sitting in an accessible directory on the filesystem. This still requires the user to manually retrieve the digitized images from the Diskus, and to convert those images from the raw YUV format to SGI RGB files. A possible future enhancement might retrieve the files directly from the Diskus, saving both time and storage space.

The application was designed to resemble a dialog box. It would be fairly easy to take the code, place the interface inside of a Motif dialog box rather than a top level window, and make that dialog accessible from the main visualization application. This would make the main application more of an integrated tool, and merge the codebases for the two applications.

The technique of masking off the ring artifact left by the catheter can occasionally produce negative side effects: if the artery is somewhat lopsided, or is not centered in the image, then the masking process can sometimes mask off part of the artery, rather than just the artifact. This introduces inaccuracies into the reconstruction; in extreme cases the reconstructed artery looks like it has holes in it, or strips removed from it lengthwise. Better techniques for dealing with the artifact are needed. Ideally, better equipment would be developed that would eliminate the artifact altogether.

Bibliography

[1] D.H.U. Kochanek and R.H. Bartels. Interpolating splines with local tension, continuity, and bias control. *Computer Graphics*, 18(3), July 1984.

[2] J. Lengyel, Donald P. Greenberg, and Richard Popp. Time-dependent three-dimensional intravascular ultrasound. In *Proceedings of SIGGRAPH '95*, 1995.

[3] J.R. Roelandt, C. di Mario, N.G. Pandian, L. Wenguang, D. Keane, C.J. Slager, P.J. de Feyter, and P.W. Serruys. Three-dimensional reconstruction of intracoronary ultrasound images. *Circulation*, 90(2), August 1994.

[4] K. Rosenfield, D.W. Losordo, K. Ramaswamy, J.O. Pastore, R.E. Langevin, S. Razvi, B.D. Kosowsky, and J.M. Isner. Three-dimensional reconstruction of human coronary and peripheral arteries from images recorded during two-dimensional intravascular utrasound examination. *Circulation*, 84(5), November 1991.

[5] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit.* Prentice Hall, second edition, 1998.

[6] F.G. St.Goar, F.J. Pinto, E.L. Alderman, H.A. Valantine, J.S. Schroeder, S.Z. Gao, E.B. Stinson, and R.L. Popp. Intracoronary ultrasound in cardiac transplant recipients. *Circulation*, 85(3), March 1992.

[7] K.R. Subramanian. Accurate 3d reconstruction of curved coronary vessels from intravascular ultrasound images. University of North Carolina at Charlotte, Department of Computer Science.

[8] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics With Open Inventor, Release 2.* Addison-Wesley, first edition, 1994.

[9] P.G. Yock, E.L. Johnson, and D.T. Linker. Intravascular ultrasound: Development and clinical potential. *American Journal of Cardiac Imaging*, 2(3), September 1988.

[10] Xinshi S. Zhou. Intravascular ultrasound image processing and 3d reconstruction. Master's thesis, University of North Carolina at Charlotte, 1995.