An Implementation of Piecewise-Linear Interpolation for 3D Slice Reconstruction

Gail Carmichael

April 25, 2008

1 Introduction

In the world of medical computing, 3D reconstruction of slice data is a common topic. A slice is a single cross-section image obtained by CT or MRI devices, and can be processed to obtain polygonal data describing the material to be analyzed. For example, if images are taken of a heart, contours around the cross sections might be found and saved as point data for further processing. An example of this is shown in Figure 1. It is then possible to reconstruct these polygonal contours into a 3D object by connecting the appropriate pieces together.

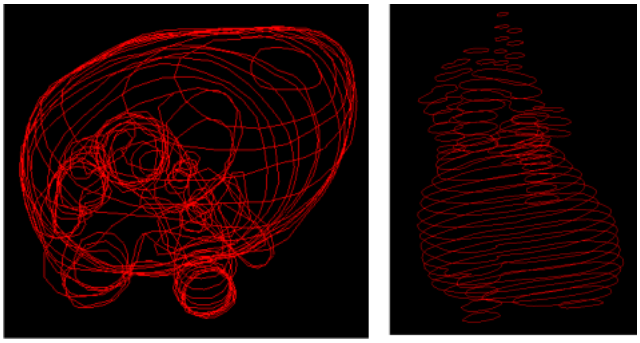

Figure 1: Contours obtained from slice images of a heart; top view (left) and side view (right).

This paper details an implementation of the 3D reconstruction process from slices consisting of simple, non-overlapping polygonal contours using the methods of [1]. It will begin with a brief selection of methods available today, an overview of the method implemented, and finally implementation details on each step.

1.1 A Sample of Reconstruction Methods

As explained in [1], some of the first solutions to the reconstruction problem worked not on polygonal point data, but with raster images. These methods simply produced intermediate raster images at various intervals between those given, and detected the bounding surface

from a series of these images. [2] went a step further and tried to convert this data into a polyhedral surface.

Most solutions assume that any raster data has already been converted into polygonal point data, as the implementation discussed here will do. The earlier attempts generally concentrated on the case where each slice contained only one contour. In other words, no polygons would have holes, and there would be only one polygon per cross-section. The survey in [3] apparently provides a good summary of these special-case methods.

Later works, such as [4, 5], could handle substantially more data, but still had some serious limitations. For example, listed three specific examples that would cause his method to fail, and could not handle branching cases where one polygonal split into two in the next slice.

, whose work forms the basis of this paper's implementation, make good progress in being able to handle more arbitrary data, and uses a inter-slice linear interpolation. This method will work when there are multiple polygonal contours on a slice, and even performs reasonably well for branching and other complicated geometry.

This work was later improved upon by [6] by removing the linearity of the process. This is accomplished by analyzing the the surface flow from one slice to the next and interpolating the resulting differential geometric quantities. Others have taken a completely different direction, using the polygon skeleton concept originally from [7]. The results of the modern algorithms are visually pleasing and seem to represent the original shape with fairly good accuracy.

2 Implementation

This section provides a brief overview of the algorithm that was implemented, then a more detailed description of the implementation itself.

2.1 Overview of the Implemented Algorithm

As mentioned, the method implemented in this paper is due to . It was chosen for its relative simplicity and good results for handling more complicate geometries like branching. This section will briefly outline the algorithm, saving most of the detail for the implementation discussion.

The first step is to read and preprocess the slice data. Each polygonal contour must be listed as a cycle of points in three dimensions, and the z-coordinate must be the same for all contours on a slice. The contours must be listed in a consistent direction; for example, the convention that polygons listed in counter-clockwise order have their area situated in their interiors, whereas clockwise contours represent holes. For the preprocessing, each edge on the polygon is subdivided into several smaller edges of equal length, thereby discretizing the contours.

Next, each pair of adjacent slices is examined in search of contours that have matching portions. To accomplish this, each contour's cycle of edges must be arbitrarily broken so polygonal chains are used instead. A voting process helps find potentially matching alignments between two chains, and possible matches are further processed to find their beginnings

and ends. Overlap is eliminated and matches are filtered by a scoring system that assesses their quality. This process uses a predefined maximum distance value to determine whether two points match, and keeps only those matching chains that have more than one point. As a result, only contours with the same orientation (i.e. clockwise or counter-clockwise) will be matched.

Those portions of contours that matched are now stitched together. That is, triangles are inserted along the chain, starting at one end and proceeding to the other. It is important to orient these triangles consistently; in the case that outer contours are listed in clockwise order, so should the triangles be. This is depicted in Figure 2.

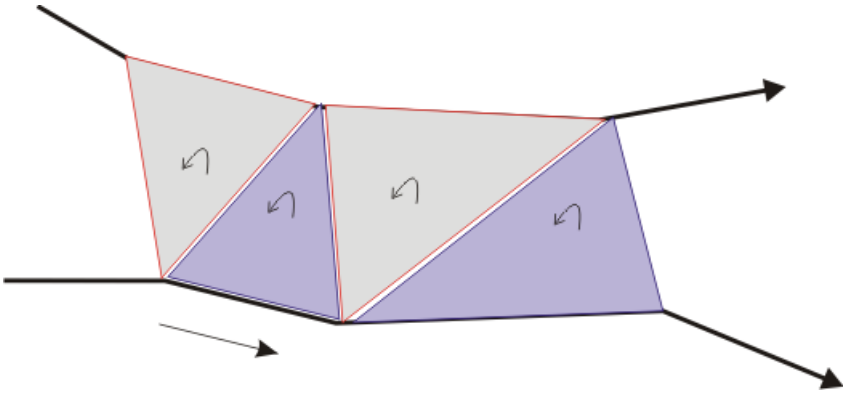

Figure 2: When stitching two matched chains together, the triangles created must be consistently oriented.

Now the algorithm finds the remaining contour edges that have not been matched. Because the previous step already took care of any edges that would overlap between the two slices by considering them a match, none of the remaining edges will cross. Furthermore, if each lower contour's direction is reversed, and all edges of the contours and the triangles are added together, then the remaining edges must form a cycle, known as a cleft. This is due to the fact that, as seen in Figure 3, triangle edges adjacent to each other will cancel out, as will the triangle and match chain edges. Only two triangle edges will remain, and they will serve and bridges from one contour to the other.

The nesting hierarchy is found for these cleft cycles, and then each group of outermost clefts and their children is used to create a complete graph. This graph will have one node for each cleft cycle and edges connecting all nodes to each other node. A minimum spanning tree is then run on this graph. Edges are added between two cycles to bridge them together if the minimum spanning tree has an edge between the two nodes representing the cycles, thereby turning the group of clefts into one single cycle. This new polygon is then triangulated. The triangles from the stitched matches and these new triangles form the mesh for the reconstruction, along with the top and bottom slice to cap off the object on either end.

2.2 Implementation Details

Details on the actual implementation will now be given. The C++ code for this project was developed in Visual Studio using OpenGL, OpenGL Utility, and GLUT libraries. The

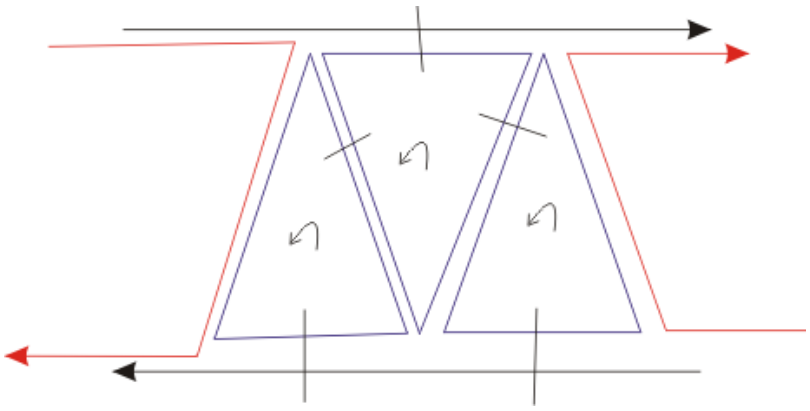

Figure 3: When the lower contour's direction is reversed, each matched edge will cancel with one triangle edge. Most triangle edges will cancel with each other, except for the two on the ends of the match chain.

software ran with reasonable running times on a laptop with a 1.79 GHz AMD processor and 512 MB of RAM. The remainder of this section will outline the main classes involved and how the algorithm was specifically implemented.

The following classes form the core of the implementation:

- Contour
- Slice
- SliceArray
- Triangle and Polyhedron
- Cleft and CleftEdgeCycle

The contour class stores a vector of three-dimensional points representing a polygonal chain. This class is responsible for computing and filtering the match candidates between two contours. It has an inner public class called MatchCandidate that stores pointers to the two contours involved in a match, and the match start and end indexes from the contour's vector. A contour also knows how to render an outline of itself in an OpenGL environment.

The Slice class is essentially a sophisticated container for a set of contours. Most of what it can do it does by forwarding messages to the contours it stores. Its most complicated function is to read a file of slice data and store it in the appropriate format. Similarly, the SliceArray class acts as a container of slices.

The Triangle and Polyhedron classes are used to store the triangle mesh that is created during the matching process. The function buildPolyhedron takes a set of matches for two slices and performs the rest of the algorithm, eventually adding to its set of triangles. It, too, knows how to render itself in an OpenGL environment.

Finally, the Cleft class stores both endpoints of an edge. The reason for this will become clear in the algorithm discussion to follow. Meanwhile, the CleftEdgeCycle represents a full

cycle of cleft edges, collectively known as a single cleft. It is able to find cycles within a set of cleft edges, as well as bridge a group of related cycles into one cycle.

There are other secondary classes in the project, used for the user interface (rendering and interaction) and triangulating the cleft cycles. Some of the code is due to [8, 9, 10].

The discussion will now move to the implementation of the algorithm outlined in Section 2.1.

The first major step is the initial search for match candidates. Although the method in [1] using a voting system to find the most promising shifts, this implementation does not do so explicitly. Instead, a loop over all possible shifts between the two contours in question searches for chains with most points in close proximity that are long enough to be considered matches. The number of points examined for each of these shifts is the maximum length of the two contours involved. This means that one contour may loop around, but any issues with range are saved until later. For each of these points, the distance between their xy-projections is checked to see whether it is within the allowable range. If not, then the number of mismatches since the start of the current match chain is compared to the maximum allowed. If the points do not fulfill all this criteria, the current chain is terminated and a new one started. Otherwise, the current match chain continues. A series of mismatches before or after the chain will not be included in it.

Next, the matches found must be adjusted so there is no overlap. To accomplish this, the indexes in the MatchCandidate class conform to the following criteria: first, the start index of a match chain will be less than the number of points of its associated contour; second, the total number of points in the match will not exceed the minimum length of the two contours; and finally, the end index will not precede the start index. This means that the end index may extend past the number of points in a contour; this is intentional and will help sort out the overlaps. The matches will be put back into range later.

A match candidate will be given a list of other matches. Once adjusted to avoid overlapping with anything in the list, the match will be added to it (assuming it is not completely contained in the other matches). To check for overlap, both the upper and lower contour indexes must be checked. When comparing two matches for overlap, the following cases may occur, where 'this match' refers to the one to be added, and 'other match' refers to one from the list:

- This match is completely contained in the other match. This match will not be added to the list.
- The other match is completely contained in this match. Break this match into two, one before and one after the other match. Recursively check for overlap for the two pieces.
- This match begins before but overlaps the other match. Adjust the end of this match to come before the other match.
- This match begins after the start of the other match but overlaps it. Adjust the beginning of this match to come after the end of the other match.
- No overlap. No action is required in this case.

After the non-overlapping matches are each processed for triangle stitching, all triangle edges and contour edges are saved as cleft edges. As mentioned above, these edges are saved with both endpoints, so finding cycles among them will be simple. Starting with any random edge, the next edge in the cycle will start with the same point this edge ends with. If all edges are stored in a map data structure, keyed on their first endpoint, finding the next edge in the cycle is as simple as looking up the second endpoint in the map.

Groups of cycles are then determined using the Jordan curve theorem. A vertical ray starting at one contour is intersected with each edge of another contour. If there is an even number of intersections, the two contours are completely disjoint; otherwise, they belong to the same group.

After the complete graph is computed for each of these groups, Prim's minimum spanning tree algorithm is used to determine where the cycles should be branched to other cycles in the group. To do the actual bridging, a random cycle is traversed from the first edge until a bridging edge is reached. Each edge is added to a new cycle. The function recursively walks around the other cycle, starting from the bridging point. When it returns, the rest of the first cycle is added to the new one. The algorithm ends by triangulating the new cycle.

3 Results

Data of a cube with a circular hollow was used to test that the implementation works on a basic level. Some slice pairs contained the same contours, which matched perfectly in the implementation. There is some complicated geometry for some of the inner corners in the model, since part of these small, triangular shaped contours needed to match with the corners above them, and part needed to match with the cube's square boundary. Figure 4 shows the slices of the cube, viewed from an angle.

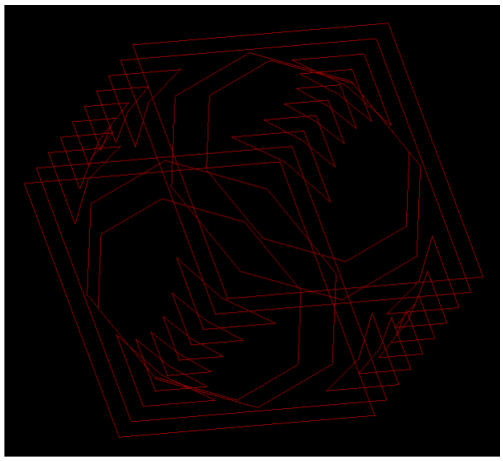

Figure 4: A visualization of the slices of a hollowed out cube.

The cube was reconstructed with no problems; that is, the whole solid was recreated without any holes. The results can be seen in Figure 5.

The next example data was that of a male torso. It did not fare quite as well as the cube, having a few holes here and there. Regardless, there are no major problems, and situations

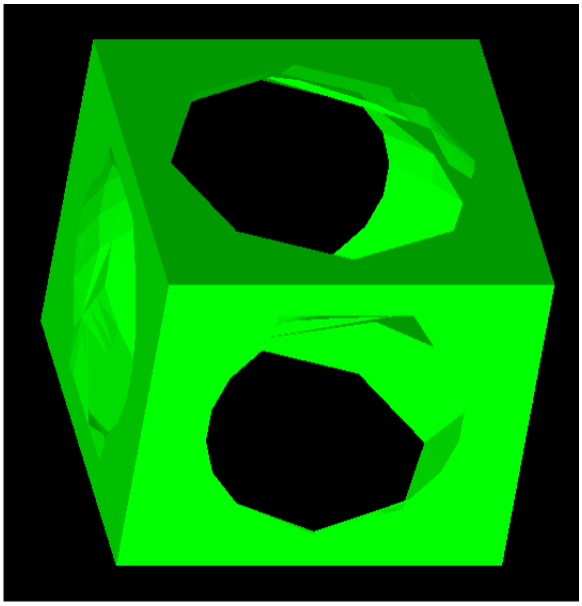

Figure 5: Reconstruction of the hollowed out cube.

of branching were indeed handled. The reconstruction is shown in Figure .

Finally, the heart whose slices were shown in Figure 1 are shown reconstructed in Figure 7. This model obviously did not fare as well as the previous two, with more holes and more instances of unnatural connections between several slices.

4 Conclusion

Based on the results of the hollowed out cube, seen in Figure 5, the algorithm is implemented reasonably well. However, some of the aspects of the implementation were very tricky, and could have easily resulted in small bugs that may lead to the holes and seen in the torso and heart in figures 6and 7. The quality of the heart reconstruction compared to the cube, on the other hand, seems to suggest that there are still limitations to this method. Indeed, if it were perfect, there would not have been so many subsequent papers written on the topic in the last decade or so. Even with these limitations, this method is relatively simple, and has much promise for use in practical applications that don't involve complex data.

References

[1] G. Barequet and M. Sharir, "Piecewise-linear interpolation between polygonal slices," *Computer Vision and Image Understanding: CVIU*, vol. 63, no. 2, pp. 251–272, 1996.

[2] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter, "Two algorithms for the three-dimensional reconstruction of tomograms," *Medical Physics*, vol. 15, pp. 320–327, may 1988.

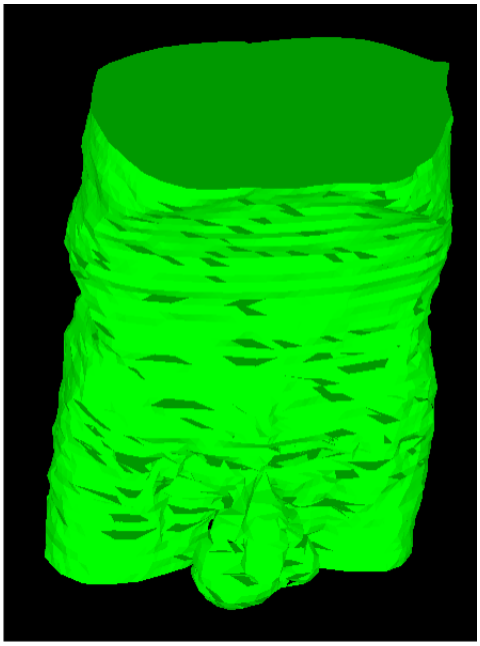

Figure 6: Reconstruction of a male torso.

[3] J. Kenneth R. Sloan and J. Painter, "From contours to surfaces: testbed and initial results," *SIGCHI Bull.*, vol. 18, no. 4, pp. 115–120, 1987.

[4] J.-D. Boissonnat, "Shape reconstruction from planar cross sections," *Comput. Vision Graph. Image Process.*, vol. 44, no. 1, pp. 1–29, 1988.

[5] A. R. Jones, P. G. Hogan, and M. J. Zyda, "Surface reconstruction from planar contours," *Computers and Graphics*, pp. 393–408, 1987.

[6] G. Barequet and A. Vaxman, "Nonlinear interpolation between slices," in *SPM '07: Proceedings of the 2007 ACM symposium on Solid and physical modeling*. New York, NY, USA: ACM, 2007, pp. 97–107.

[7] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner", ""a novel type of skeleton for polygons"," *"J.UCS: Journal of Universal Computer Science"*, vol. "1", no. "12", pp. "752–761".

[8] B. Smith, "Arcball."

[9] B. Jacobs, "Video tutorials rock."

[10] M. Estevao, "Polygon tessellation in opengl."

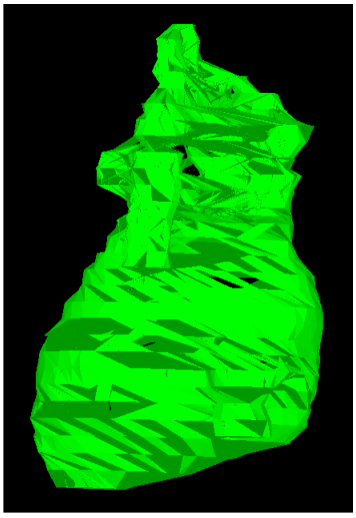

Figure 7: Reconstruction of a heart.