1995

Decimation of 2D Scalar Data with Error Control

Daniel R. Schikore

Chandrajit L. Bajaj

Report Number:
95-004

DECIMATION OF 2D SCALAR DATA WITH ERROR CONTROL

Daniel R. Schikore
Chandrajit L. Bajaj

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR-95-004
January 1995

Decimation of 2D Scalar Data with Error Control

Daniel R. Schikore *Chandrajit L. Bajaj*

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
drs@cs.purdue.edu
bajaj@cs.purdue.edu

Abstract

Scientific applications frequently use dense scalar data defined over a 2D mesh. Often these meshes are created at a high density in order to capture the high frequency components of sampled data or to ensure an error bound in a physical simulation. We present an algorithm which drastically reduces the number of triangle mesh elements required to represent a mesh of scalar data values, while maintaining that errors at each mesh point will not exceed a user-specified bound. The algorithm deletes vertices and surrounding triangles of the mesh which can be safely removed without violating the error constraint. The hole which is left after removal of a vertex is retriangulated with the goal of minimizing the error introduced into the mesh. Examples using medical data demonstrate the utility of the decimation algorithm. Suggested extensions show that the ideas set forth in this paper may be applied to a wide range of more complex scientific data.

Keywords: Computer Graphics, Volume Visualization, Terrain Visualization, Decimation, Isocontouring.

1 Introduction

Scientific data is commonly represented by sampled or generated scalar data defined over a mesh of discrete data points in 2D or 3D. Often, the data is collected at high densities due to the method of sampling or the need to accurately perform a complex physical simulation. Visualizing and computing from the large meshes which result is a computationally intensive task. It is often the case that the data is not as complex as the original mesh, and may in fact be accurately represented with a fraction of the number of mesh elements.

One field which uses dense scalar data is medical imaging. Current medical scanning devices, such as Computed Tomography (CT) and Magnetic Resonance Imaging (MRI), produce hundreds or thousands of 2D slices of scalar data on rectilinear grids of up to 1024 by 1024 pixels, and future devices will output data at an even finer resolution. There are often large regions of each slice which are empty space, outside the region of interest, as well as large structures of almost equal data value within the region of interest.

A similar type of data is used in the earth sciences to represent geological features. Digital Elevation Models (DEM's) are height fields which depict a geographic region as a mesh of elevation values. Much geographic data also has the fortunate property of having many regions which are flat or of constant slope, yet a regular mesh in unable to take advantage of the coherence in the data.

There are many other types of scalar data found in scientific study, including pressure, temperature, and velocity (magnitude). Outside the realm of scientific study, one can even create polygonal meshes of arbitrary greyscale images for displaying 2D images in 3D. This paper describes an algorithm for reducing the complexity of the meshes, while maintaining a bound on the error in the decimated mesh. The decimation algorithm takes advantage of the spatial coherence in the data defined over the mesh, which mesh generators and fixed sized scanning devices cannot do without prior knowledge of the data. We will first describe work in the related field of polygon decimation. The mesh decimation algorithm and its implementation will then be presented, along with illustrations of the usefulness of the decimated meshes in one particular domain, medical imaging. Several extensions of the algorithm reveal that we are not limited to applying this work to 2D scalar data.

2 Related Work

Previous work in data decimation has centered around two fundamental methods. First, data may be decimated in its original form, with data values defined at mesh points. Other methods for decimation extract information from a mesh in the form of lines or polygons, and decimate the resulting geometric primitives. Both provide valuable background information.

Early work in mesh decimation was performed by Fowler, et al[1]. Using terrain data defined over a dense rectilinear grid, a sparse triangular approximation to the data was computed. The approach required that critical values in the data (maxima, minima, saddle points, ridges, etc.) be isolated and formed into an initial triangulation, which was then adaptively refined to meet an error criteria.

Decimation of arbitrary polygonal surfaces is more general and has been the subject of several recent works. Polygonal surface decimation can be applied directly to the case of 2D mesh decimation for scalar data, by simply taking the data values to be heights and mapping the 2D mesh into a 3D polygonal mesh. It is important to note, however, that by discarding the notion of a 2D domain over which variables are defined, in general these methods cannot provide error control for the original data points in the mesh. In decimation of polygonal models, Turk[5] used point repulsion on the surface of a polygonal model to generate nested sets of candidate vertices for retriangulation of models at various levels of detail. Schroeder, et al.[4] decimate polygonal models by deletion of vertices based on an error criteria, and local retriangulation with a goal of maintaining good aspect ratio in the resulting triangulation.

3 Scalar Data Mesh Decimation

The goal of mesh decimation is to reduce the number of mesh elements required to represent the sampled data, while not exceeding an error bound at any of the original mesh vertices. The input mesh is a set of points in 2D, along with scalar data values defined at each point, as well as an edge-connected triangulation of the points. The resulting decimated mesh will consist of a subset of the original mesh points and data values, and a new edge-connected triangulation. Discrete data defined in a triangular mesh is generally made continuous by linearly interpolating data values along the edges of the mesh, and across the interior of the triangle. It is our goal to generate a decimated mesh, such that the interpolated data values in the decimated mesh are within a user-specified error bound of the original sampled data values in the dense mesh.

3.1 Algorithm Overview

The algorithm for decimation is straightforward. Vertices are considered candidates for deletion based on their classification, determined by examining the configuration of each vertex with their surrounding vertices. If a vertex is a valid candidate, the hole which would result from the removal of the vertex and its surrounding triangles is examined. If a valid retriangulation can be found which maintains the error bound for all deleted points, the triangles surrounding the vertex are deleted, and the new triangulation is added.

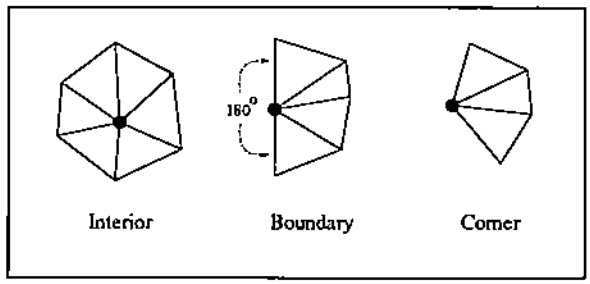


Figure 1: Vertex Classifications

3.2 Classification of Vertices

The first step in attempting to remove a vertex is to determine its classification. There are three simple classifications, illustrated in Figure 1:

Interior - a vertex with a complete cycle of adjacent triangles.

Boundary - a vertex with a half cycle of adjacent triangles which form a 180 degree angle at the boundary of the mesh.

Corner - a vertex with a half cycle of adjacent triangles which do **not** form a 180 degree angle.

Interior and boundary vertices are considered as candidates for deletion, and may be removed based on the error computation described in the following section. Corner vertices are not deleted because they define the domain of the data, a 2D region over which data values are defined. Decimation of the corner vertices would give a different domain in the decimated mesh. With our goal of controlling error at all points defined in the original domain, it would be unacceptable to delete a corner vertex, because we cannot define an error value for a point which is not in the domain of the decimated mesh.

3.3 Computing Errors

We have stated as our goal that the errors at each of the deleted points will be bounded. The error that we are bounding is the error in data value from the original value defined in

the mesh to the value in the decimated mesh. Values in the decimated mesh are determined by a linear interpolation of data values within the triangles of the decimated mesh. This is accomplished by maintaining two error values for each triangle in the mesh. The error values are upper bounds on the error already introduced into the decimation for all deleted vertices which lie in the triangle. The first error value indicates an error above the triangle. In other words, the error represents the maximum error for all deleted vertices which lie in the triangle whose original data value is greater than the interpolated data value in the decimated mesh. The second error value is the complementary case for vertices whose original value is below the interpolated value. This error represents the maximum difference between the original data value and the interpolated data value for all vertices within the triangle whose original data value is below the interpolated value. Of course, the errors are initialized to zero for the original mesh. Note that these error values are upper bounds, and not exact values for the error within a triangle. This allows us to use only a few simple rules for propagating and accumulating errors from successive removals of vertices.

When examining a hole for retriangulation, we must define a rule for propagating errors from the previous triangulation to the new triangulation. Error bounds on the new triangulation will depend only on the error bounds for the triangles in the previous triangulation, as described below. It is useful to infer bounds on the errors along edges from the bounds on the errors within the triangles to which they belong. The error along an edge which lies on the boundary of the hole inherits the error values of the triangle to which it belongs. This is necessary because any triangle in the new triangulation which uses this edge includes a portion of the triangle from the previous triangulation, and thus the errors must propagate to the new triangle. This rule is also sufficient, because all points on the edge also belong to the triangle, and there can be no point on the triangle which has a larger error value than the error bounds for the triangle.

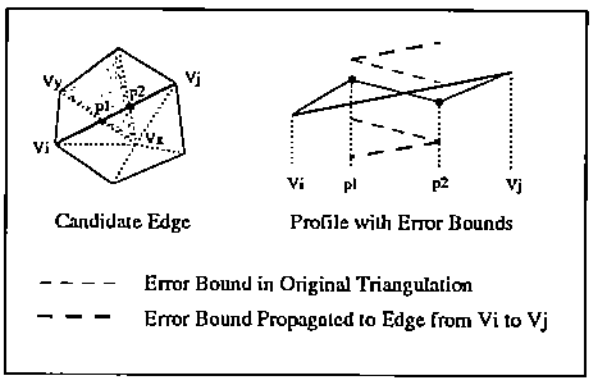


Figure 2: Errors for Interior Vertex

3.3.1 Error in Interior Vertices

With this rule in mind, consider the case of an interior vertex, shown in Figure 2. Vertex v_x is a candidate for deletion, and the old triangulation is shown in dotted lines. Given the edge from v_i to v_j where v_i and v_j are non-adjacent vertices, we can compute errors for this edge as follows. Separate errors are computed for each triangle in the old triangulation which the new edge crosses. Edge v_iv_j in Figure 2 has a single entry point into and exit point from each of the three triangles which it crosses. The entry point for the shaded triangle is $p1$, and the exit point is $p2$. The error bound for the new edge will be the maximum of the error bounds within each of the triangles which it crosses. In order to propagate the error bounds from the old triangles to the retriangulation, we must assume the worst possible case for each introduced edge. That is, in order to determine how an old triangle affects the error along an introduced edge, we assume that the error at each point in the old triangle is equal to the maximum error within the triangle, both above and below. We first compute an **introduced error** for the entry and exit points. The introduced error is the signed distance, in data value, from the old triangle to the new triangle at the given points. We also define the **maximum introduced error** to be the greatest of the signed introduced errors, and the **minimum introduced error** to be the least of the signed introduced errors. The data value for the old triangulation is linearly interpolated from the data values at v_x and v_y. The data value for the new edge is interpolated from the values at v_i and v_j. Introduced errors between the two points are linear combinations of the errors at the endpoints ($p1$ and $p2$) because the data values within a triangle are linearly interpolated. Therefore, the endpoints alone give us the maximum introduced error, as well as the minimum introduced error, along a segment of the introduced edge. Again assuming the worst case, suppose that the maximum error above the old triangle occurred at the maximum introduced error. The sum of the old error bound and the maximum introduced error would be the new error bound above the edge. Likewise, assume that the previous maximum error below the triangle occurred at the minimum introduced error. Because the introduced errors are signed values, we take the difference of the previous error bound below the triangle and the minimum introduced error to be the new error bound below the introduced edge. The error bounds are shown as dashed lines in the profile in Figure 2. Note that if both signed errors are positive, the error bound below the edge will decrease, and likewise the error bound above the edge will decrease if both introduced errors are negative. This cancellation of errors occurs because all points along the introduced edge move in the same direction away from the previous triangulation, closing in on the previous error bound in that direction. This is the main motivation for maintaining error bounds above and below the triangles. The possibility of cancellation of errors during retriangulation increases the potential for further decimation, because by lowering error bounds we provide more opportunity for introduced error at later stages.

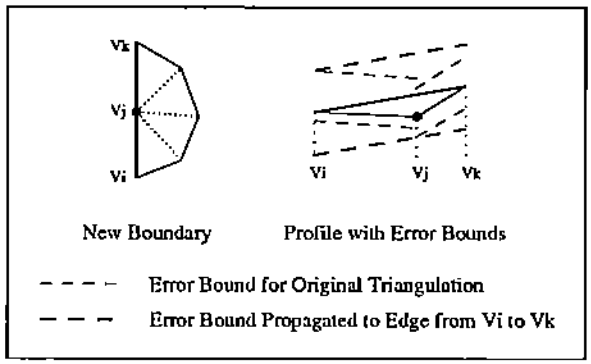


Figure 3: Errors in Boundary Vertices

3.3.2 Error in Boundary Vertices

The case of a boundary vertex, shown in Figure 3, requires one additional step. Removal of v_j in Figure 3 introduces an edge from v_i to v_k. This edge will have at least the maximum of the errors of the edges from v_i to v_j and v_j to v_k. The error will be compounded in the direction of the error introduced by removing v_j. We compute the added error by taking the linearly interpolated data value at v_j from the data values at v_i and v_k, and subtracting the original data value at v_j. This introduced error will be added to the error above the edge if the introduced error is positive, and added to the error below the edge if the introduced error is negative. Errors for edges interior to the hole are computed in the same manner as for interior vertices.

3.3.3 Error Summary

We have completed a definition of error and rules for propagating these errors to edges during retriangulation such that the error bounds are maintained. The only remaining point is to state a rule for determining the error bounds for a new triangle which is contained within a hole. We take the maximum of the error bounds for the edges composing a triangle as the bound for the triangle itself. As was the case for edge errors, this is sufficient because all of our approximations to the data are linear. Errors for edges around the boundary take into account the original triangle and associated errors, while errors for edges within the hole account for any error which is added from the retriangulation. The next step is to propose a method for retriangulating using the rules which have been defined.

3.4 Retriangulation

The heart of the decimation is the retriangulation. This process is shown in figure 4. For each candidate vertex, we search for a retriangulation which does not violate the error criteria. This search must be efficient, therefore we cannot search all possible triangulations and use the one which minimizes the error. We attempt to minimize the error by successively choosing edges interior to the hole which create

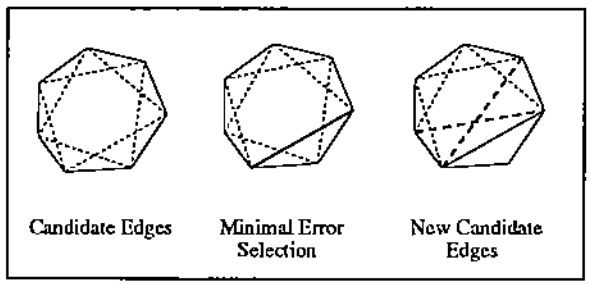


Figure 4: Illustration of Retriangulation Process

a polygon with an error which does not exceed our bounds. First, we create a candidate list of edges by computing the errors for all edges between vertices which are separated by one vertex. Candidate edges are those which have an error less than our error bounds. We propose two methods for selecting an edge from among the candidate edges to create a triangle in the new triangulation. First, we may choose the edge which has the least error. This method will attempt to approximate the original data as closely as possible. A second possible selection criteria is to choose the edge which *introduces* the least error into the retriangulation. This method attempts to closely approximate the previous triangulation in the new triangulation.

To perform the triangulation, we iteratively choose an edge by one of the above selection criteria, while at the same time adding new edges to the candidate list with vertices which have become one-separated due to the edge selection. This retriangulation scheme is not an exhaustive search of the possible triangulations, which is quadratic in the number of vertices. We designed this method to fulfill the requirement of efficient retriangulation, as it performs linear time search for a triangulation which satisfies our error bounds. For this reason, it may not find the retriangulation which best approximates the original data. In fact, it may not find an acceptable triangulation even if one exists. It is possible that by choosing a particular edge from the list of all candidate edges, edges which are part of an acceptable triangulation will be eliminated, and triangulation may proceed to a dead end.

It is also possible to use a hybrid of two or more selection schemes to improve the possibility that a valid retriangulation will be found. For example, one may attempt to find a triangulation using an alternate selection scheme if previous schemes fail, or one may even alternate selection schemes for each edge selection during a single retriangulation. Our current implementation uses the former hybrid scheme, first attempting a retriangulation by selecting the edge which introduces the minimum error, and upon failure, attempting a retriangulation by selecting the edge with minimum error. Notice that there is a tradeoff involved between the number of triangulations which are inspected, and the efficiency of the retriangulation process. Of course, the overall performance of the decimation is likely to improve if more triangulations are inspected, because the probability of finding a valid retriangulation is increased.

4 Results

Our results indicate that error bounded decimation produces results comparable to other decimation methods with several added benefits. First, it achieves a goal of error control which some methods are unable to accomplish for specific types of data. Error bounded decimation also has the advantage that the only necessary parameters are the error bounds. Other methods require the user to specify "feature" parameters, which the algorithm uses to maintain high-frequency edges in the data. Error bounded decimation naturally retains features by choosing a retriangulation which closely approximates the original data.

Our first test case was a 256 by 256 scan of a human heart. The data values defined at each point are 8 bits in resolution. Figure 5 shows the original slice as a 256 by 256 greyscale image. The heart data was decimated by triangulating each of the 255^2 quadrilateral data elements into two triangles, for a total initial mesh of 127K triangles. In Figures 6 and 7 the left image shows a decimated triangulation of the data, while the right image shows a gouraud shaded triangulation using the data values at the vertices in the decimated triangulation.

Our second test case was a 512 by 512 scan of a human head. The original CT image is displayed in Figure 8. Data values for the head are 16 bit integers. The error parameter for decimation is an absolute error bound, therefore the error parameters for the test cases using the head data set were chosen to be proportionally higher than the error parameters chosen for the test cases using the 8 bit resolution heart data set. Notice that in both the 8 bit heart data and the 16 bit head data, an absolute error bound of only 2 resulted in decimation levels of over 40%. This is a direct result of the spatial coherence in sampled data which is exploited by the decimation algorithm. Figures 9 and 10 show example decimated meshes for the head in wireframe and gouraud. Listed below are decimation percentages for various error values in our two test data sets:

Heart Data

Error	*Percentage*
2	41%
4	60%
8	73%
16	82%

Head Data

Error	*Percentage*
2	48%
20	56%
100	70%
200	78%
500	86%
1000	90%

Tables of Decimation Results

5 Applications using decimated meshes

Decimated meshes of data are useful for several purposes. Many operations on dense meshes waste much of their time determining which parts of the data are needed to compute the results. Straightforward isocontouring of data, for example, examines each element of data to determine if there is an intersection with the surface of interest[2]. It has been estimated that 30% to 70% of the time spent naively computing an isosurface from such a 3D grid of data is wasted examining cells which have no intersection with the surface of interest[7]. For this reason, methods have been developed which speed the computation of isocontours[6]. However, such speedup methods still produce a large number of primitives as a result, sometimes on the order of millions, pushing current graphics hardware to the limit. To compensate for this, polygon decimation algorithms have been developed to reduce the number of polygons needed to represent a surface[4]. Recent work addresses the specific problem of the sheer number of polygons produced by isosurfacing in rectilinear 3D data. Interpolation schemes for locating surface intersections within volume elements are modified, with a goal of decreasing the necessary postprocessing time required to decimate the resulting triangulation[3]. As we have seen, decimation can reduce the source data by as much as 90%. Performing the desired operation on the decimated data will allow tremendous speedups, as well as reducing the number of primitives generated, taking the place of both speedup methods and postprocessing at the price of a setup cost to decimate the original data. It is necessary to note that the decimated mesh loses some of the information present in the dense mesh, and as such, components of the isocontour may not be detected when computing from the decimated mesh. What is suggested is that for interactive browsing of a data set, it would be useful to quickly compute and display multiple isocontours computed from the decimated mesh, and perhaps compute the full resolution isocontour when the user decides to examine an isocontour more closely.

Several examples are given in Figures 11, 12, and 13. Figure 13 is a generated scalar field sampled from a mathematical function. In each figure, the left image is the original scalar data field with an isocontour computed directly from the dense data. The corresponding image on the right is a decimated approximation of the data, with the isocontour for the same isovalue computed from the decimated mesh. Notice that while there are 80% - 90% fewer elements in the decimated meshes, and 50% - 66% fewer line segments generated from the decimated meshes, the differences in the isocontours generated in the two meshes is almost negligible, and certainly a reasonable approximation for user interaction and browsing of contours.

6 Extensions

This work may be extended in many ways. The following four suggested improvements and extensions to the algorithm will extend the usefulness of this method to many types of scientific data as well as improve the performance:

Allow Quad Mesh Elements - by allowing quadrilateral mesh elements as well as triangular mesh elements, we gain in two ways. First, source data which is quadrilateral in nature need not be triangulated in order to be processed,

saving from having to double the number of elements before decimation. Second, retriangulation (or remeshing) would not necessarily have to reduce the hole to triangles. This may involve a more complicated search algorithm to attempt to remesh a hole, or it may use a simple extension of the one given here in which we check edges that not only chop of a triangle, but chop off a quadrilateral element as well. In either case, more possible triangulations can only improve the prospects for decimation.

Extend to 3D - The extension of this algorithm to 3D is relatively simple. Using tetrahedral elements, deletion of a vertex creates a hole which is a star-polytope. The main challenge here is choosing a new set of tetrahedra to fill the hole. Isosurface generation and Volume Visualization could both benefit from the computation of a decimated 3D mesh.

Extend to arbitrary meshes - The ideas set forth in this paper need not apply only to scalar data over a 2D domain. This restriction is only for the definition of error, which is an error in data value for a given 2D point. A more general 3D polygon mesh decimation algorithm may be implemented using the same ideas and a modified definition for error. For example, error may be defined as the distance to the retriangulated mesh in the direction of the average plane for the vertices making up the hole.

Decimation of non-scalar data - Many scientific applications have many variables defined over a single mesh. Data decimation need not be only for scalar data. Given a vector of scalar data values at each point of a mesh, we can keep error values for each variable, decimating the mesh only when the error for all variables is kept within our error limits. The performance of simultaneous decimation with several variables will depend largely on the correlation between the variables.

Acknowledgements

We are grateful to the of research centers of the University of North Carolina for anonymous ftp access to the head data set, and to Tsuyoshi Yamamoto and Hiroyuki Fukuda of Hokkaido University for the use of the heart data. Our thanks also to Katherine Price, for many useful discussions and helpful comments. This research was supported in part by NSF grants CCR 92-22467 and GER-9253915-02, AFOSR grants F49620-93-10138 and F49620-94-10080, and ONR grant N00014-91-1-0370.

References

[1] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. In *Computer Graphics (SIGGRAPH '79 Proceedings)*, volume 13(3), pages 199–207, August 1979.

[2] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[3] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In R. D. Bergeron and A. E. Kaufman, editors, *Visualization '94 Proceedings*, pages 281–287, October 1994.

[4] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26(2), pages 65–70, July 1992.

[5] Greg Turk. Re-tiling polygonal surfaces. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26(2), pages 55–64, July 1992.

[6] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation extended abstract. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24(5), pages 57–62, November 1990.

[7] Jane Wilhelms and Allen Van Gelder. Topological considerations in isosurface generation extended abstract. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24(5), pages 79–86, November 1990.

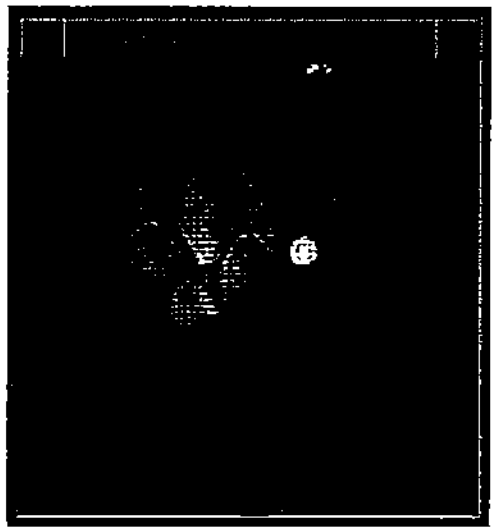
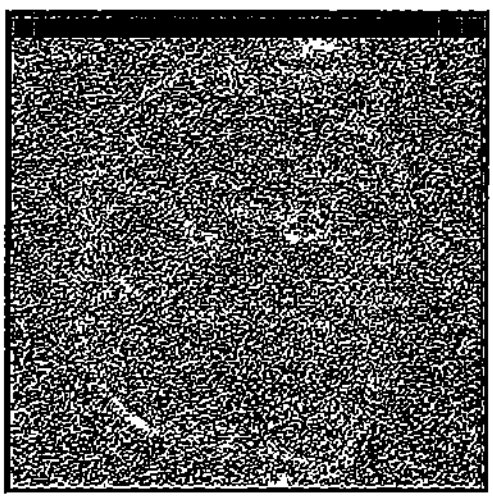


Figure 5: Original Heart CT Image 127K Triangles



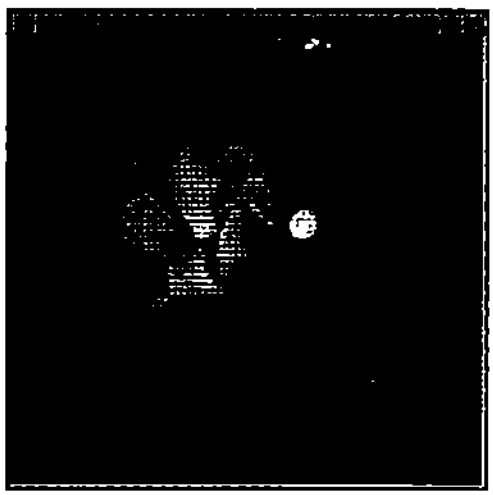
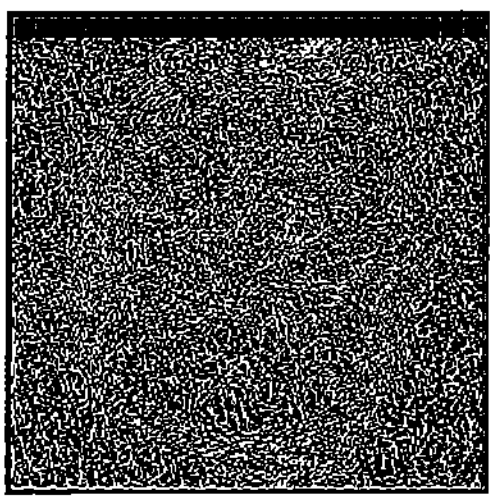


Figure 6: Error = 8 73 % Decimation 33K Triangles



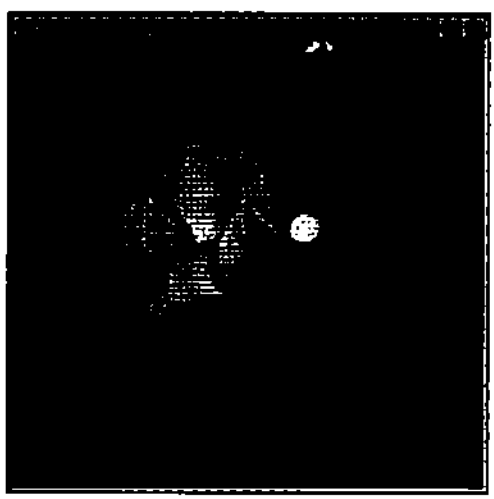


Figure 7: Error = 16 82 % Decimation 22K Triangles

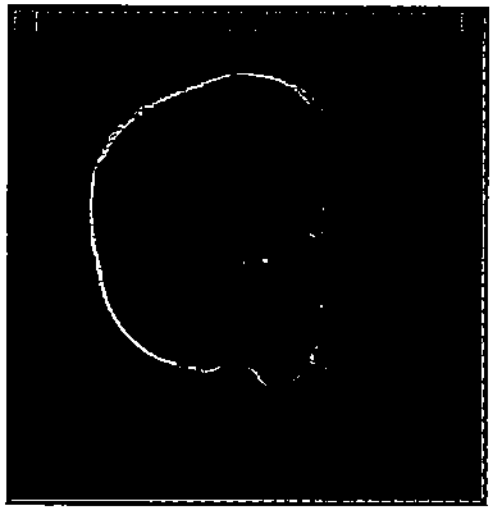
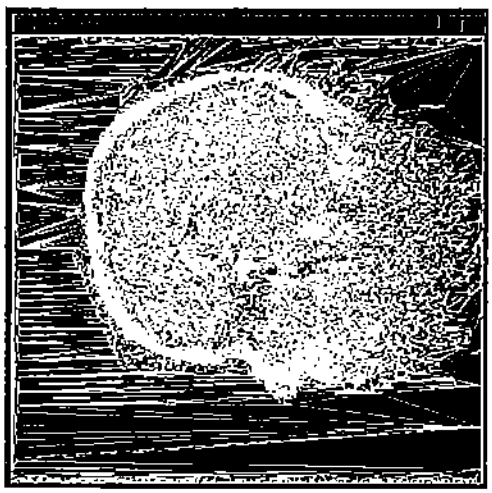


Figure 8: Original Head CT Image 517K Triangles



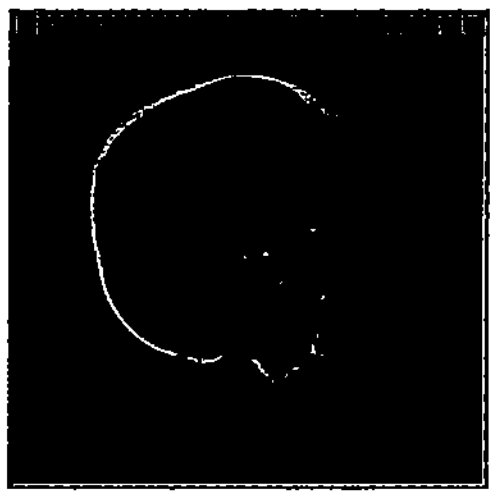
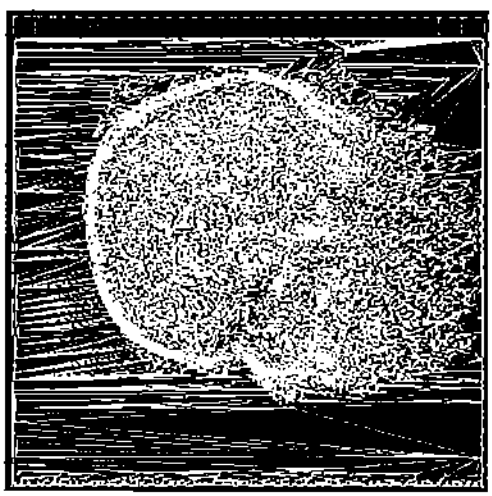


Figure 9: Error = 500 86 % Decimation 70K Triangles



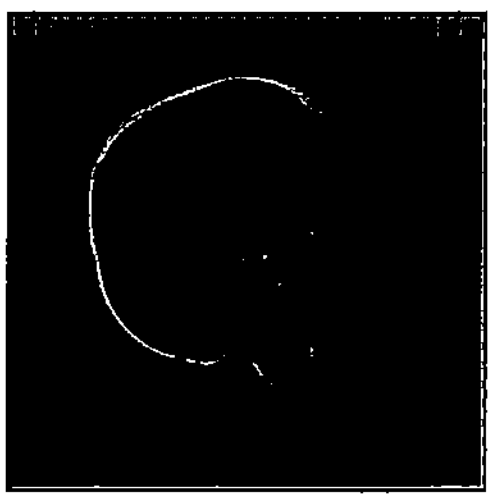


Figure 10: Error = 1000 90% Decimation 50K Triangles

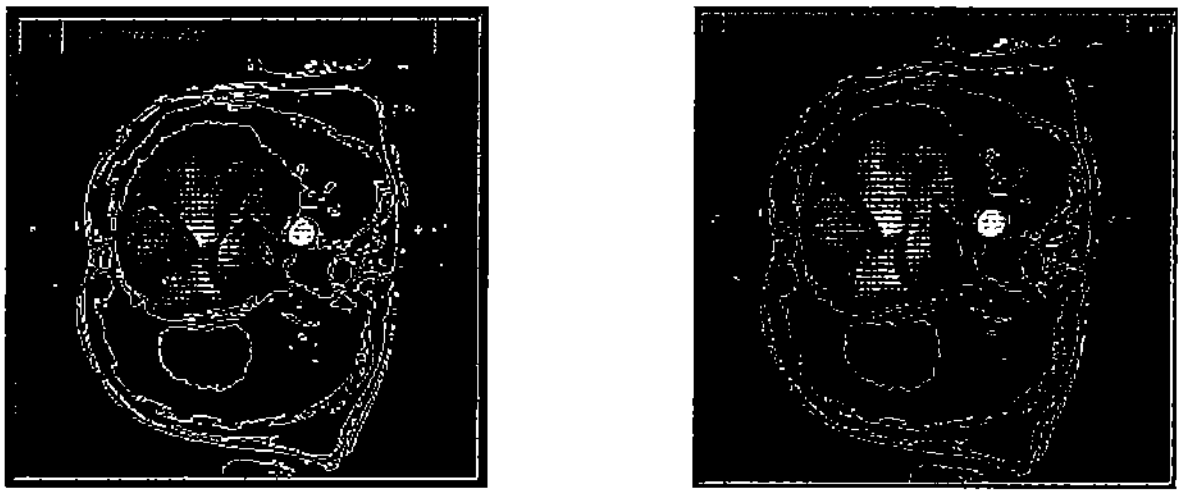


Figure 11: Left: Isocontour from original data (6419 edges)
Right: Isocontour from 82% decimated data (2357 edges)

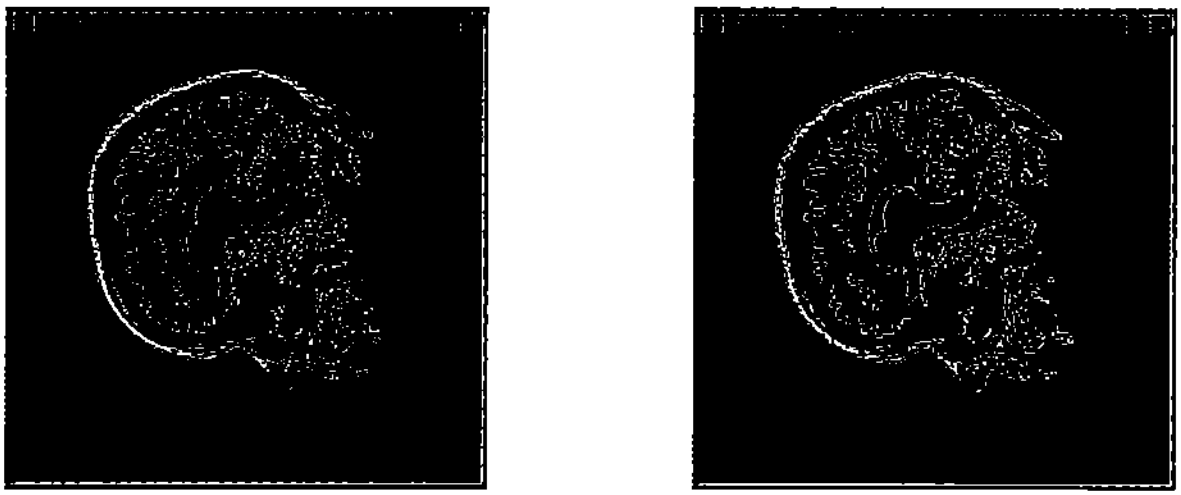


Figure 12: Left: Isocontour from original data (12594 edges)
Right: Isocontour from 86% decimated data (6147 edges)

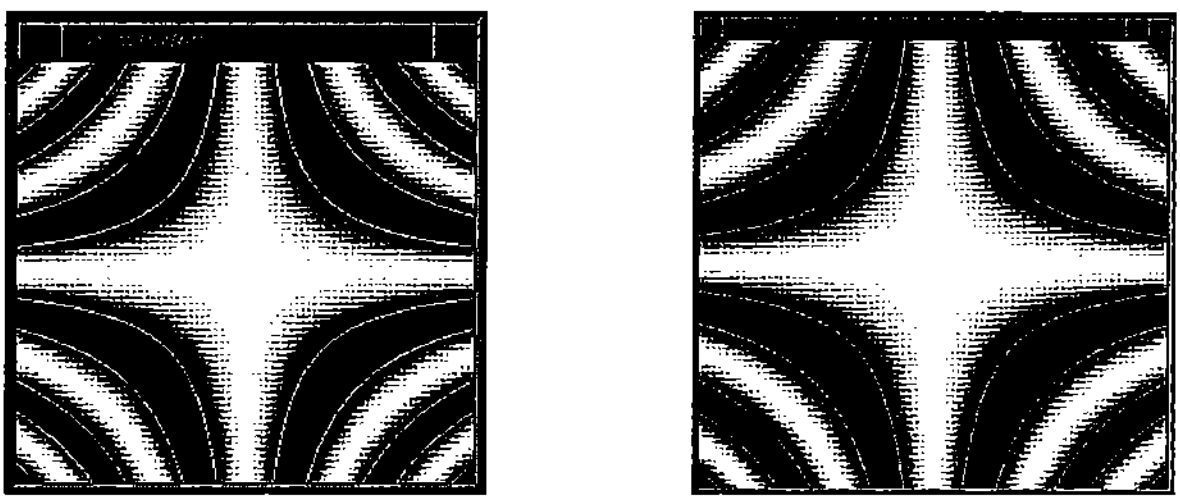


Figure 13: Left: Isocontour from original data (2552 edges)
Right: Isocontour from 90% decimated data (861 edges)