ABSTRACT

ANALYSIS AND REPAIR OF STL FILES

By

Anthony D. Martin

May 1998

STL is the primary format used for the transfer of a model from a 3D CAD system to a rapid prototyping machine. A rapid prototyping machine uses the model definition contained within the STL file to produce a physical model. Although most 3D CAD systems can export STL files, many produce STL files containing errors. There are a number of common problems with STL files that can cause undesired flaws in the resulting part if the file isn't repaired first. This thesis presents a computer program that can verify and, if necessary, repair many common errors. The program can also perform a number of utility operations such as scaling, translation, calculation of volume, and conversion from the STL format to a number of other formats. A description of the algorithms used and the source code for the program is also included.

ANALYSIS AND REPAIR OF STL FILES

A THESIS

Presented to the Department of Mechanical Engineering

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By Anthony D. Martin

B.S., California State University Long Beach, May 1993

May 1998

WE, THE UNDERSIGNED MEMBERS OF THE COMMITTEE,

HAVE APPROVED THIS THESIS

ANALYSIS AND REPAIR OF STL FILES

By

Anthony D. Martin

COMMITTEE MEMBERS

____

C. Barclay Gilpin, Ph.D. (Chair)   Mechanical Engineering

____

Ortwin Ohtmer, Ph.D.   Mechanical Engineering

____

Karl H. Grote, Ph.D.   Mechanical Engineering

ACCEPTED AND APPROVED ON BEHALF OF THE UNIVERSITY

____

Mihir K. Das, Ph.D.
Associate Dean for Instruction, College of Engineering

California State University, Long Beach

May 1998

ACKNOWLEDGEMENTS

I sincerely thank Dr. C. Barclay Gilpin for his guidance and assistance in making this thesis a success. I especially want to thank the Internet community for reporting bugs and for requesting additional features for the Admesh program. Foremost, I thank my wife, Andreea Martin, for the support she gave me during the preparation of this work.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In recent years, an industry of producing three-dimensional models directly from 3D CAD data has grown rapidly. Several companies produce machines that can fabricate a physical three-dimensional model out of various materials including plastic, paper and metal. Generally, the machines run unattended and quickly produce an accurate model directly from CAD data without the need for a highly skilled model-maker or machinist. These machines are generally known as rapid-prototyping machines and the industry that has developed around these machines is called the rapid-prototyping industry.

STL Files

One of the first companies to produce rapid prototyping machines was 3D Systems. 3D Systems needed a file format that could be exported from various CAD systems and used as input to their machine. The format they developed is called the STL file format. Most 3D CAD systems on the market today can export STL files and it has become the standard file format for the rapid prototyping industry.

Problems

Many STL files have errors that can result in various flaws in the resulting model if they are not fixed first. A perfect STL file should not have any "holes" in the surface that would leave the solid representation of that part undefined. It is quite common, however, for a CAD system to create an imperfect STL file as a result of the file being produced from an incomplete part or "unstitched" surfaces, or due to a faulty algorithm used to generate the STL file.

Admesh

Admesh is a program that was written to verify the correctness of an STL file as well as to repair flawed STL files. It can also perform a number of related utility operations such as scaling, rotating, translating, calculating part volume, and converting STL files to various other formats.

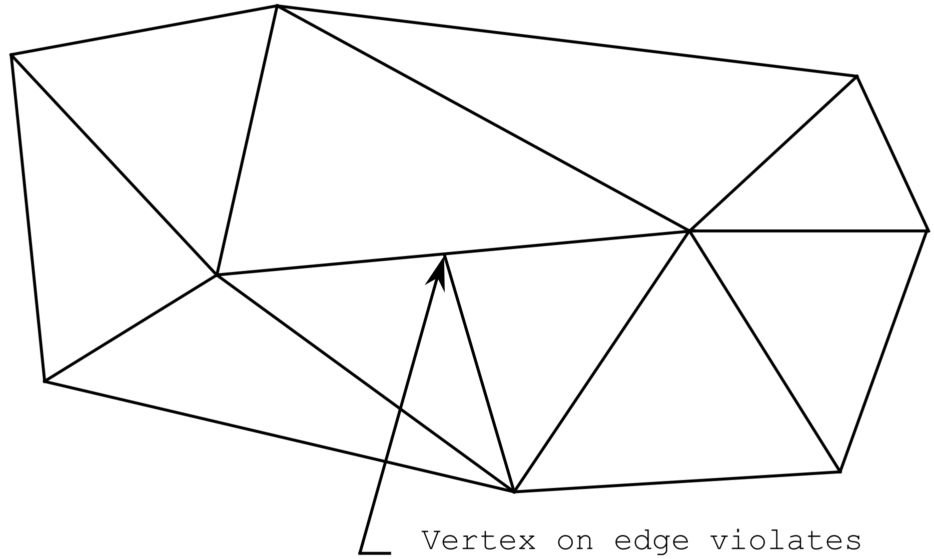
CHAPTER 2

STL FILE FORMAT

One of the primary reasons for the popularity of the STL file format is its simplicity. By using a simple triangular facet representation of the part, it is relatively easy for CAD systems to export an STL file as well as for makers of rapid-prototyping machines to read and process the format.

An STL file represents the surface of a solid as a mesh of triangular facets. In a valid STL file, the vertex-to-vertex rule must be met. That is, each triangular facet should have three adjacent facets that share two vertices. Vertices that fall on the edge of an adjacent facet are not permitted. The vertex-to-vertex rule is shown in Figure 1.

For each facet, three vertices are specified using a right-handed, Cartesian coordinate system. The vertices should be oriented counter-clockwise when viewed from outside the part. Each facet also has a corresponding unit normal vector that points away from the object.



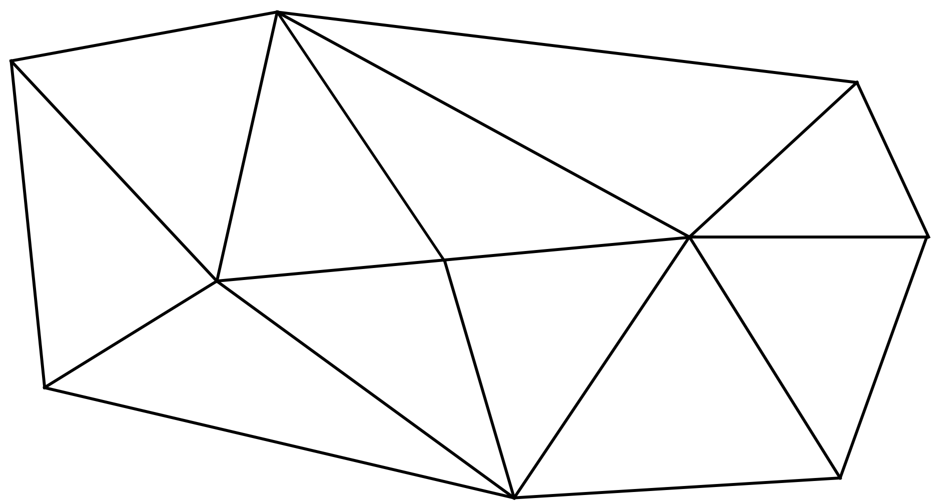


Figure 1. Vertex-to-vertex rule

There are two types of STL files--ASCII and binary. Since ASCII STL files are about five times the size of a binary STL file, the ASCII format is intended primarily for debugging and experimenting, rather than production use. In most cases, the binary format is preferred.

ASCII STL Format

The ASCII STL file format is the easiest to visualize since it can be created and edited with a standard text editor. Each line in an ASCII STL file contains one of the following entries:

```
solid [part name]
facet normal <i value> <j value> <k value>
outer loop
vertex <x value> <y value> <z value>
endloop
endfacet
endsolid [part name]
```

Although the STL specification shows all keywords in lowercase, many ASCII STL files have uppercase keywords. Therefore, any tool which reads STL files should accept either uppercase or lowercase keywords. Most ASCII STL files are indented to more clearly show the structure.

The indentation is not specified in the STL specification, so a tool which reads STL files should accept any amount of indentation. An example ASCII STL file is shown in Figure 2.

Binary STL Format

Binary STL files store the same triangular facet information as ASCII files, but instead of representing the coordinates in ASCII, they are saved as 32-bit floating-point numbers. The binary format is much more compact than the ASCII format, with files about one-fifth the size of ASCII files, and it is therefore faster to read and write these files. The binary format also prevents any loss of accuracy that may occur during the conversion between floating point and ASCII. All entries in a binary STL file are little endian, and all floating point data is single precision (32-bit) in accordance with ANSI/IEEE standard 754. The number of facets is specified with a 32-bit unsigned integer. The binary format is shown in Figure 3.

```
solid tetra.stl
  facet normal  0.00000000  0.00000000 -1.00000000
    outer loop
      vertex  0.00000000  1.00000000  0.00000000
      vertex  1.00000000  0.00000000  0.00000000
      vertex  0.00000000  0.00000000  0.00000000
    endloop
  endfacet
  facet normal -1.00000000  0.00000000  0.00000000
    outer loop
      vertex  0.00000000  1.00000000  0.00000000
      vertex  0.00000000  0.00000000  0.00000000
      vertex  0.00000000  0.00000000  1.00000000
    endloop
  endfacet
  facet normal  0.00000000 -1.00000000  0.00000000
    outer loop
      vertex  0.00000000  0.00000000  0.00000000
      vertex  1.00000000  0.00000000  0.00000000
      vertex  0.00000000  0.00000000  1.00000000
    endloop
  endfacet
  facet normal  0.57735025  0.57735025  0.57735025
    outer loop
      vertex  1.00000000  0.00000000  0.00000000
      vertex  0.00000000  1.00000000  0.00000000
      vertex  0.00000000  0.00000000  1.00000000
    endloop
  endfacet
endsolid tetra.stl
```

Figure 2. ASCII STL file of a tetrahedron

Number of Bytes	Description
80	Header: Optional ASCII text
4	Unsigned int = number of facets
First Facet	
4	Float normal x
4	Float normal y
4	Float normal z
4	Float vertex 1 x
4	Float vertex 1 y
4	Float vertex 1 z
4	Float vertex 2 x
4	Float vertex 2 y
4	Float vertex 2 z
4	Float vertex 3 x
4	Float vertex 3 y
4	Float vertex 3 z
2	Not used (set to zero)
Second Facet	
4	Float normal x
4	Float normal y
4	Float normal z
4	Float vertex 1 x
4	Float vertex 1 y
4	Float vertex 1 z
4	Float vertex 2 x
4	Float vertex 2 y
4	Float vertex 2 z
4	Float vertex 3 x
4	Float vertex 3 y
4	Float vertex 3 z
2	Not used (set to zero)

Facets continue to end of file (EOF).

Note: All numbers are little endian. Floats are IEEE 754 standard, 32-bit floating-point numbers.

Figure 3. Binary STL file format

CHAPTER 3

ADMESH

Admesh is a program that was written to process STL files. This chapter describes the algorithms that are used by Admesh to verify and repair common errors in STL files. The source code for Admesh is provided in Appendix A.

Verifying the Correctness of an STL File

To verify the correctness of an STL file, a neighbors list must be generated. This list contains, for each facet, the three neighbors of that facet. For example, the simple part listed in Figure 2 has the following neighbors list:

Facet 0: Neighbors 1, 2, 3

Facet 1: Neighbors 0, 2, 3

Facet 2: Neighbors 0, 1, 3

Facet 3: Neighbors 0, 1, 2

Such a list is also known as a connectivity list. To create the neighbors list, for each facet the entire file must be searched to find the three neighboring facets that share two vertices with that facet. The most obvious algorithm for solving this problem is shown here:

1. Read all facets from the file into memory.
2. Search the entire file for a facet with two vertices that match the first two vertices of the first facet.
3. If a matching facet is found, add its facet number to the neighbors list for the first facet.
4. Repeat steps 2 and 3 for the second and third vertices of the first facet.
5. Repeat steps 2 and 3 for first and third vertices of the first facet.
6. Repeat steps 2 through 5 for all remaining facets.

The problem with this algorithm is that it is very inefficient since it searches the entire file many times, once for each facet. This algorithm is of time complexity of the order $O(N^2)$, which means that the number of cycles required to build the neighbors list is proportional to the square of the number of facets. The exact number of cycles can be determined by the formula

$$\frac{N(N - 1)}{2}$$

where N is the number of facets.

It is not uncommon for an STL file to have well over 100,000 facets. If this algorithm were used on a file with only 100,000 facets, it would have to go through 4,999,950,000 cycles to verify the entire file. It could easily take many hours or even days to process a large STL file using this algorithm.

The algorithm that Admesh uses is much more efficient. The algorithm is shown here:

1. Read all facets from the file into memory.
2. Combine the first two vertices of the first facet into an edge structure.
3. Use a hashing function to generate a hash table index from the edge structure.
4. Insert the edge structure into the hash table at the index location.
5. Repeat steps 2 through 4 for the second and third vertices of the first facet.
6. Repeat steps 2 through 4 for first and third vertices of the first facet.
7. Repeat steps 2 through 6 for the rest of the facets, checking if a matching edge already exists in the hash table at the index location. If the

matching edge is already there, add the facet to the neighbors list and remove the edge from the hash table.

Since this process only makes one pass through the file, the algorithm is close to a time complexity of order $O(N)$. This means that for a file with 100,000 facets, only 100,000 cycles would be required to verify the entire file. If each cycle takes the same amount of time as the first method, this file can be verified 40,000 times faster using this algorithm. For larger STL files the time difference is even greater, so it is very important for Admesh to use an efficient algorithm. The algorithm isn't exactly $O(N)$ since there will be some collisions in the hash table that require traversing a list of edges at the same index to find the matching edge (Sedgewick, 1990). In practice, the effect of collisions on performance has proven to be negligible.

Repairing STL Files

Once Admesh has generated the neighbors list, it knows which facets, if any, are unconnected, i.e. which ones have fewer than three neighbors. At this point, Admesh will use two different methods to try to connect these facets.

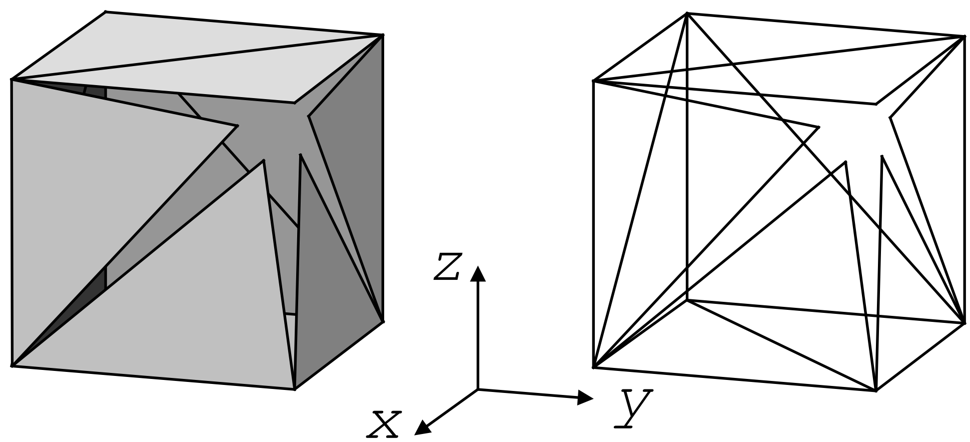
Connecting Nearby Facets

The first method that Admesh uses assumes that some of the facets might be disconnected simply due to floating-point rounding or truncation. Disconnected facets might only be disconnected by a very small amount. To connect nearby facets, Admesh divides the part volume into many small cubes. The size of the cubes is determined by the "nearby tolerance" factor. The algorithm is the same as that used in the initial check, but instead of representing the vertices as floating-point values, integers are used. Vertices of unconnected facets that fall within a single cube are joined into one node. Admesh starts with a small tolerance, which it initially sets to the length of the shortest edge of any facet in the file. If there are still unconnected facets after the first nearby check, then the tolerance is increased and the file is processed again. Starting with a relatively small tolerance reduces the chance of connecting facets that shouldn't be neighbors.

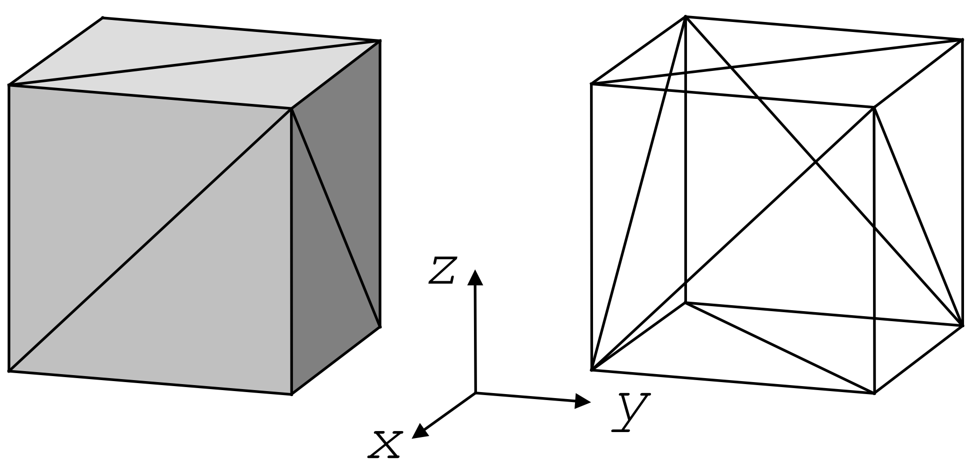
Figure 4 shows an STL model of a cube with some disconnected facets. The nearby check of Admesh was able to repair the STL file to produce the cube shown in the bottom half of the figure. Appendix B contains the listing of the STL file before and after processing, as well as the Admesh summary report produced when the file was processed.

Filling Holes

If there are unconnected facets remaining after the nearby check, then Admesh tries to repair holes in the STL file by adding facets. For each unconnected edge, Admesh adds a facet between that edge and the nearest unconnected edge. First, any facets that have zero neighbors are removed since this algorithm requires at least one connected edge. The process is repeated until all facets in the file are connected. This method is guaranteed to connect all of the facets in the file. However, the resulting part or parts may not resemble the



Cube with 10 disconnected edges.

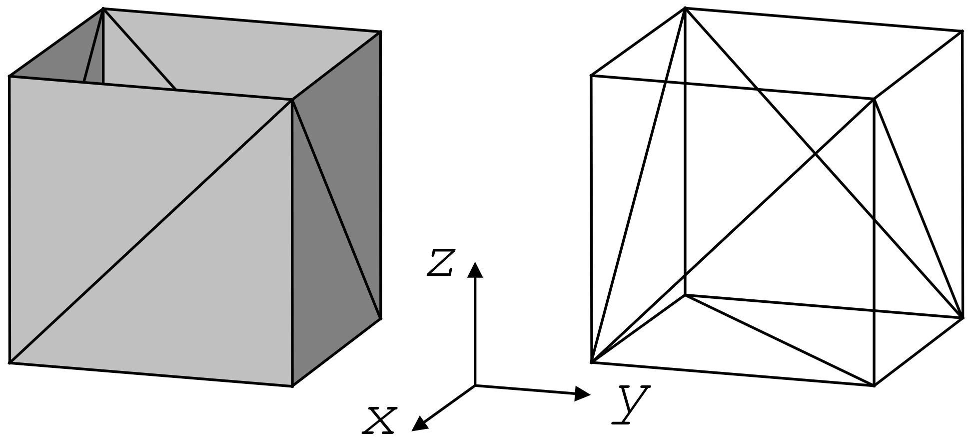


Cube after being processed by Admesh.

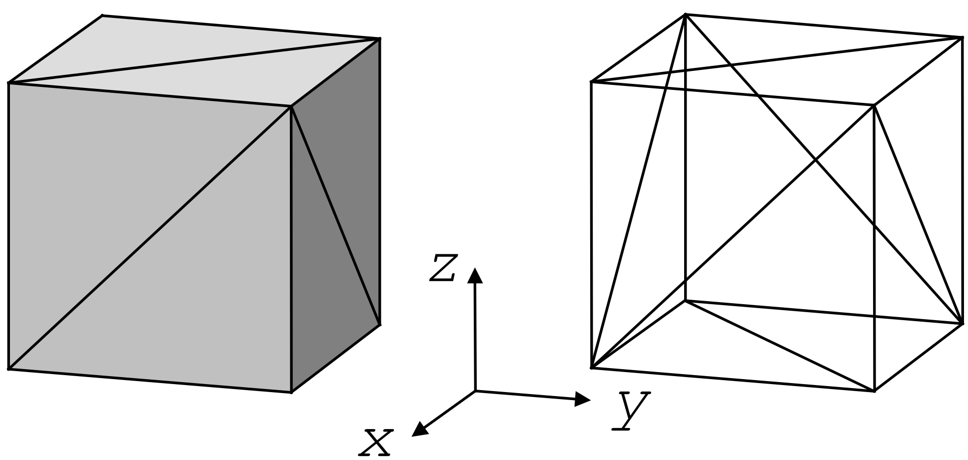
Figure 4. Connecting nearby facets

However, the resulting part or parts may not resemble the desired shape. If the file has many separate groups of disconnected facets, then a new part will be created for each group, although the file might only be expected to contain one part. Also, it is possible that the front ofone facet will be connected to the back of another facet, producing a "mobius part" that doesn't represent a valid solid. Fortunately, Admesh provides clues to such occurrences in its summary report. If the number of parts listed by Admesh is greater than expected, then the part may need to be processed again with different parameters to produce the desired result. Also, if the "Backwards edges" parameter is greater than zero, then the part is not a valid solid and should be reprocessed by Admesh using different parameters.

Figure 5 shows an STL model of a cube that has a hole in the top surface. The filling holes process of Admesh was able to repair the STL file by adding two facets to produce the cube shown in the bottom half of the figure. Appendix B contains the listing of the STL file before and after processing, as well as the processing summary produced by Admesh.



Cube missing two facets on the top face.



Cube after being processed by Admesh.

Figure 5. Filling holes

Fixing Normals

According to the STL specification, the three vertices of each facet should follow the "right hand rule"; that is, the vertices should be oriented counter-clockwise when viewed from the outside of the part. Given three vertices of each facet, the normal vector can be generated by calculating the cross product of two adjacent vectors as follows:

$$\mathbf{N} = \mathbf{T_1} \times \mathbf{T_2}$$

where

$$\mathbf{T_1} = \mathbf{V_2} - \mathbf{V_1}$$

and

$$\mathbf{T_2} = \mathbf{V_3} - \mathbf{V_1}$$

$\mathbf{V_1}$, $\mathbf{V_2}$, and $\mathbf{V_3}$ represent the three vertices of the facet (Glassner, 1990). The unit normal is as follows:

$$\frac{\mathbf{N}}{|\mathbf{N}|}$$

A common problem with STL files is that some or all of facets are oriented backwards; i.e. the vertices are listed in a clockwise direction when viewed from outside the part. Admesh repairs the orientation of facets according to the following algorithm:

1. Generate the neighbors list for all facets.

2. Inspect the first neighbor of the first facet to determine the orientation of its vertices.
3. If the neighboring facet is oriented in the same direction as the first facet, mark the neighbor as checked. If the neighboring facet is oriented opposite to the first facet, reverse the neighbor, and mark it as checked.
4. Repeat steps 2 and 3 until all facets have been checked.

The above algorithm ensures that all connected facets are oriented in the same direction. However, if the first facet is backwards, then all facets will be backwards. The correct orientation for the entire part is determined after the volume has been calculated. If the volume is negative, then all of the facets must be backwards. Admesh then reverses all facets to set the correct orientation.

An STL file contains a normal vector for each facet. Since the normal vector can be easily calculated given the three vertices of the facet, the normal vector that is included in the STL file for each facet is redundant. In practice, occasionally the normal vector included in the STL file doesn't match the calculated normal. Admesh calculates the normal for each facet and compares it with the normal from the STL file. If both normals match within a small tolerance, then the original normal was correct. If the normals are different, then the counter for "normals fixed" is incremented. In either case, the original normal is replaced with the calculated normal.

Calculating Part Volume

The volume of a part is useful for some rapid prototyping systems to determine how much material will be used to create the part. The volume is also used to determine the correct orientation of facet vertices. Part volume is calculated as follows:

1. Choose an arbitrary point near or on the part.
2. Initialize the volume variable to zero.
3. Calculate the volume created by the tetrahedron formed between the point and the first facet.
4. Add the calculated volume to the total volume.
5. Repeat steps 2 and 3 for all facets.

The above algorithm works whether or not the original point chosen is inside the part. If the original point is below the current facet, then the volume will be positive. If the original point is above the facet, then the volume will be negative. When all of the volumes are summed, the resulting value is the total part volume.

CHAPTER 4

ADMESH USER MANUAL

This chapter describes the use of Admesh version 0.95. Admesh is a program for processing triangulated solid meshes. Currently, Admesh only reads the STL file format that is used for rapid prototyping applications, although it can write STL, VRML, OFF, and DXF files.

Installation

To install Admesh, you will need a system with a C compiler, unless there is a precompiled binary available. On a UNIX system, follow these steps:

1. Get the file admesh-0.95.tar.gz
2. Extract the archive. i.e. type something like the following:

   tar -zxvf admesh-0.95.tar.gz

   or if that doesn't work, try the following:

   cat admesh-0.95.tar.gz | gzip -d | tar xvf -

   The source files will be extracted into a directory called admesh-0.95

3. cd admesh-0.95

4. Enter the following:

./configure

make

This should create an executable file called admesh. Admesh consists of only one stand-alone executable and there are no configuration files or environment variables to be set.

Admesh can be compiled with a 32-bit compiler under Windows, but the user will need to create a makefile or project file that works with that system.

Invoking Admesh

Admesh is executed as follows:

admesh [option]... file

By default, Admesh performs all of the mesh checking and repairing options on the input file. This means that it checks exact, nearby, remove-unconnected, fill-holes, normal-directions, and normal-values. The file type (ASCII or binary) is automatically detected. Important: Unless one of the --write options is used, the repaired STL file will not be saved. The original input file is never modified unless it is specified by one of the --write options. If the following command line was input

```
admesh sphere.stl
```

the file sphere.stl would be opened and read, it would be checked and fixed if necessary, and the results of processing would be printed out. The repaired STL file would not be saved since it wasn't specified with one of the --write options.

The default value for tolerance is the length of the shortest edge of the mesh. The default number of iterations is 2, and the default increment is 0.01% of the diameter of a sphere that encloses the entire mesh.

If any of the options --exact, --nearby, --remove-unconnected, --fill-holes, --normal-directions, --reverse-all, --normal-values, or --no-check are given, then no other checks besides those specified will be performed unless they are required by Admesh before the specified check can be done. For example the following command line:

```
admesh --remove-unconnected \
--write-binary-stl=spherefix.stl sphere.stl
```

would first do an exact check because it is required, and then the unconnected facets would be removed. The results would be printed and the repaired file would be saved as spherefix.stl. No other checks would be done.

Examples

To perform all checks except for nearby, the following command line would be used:

```
admesh --exact --remove-unconnected --fill-holes \
     --normal-directions --normal-values sphere.stl
```

The same results could be achieved using the short options:

```
admesh -fudev sphere.stl
```

The following command lines do the same thing:

```
admesh sphere.stl
admesh -fundev sphere.stl
admesh -f -u -n -d -e -v sphere.stl
```

since the -fundev options are the default options. To eliminate one of the checks, just remove the letter of the check to eliminate.

Option Summary

Admesh supports the following options, grouped by type.

Mesh Transformation Options

--x-rotate=angle   Rotate CCW about x-axis by angle degrees

--y-rotate=angle   Rotate CCW about y-axis by angle degrees

--z-rotate=angle   Rotate CCW about z-axis by angle degrees

--xy-mirror   Mirror about the xy plane

--yz-mirror   Mirror about the yz plane

--xz-mirror   Mirror about the xz plane

--scale=factor   Scale the file by factor (multiply by factor)

--translate=x,y,z   Translate the file to x, y, and z

--merge=name   Merge file called name with input file

Mesh Checking and Repairing Options

-e, --exact   Only check for perfectly matched edges

-n, --nearby   Find and connect nearby facets.

-t, --tolerance=tol   Initial tolerance to use for nearby check = tol

-i, --iterations=I   Number of iterations for nearby check = i

-m, --increment=inc   Amount to increment tolerance between iterations

-u, --remove-unconnected Remove facets that have 0 neighbors

-f, --fill-holes Add facets to fill holes

-d, --normal-directions Check and fix orientation of normals (i.e. clockwise or counterclockwise)

--reverse-all Reverse the directions of all facets and normals

-v, --normal-values Check and fix normal values

-c, --no-check Don't do any checks on the input file

File Output Options

-b, --write-binary-stl=name Output a binary STL file called name

-a, --write-ascii-stl=name Output an ASCII STL file called name

--write-off=name Output a Geomview OFF format file called name

--write-dxf=name Output a DXF format file called name

--write-vrml=name Output a VRML 1.0 format file called name

Miscellaneous Options

--help Display this help and exit

--version   Output version information and exit

Mesh Transformation Options

--x-rotate=angle

--y-rotate=angle

--z-rotate=angle

Rotate the entire mesh about the specified axis by the given number of degrees. The rotation is counter-clockwise about the axis as seen by looking along the positive axis towards the origin, assuming a right-handed coordinate system.

--xy-mirror

--yz-mirror

--xz-mirror

Mirror the mesh about the specified plane. Mirroring involves reversing the sign of all of the coordinates in a particular axis. For example, to mirror a mesh about the xy plane, the signs of all of the z coordinates in the mesh are reversed.

--scale=factor

Scale the mesh by the given factor. This multiplies all of the coordinates by the specified number. This option could be used to change the "units" (there are no units explicitly specified in an STL file) of the mesh. For example, to change a part from inches to millimeters, just use the --scale=25.4 option.

--translate=x,y,z

Translate the mesh to the position x,y,z. This moves the minimum x, y, and z values of the mesh to the specified position. For example, given a mesh that has the following initial minimum and maximum coordinate values:

```
Min X =  4.000000, Max X =  5.000000
Min Y =  1.000000, Max Y =  3.000000
Min Z = -7.000000, Max Z = -2.000000
```

if the option --translate=1,2,3 is specified, the final values will be:

```
Min X =  1.000000, Max X =  2.000000
Min Y =  2.000000, Max Y =  4.000000
Min Z =  3.000000, Max Z =  8.000000
```

The translate option is often used to translate a mesh with arbitrary minimum and maximum coordinates to 0,0,0. Usually, translation is also required when merging two files.

--merge=name

Merge the specified file with the input file. No translation is done, so if, for example, a file was merged with itself, the resulting file would end up with two meshes exactly the same, occupying exactly the same space. So generally, translations need to be done to the files being merged so that when the two meshes are merged into one, the resulting parts are properly spaced. If you know the nature of the parts to be merged, it is possible to nest one part inside the other. Note, however, that no warnings will be given if one part intersects with the other.

It is possible to place one part against another, with no space in between, but you will still end up with two separately defined parts. If such a mesh was made on a rapid-prototyping machine, the result would depend on the type of machine. Machines that use a photopolymer would produce a single solid part because the two parts would be "bonded" during the build process. Machines that use a cutting process would yield two or more parts.

A copy of a mesh can be made by using the --merge and --translate options at the same time. For example, given a file called block.stl with the following size:

```
Min X =  0.000000, Max X =  2.000000
Min Y =  0.000000, Max Y =  2.000000
Min Z =  0.000000, Max Z =  2.000000
```

to create a file called 2blocks.stl that contains two of the parts separated by 1 unit in the x direction, the following command line would be used:

```
admesh --translate=3,0,0 --merge=block.stl \
               --write-binary=2blocks.stl block.stl
```

This would yield a binary STL file called 2blocks.stl with the following size:

```
Min X =  0.000000, Max X =  5.000000
Min Y =  0.000000, Max Y =  2.000000
Min Z =  0.000000, Max Z =  2.000000
```

Mesh Checking and Repairing Options

-e, --exact

Check each facet of the mesh for its 3 neighbors. Since each facet is a triangle, there should be exactly 3 neighboring facets for every facet in the mesh. Since the mesh defines a solid, there should be no unconnected edges in the mesh. When this option is specified, the 3 neighbors of every facet are searched for and, if found, the neighbors are added to an internal list that keeps track of the neighbors of each facet. A facet is only considered a neighbor if two of its vertices exactly match two of the vertices of another facet. That means that there must be 0 difference between the x, y, and z coordinates of the two vertices of the first facet and the two vertices of the second facet.

Degenerate facets (facets with two or more vertices equal to each other) are removed during the exact check. No other changes are made to the mesh. An exact check is always done before any of the other checking and repairing options even if --exact isn't specified. There is one exception to this rule; no exact check needs to be done before the --normal-values option.

-n, --nearby

-t, --tolerance=tol

-i, --iterations=i

-m, --increment=inc

Checks each unconnected facet of the mesh for facets that are almost connected but not quite. Due to rounding errors and other factors, it is common for a mesh to have facets with neighbors that are very close but don't match exactly. Often, this difference is only in the 5th decimal place of the vertices, but these facets will not show up as neighbors during the exact check. This option finds these nearby neighbors and it changes their vertices so that they match exactly. The exact check is always done before the nearby check, so only facets that remain unconnected after the exact check are candidates for the nearby check.

The --tolerance=tol option is used to specify the distance that is searched for the neighboring facet. By default, the tolerance is set automatically by Admesh to be the length of the shortest edge of the mesh. This value is used because it makes it unlikely for a facet that shouldn't be a neighbor to be found and matched as a neighbor. If the tolerance is too big, then some facets could end up connected that should definitely not be connected. This could create a "mobius part" that is not a valid solid. If this occurs, it can be seen by checking the value of "Backwards edges" in the processing summary. (The number of backwards edges should be 0 for a valid solid.)

The --iterations=i and --increment=inc options are used together to gradually connect nearby facets using progressively larger tolerances. This helps to prevent incorrect connects but can also allow larger tolerances to be used. The --iterations option gives the number of times that facets are checked for nearby facets, each time using a larger tolerance. The --increment=inc option gives the amount that the tolerance is increased after each iteration. The number specified by 'inc' is added to the tolerance that was used in the previous iteration. If all of the facets are connected, no further nearby checks will be done.

-f, --fill-holes

Fill holes in the mesh by adding facets. This is done after the exact check and after nearby check (if any nearby check is done). If there are still unconnected facets, then facets will be added to the mesh, connecting the unconnected facets, until all of the holes have been filled. This is guaranteed to completely fix all unconnected facets. However, the resulting mesh may or may not be what the user expects.

-d, --normal-directions

Check and fix if necessary the directions of the facets. This only deals with whether the vertices of all the facets are oriented clockwise or counterclockwise, it doesn't check or modify the value of the normal vector. Every facet should have its vertices defined in a counterclockwise order when looked at from the outside of the part. This option will orient all of the vertices so that they are all facing in the same direction. It is possible that this option will make all of the facets face inwards instead of outwards. The algorithm tries to determine which direction is inside and outside by checking the value of the normal vector, so the chance is very good that the resulting mesh will be correct. It doesn't explicitly check to find which direction is inside and which is outside. However, when the part volume is calculated, if the volume is negative then it is assumed that the normal directions are all backwards and all of the facets are reversed.

--reverse-all

Reverses the directions of all of the facets and normals. If the --normal-directions option ended up making all of the facets face inwards instead of outwards, then this option can be used to reverse all of the facets. This option also fixes and updates the normal vector for each facet.

-v, --normal-values

Checks and fixes, if necessary, the normal vectors of every facet. The normal vector will point outward for a counterclockwise facet. The length of the normal vector will be 1.

-c, --no-check

Don't do any checks or modifications to the input file. By default, Admesh performs all processes (exact, nearby, remove-unconnected, fill-holes, normal-directions, and normal-values) on the input file. If the --no-check option is specified, no checks or modifications will be made on the input file. This could be used, for example, to translate an ASCII STL file to a binary STL file, with no modifications made. A command line such as the following might be used:

```
admesh --no-check --write-binary-stl=newblock.stl \
                    --translate=0,0,0 block.stl
```

This would open the file block.stl, translate it to 0,0,0 no checks would be performed and a binary STL file of the translated mesh would be written to newblock.stl.

-b, --write-binary-stl=name

-a, --write-ascii-stl=name

Write a binary STL file with the name specified. The input file is not modified by Admesh so the only way to preserve any modifications that have been made to the input file is to use one of the --write options. If the user wants to modify (overwrite) the input file, then the input file can also be specified for the --write option. For example, to convert an input ASCII STL file called sphere.stl to a binary STL file, overwriting the original file, and performing no checks, the following command line would be used:

```
admesh --write-binary-stl=sphere.stl \
                              --no-check sphere.stl
```

--help

Display the possible command line options with a short description, and then exit.

--version

Show the version information for Admesh, and then exit.

Admesh Output Summary

After Admesh has processed a mesh, it prints out a page of information about that mesh. The output looks like the following:

```
====== Results produced by ADMesh version 0.95 ==========
Input file     : sphere.stl
File type      : Binary STL file
Header         : Processed by ADMesh version 0.95
============== Size ==============
Min X      = -1.334557, Max X = 1.370952
Min Y      = -1.377953, Max Y = 1.377230
Min Z      = -1.373225, Max Z = 1.242838
========= Facet Status ========== Original ====== Final =
Number of facets                 :  3656          3656
Facets with 1 disconnected edge  :    18             0
Facets with 2 disconnected edges :     3             0
Facets with 3 disconnected edges :     0             0
Total disconnected facets        :    21             0
=== Processing Statistics ===   ===== Other Statistics ==
Number of parts       :     1      Volume   :  10.889216
```

```
Degenerate facets     :     0
Edges fixed           :    24
Facets removed        :     0
Facets added          :     0
Facets reversed       :     0
Backwards edges       :     0
Normals fixed         :     0
```

Description of Summary

The following describes the summary information line by line.

```
Input file     : sphere.stl
```

The name of the input STL file.

```
File type : Binary STL file
```

The type of input file. Currently, the only two possibilities are Binary STL file and ASCII STL file. Admesh automatically detects the type of input file.

```
Header    : Processed by ADMesh version 0.95
```

The header of the STL file. This string is within the first 80 bytes of a binary STL file or the first line of an ASCII STL file. This usually contains the name of the CAD system that has created that file, or the last program to process that file. Admesh puts its own string in the header when it saves the file.

```
============== Size ==============
Min X      = -1.334557, Max X = 1.370952
Min Y      = -1.377953, Max Y = 1.377230
Min Z      = -1.373225, Max Z = 1.242838
```

This section gives the boundaries of the mesh. The mesh will fit just inside a box of this size.

```
========= Facet Status ========== Original ====== Final =
Number of facets                 :  3656          3656
Facets with 1 disconnected edge  :    18             0
Facets with 2 disconnected edges :     3             0
Facets with 3 disconnected edges :     0             0
Total disconnected facets        :    21             0
```

This section contains information about the quality of the mesh before and after processing by Admesh. The number of facets is indicative of the complexity and accuracy of the mesh. Disconnected facets will fall into 3 categories. Some facets will have only one disconnected edge, some will have 2 edges disconnected, and some will have all 3 edges disconnected. Of course, for a valid solid mesh, there should be 0 disconnected facets.

```
=== Processing Statistics ===
Number of parts       :     1
```

This is the total number of separate parts in the file. This can be a very useful indication of whether your file is correct. Sometimes, the user of the CAD system that creates the mesh just puts several pieces together next to each other, and then outputs the mesh. This might not cause any problems for a rapid prototyping system that uses a photopolymer because all of the parts will be "glued" together anyway during the build. A rapid prototyping machine that is based on cutting will cut each one of the parts individually and the result will be many parts that need to be glued together. The number of parts is counted during --normal-directions, so if the --normal-directions check is eliminated, then the number of parts will read 0.

Degenerate facets : 0

This is the number of degenerate facets in the input file. A degenerate facet is a facet that has two or more vertices exactly the same. The resulting facet is just a line (if two vertices are the same) or could even be a point (if all 3 vertices are the same). These facets add no information to the file and are removed by Admesh during processing.

Edges fixed : 24

This is the total number of edges that were fixed by moving the vertices slightly during the nearby check. This does not include facets that were added by --fill-holes.

Facets removed : 0

This is the total number of facets removed. There are two cases where facets might be removed. First, all degenerate facets in the input file are removed. Second, if there are any completely unconnected facets (facets with 3 disconnected edges) after the exact and nearby checks, then these facets will be removed by --remove-unconnected.

Facets added   :   0

This is the number of facets that have been added by Admesh to the original mesh. Facets are only added during --fill-holes. So this number represents the number of facets that had to be added to fill all of the holes, if any, in the original mesh.

Facets reversed   :   0

This is the number of facets that were reversed during --normal-directions. This only relates to the order of the vertices of the facet (clockwise or counterclockwise), it has nothing to do with the value of the normal vector.

Backwards edges   :   0

This is the number of edges that are backwards. After Admesh has finished all of the checks and processing, it verifies the results. If the --normal-directions check has been done then the number of backwards edges should be zero. If it is not, then a "mobius part" has been created which is not a valid solid mesh. In this case the original file should be processed again with Admesh using a smaller tolerance on the nearby check or with no nearby check.

```
Normals fixed    :     0
```

This is the number of normal vectors that have been fixed. During the normal-values check, Admesh calculates the value of every facet and compares the result with the normal vector from the input file. If the result is not within a fixed tolerance, then the normal is counted as fixed. However, for consistency, every normal vector is rewritten with the new calculated normal, even if the original normal was within tolerance.

```
===== Other Statistics =====
Volume   :  10.889216
```

This is the total volume of the part or parts in cubic units.

CHAPTER 5

RESULTS

To research errors with STL files and their frequency of occurrence, 150 STL files from many different sources were collected. Admesh then processed each of these files and the results are discussed below. There are two main types of errors, disconnected facets and incorrect normals. Disconnected facets occur when a facet has fewer than three adjacent neighboring facets. Incorrect normals are the result of backward orientation of the vertices, or an invalid normal vector.

Disconnected Facets

There were 68 files with disconnected facets, out of which 53 had fewer than 5%, 11 had 5%-20%, and 4 had more than 20% disconnected facets.

Admesh was able to completely repair all disconnected facets in 43 of the 68 imperfect files during one or two iterations of the nearby check. The fact that such a high percentage of the files were fixed with the nearby check reveals that most of the disconnected facets were likely due to rounding or truncation error when the mesh was created.

The remaining 25 imperfect files were repaired by filling holes or by a combination of filling holes and a nearby check. Many of these files only required a small percentage of facets to be added to fill all holes. A few files needed a relatively large number of facets to be added, and the resulting mesh was likely not the desired result. These few seriously damaged STL files may require the mesh to be repaired interactively, or the model to be repaired in the CAD system before generating a new STL file.

Repairing Normals

Of the 150 STL files processed, 50 files had backward orientation of the vertices, or an invalid normal vector.

Ten files had the vertices oriented backwards, i.e. the vertices were listed clockwise when viewed from outside the part instead of counterclockwise as required by the STL specification. Admesh reversed the vertices for all backward facets.

The other 40 files had the value of the normal vector wrong. Admesh also repaired the values of these normals. Since the value of the normal vector is redundant given the three vertices of the facet, incorrect normal values aren't likely to affect the quality of the model since software that reads an STL file should calculate the normals from the vertex data. Some software that generates STL files will set the normal vector to zero to force calculation of the normal vector directly from the vertices.

CHAPTER 6

STL AND OTHER FORMATS

STL is the format most used for input to rapid prototyping machines. However there are other file formats that maintain similar information to STL. Admesh can convert STL to a few of these formats. This allows STL data to be imported into software that doesn't read STL files, but does read an alternate format output by Admesh. Each of these formats has its advantages and disadvantages.

Advantages and Disadvantages of STL

Since each entry in an STL file is a single triangular facet, it is very easy to write the file. Most 3D CAD systems tessellate the part for rendering the object on the screen, so the STL output is a simple extension of this code. The simplicity of the STL format is one of the keys to its success.

There are a few disadvantages to the STL format. One of the most obvious quirks of the format is its inclusion of the normal vector. The three vertices of each facet determine the value of the normal vector, not the normal information included in the STL file. The argument could be made that including it in the file eliminates the need to recalculate it every time the file is read. In practice, the time required for a computer to calculate normal vertices is relatively insignificant, and is likely to be less than the time it takes to read the normal information from disk.

To guarantee that a physical model can be built from the STL file, the facets must be connected completely such that they form a continuous surface. Since the STL format represents each facet individually, there is no intrinsic information about the connectivity of the facets. The only way to determine if an STL model is continuous is to check if each facet has three neighboring facets. The neighbors are found by reading each facet from the file and searching for three other facets that have two vertices with exactly the same coordinates.

DXF

Autodesk developed the DXF file format for exporting CAD data from AutoCAD. DXF is an ASCII format that is much more complicated than STL since it has support for text, layers, colors, dimensions, and many other entities. The DXF file that is exported by Admesh uses a very small subset of the DXF format. Admesh only uses the 3DFACE entity type, which represents four corners of a facet. Since STL only contains three vertices per facet, the fourth vertex of the 3DFACE is set equal to the third vertex. The DXF file that is output by Admesh is different from the STL file primarily in syntax. The method of defining the mesh is the same as STL and the DXF file contains the same data, minus the normal vectors.

The DXF format can be read by many software packages that don't read STL. For example, some 3D rendering and animation packages can only read DXF, so converting STL to DXF allows rapid prototyping models to be imported into these packages.

OFF

OFF is an ASCII format that is used by Geomview, a program written at the Geometry Center at the University of Minnesota (Phillips, 1996). OFF is a good example of a "shared vertex" format. A shared vertex file only lists the coordinates of each vertex once. Then for each facet, a list of integer numbers point to each vertex of the facet. Since the shared vertex format only lists each vertex once instead of multiple times, this format prevents unmatched vertices caused by rounding or truncation. Shared vertex formats are also more compact than unshared formats like STL since the coordinates of each vertex are only listed once and then the facets are defined with 3 integers that point to the vertices. Following is the OFF file generated by Admesh from the tetra.stl file shown in Figure 2.

```
OFF
4 4 0
     0.000000 1.000000 0.000000
     1.000000 0.000000 0.000000
     0.000000 0.000000 0.000000
     0.000000 0.000000 1.000000
     3 0 1 2
     3 0 2 3
     3 2 1 3
     3 1 0 3
```

Lines 3-6 list the coordinates of the vertices. In lines 7-10, the first number represents the number of vertices per facet, and the three numbers that follow correspond to the appropriate vertex coordinate that was defined in lines 3-6.

The OFF file is much more compact than STL, in this case the OFF is only 168 bytes compared to 1,297 bytes for the STL file of the same object, about 87% smaller. The binary STL file of the tetrahedron is 284 bytes, but the binary OFF file of the same object is only 122 bytes.

VRML

VRML is an ASCII format that was designed to represent 3D models on the World Wide Web (Bell, 1996). The VRML file exported by Admesh is also a shared vertex format. VRML has support for many attributes such as textures and lighting, but the VRML file output by Admesh uses a small subset of the available attributes.

VRML can be read by many rendering and animation packages that can't read STL, so it is a useful format for exporting STL files to those packages.

Figure 6 shows a VRML file that was generated by Admesh from the tetra.stl shown in Figure 2.

```
#VRML V1.0 ascii

Separator {
     DEF STLShape ShapeHints {
          vertexOrdering COUNTERCLOCKWISE
          faceType CONVEX
          shapeType SOLID
          creaseAngle 0.0
     }
     DEF STLModel Separator {
          DEF STLColor Material {
          emissiveColor 0.700000 0.700000 0.000000
          }
          DEF STLVertices Coordinate3 {
               point [
                    0.000000 1.000000 0.000000,
                    1.000000 0.000000 0.000000,
                    0.000000 0.000000 0.000000,
                    0.000000 0.000000 1.000000]
          }
          DEF STLTriangles IndexedFaceSet {
               coordIndex [
                    0, 1, 2, -1,
                    0, 2, 3, -1,
                    2, 1, 3, -1,
                    1, 0, 3, -1]
          }
     }
}
```

Figure 6. VRML file

CHAPTER 7

CONCLUSION

The STL format is the most used format for rapid prototyping. As this thesis showed, however, there are a number of common errors in STL files. Before using a rapid prototyping machine to build a model, the STL file should be checked for errors.

This thesis presented a program called Admesh that is a fast and easy to use program that can analyze and, if necessary, repair STL files. In most cases, Admesh can repair the errors in STL files without requiring the user to manually correct the file or to regenerate the file from the CAD system.

Admesh can also convert STL files to a number of other formats. These formats allow rapid prototyping models to be used in applications such as 3D animation and rendering, and to be viewed over the Internet.

Since the first version of Admesh was made freely available on the Internet in 1995, it has been quite popular. Many people worldwide have used Admesh on machines ranging from inexpensive PCs to Cray supercomputers.

There have been several new versions of Admesh since the first release and a number of features were added. Admesh will continue to be supported, with new features and support for additional file formats.

APPENDICES

APPENDIX A

ADMESH SOURCE CODE

stl.h

```
/*  ADMesh -- process triangulated solid meshes
 *  Copyright (C) 1995, 1996  Anthony D. Martin
 *
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License as
 *  published by the Free Software Foundation; either version 2, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc.
 *  59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 *  Questions, comments, suggestions, etc to <amartin@engr.csulb.edu>
 */

#include <stdio.h>

#define STL_MAX(A,B) ((A)>(B)? (A):(B))
#define STL_MIN(A,B) ((A)<(B)? (A):(B))
#define ABS(X)  ((X) < 0 ? -(X) : (X))

#define LABEL_SIZE             80
#define NUM_FACET_SIZE         4
#define HEADER_SIZE            84
#define STL_MIN_FILE_SIZE      284
#define ASCII_LINES_PER_FACET  7
#define SIZEOF_EDGE_SORT       24

typedef struct
{
  float x;
  float y;
  float z;
}stl_vertex;

typedef struct
{
  float x;
  float y;
  float z;
}stl_normal;

typedef char stl_extra[2];

typedef struct
```

```
{
  stl_normal normal;
  stl_vertex vertex[3];
  stl_extra  extra;
}stl_facet;
#define SIZEOF_STL_FACET       50

typedef enum {binary, ascii} stl_type;

typedef struct
{
  stl_vertex p1;
  stl_vertex p2;
  int        facet_number;
}stl_edge;

typedef struct stl_hash_edge
{
  unsigned       key[6];
  int            facet_number;
  int            which_edge;
  struct stl_hash_edge  *next;
}stl_hash_edge;

typedef struct
{
  int   neighbor[3];
  char  which_vertex_not[3];
}stl_neighbors;

typedef struct
{
  int   vertex[3];
}v_indices_struct;

typedef struct
{
  char          header[81];
  stl_type      type;
  int           number_of_facets;
  stl_vertex    max;
  stl_vertex    min;
  stl_vertex    size;
  float         bounding_diameter;
  float         shortest_edge;
  float         volume;
  unsigned      number_of_blocks;
  int           connected_edges;
  int           connected_facets_1_edge;
  int           connected_facets_2_edge;
  int           connected_facets_3_edge;
  int           facets_w_1_bad_edge;
  int           facets_w_2_bad_edge;
  int           facets_w_3_bad_edge;
```

```
  int           original_num_facets;
  int           edges_fixed;
  int           degenerate_facets;
  int           facets_removed;
  int           facets_added;
  int           facets_reversed;
  int           backwards_edges;
  int           normals_fixed;
  int           number_of_parts;
  int           malloced;
  int           freed;
  int           facets_malloced;
  int           collisions;
  int           shared_vertices;
  int           shared_malloced;
}stl_stats;

typedef struct
{
  FILE          *fp;
  stl_facet     *facet_start;
  stl_edge      *edge_start;
  stl_hash_edge **heads;
  stl_hash_edge *tail;
  int           M;
  stl_neighbors *neighbors_start;
  v_indices_struct *v_indices;
  stl_vertex    *v_shared;
  stl_stats     stats;
}stl_file;


extern void stl_open(stl_file *stl, char *file);
extern void stl_close(stl_file *stl);
extern void stl_stats_out(stl_file *stl, FILE *file, char
*input_file);
extern void stl_print_edges(stl_file *stl, FILE *file);
extern void stl_print_neighbors(stl_file *stl, char *file);
extern void stl_write_ascii(stl_file *stl, char *file, char *label);
extern void stl_write_binary(stl_file *stl, char *file, char *label);
extern void stl_check_facets_exact(stl_file *stl);
extern void stl_check_facets_nearby(stl_file *stl, float tolerance);
extern void stl_remove_unconnected_facets(stl_file *stl);
extern void stl_write_vertex(stl_file *stl, int facet, int vertex);
extern void stl_write_facet(stl_file *stl, char *label, int facet);
extern void stl_write_edge(stl_file *stl, char *label, stl_hash_edge
edge);
extern void stl_write_neighbor(stl_file *stl, int facet);
extern void stl_write_quad_object(stl_file *stl, char *file);
extern void stl_verify_neighbors(stl_file *stl);
extern void stl_fill_holes(stl_file *stl);
extern void stl_fix_normal_directions(stl_file *stl);
extern void stl_fix_normal_values(stl_file *stl);
extern void stl_reverse_all_facets(stl_file *stl);
```

```
extern void stl_translate(stl_file *stl, float x, float y, float z);
extern void stl_scale(stl_file *stl, float factor);
extern void stl_rotate_x(stl_file *stl, float angle);
extern void stl_rotate_y(stl_file *stl, float angle);
extern void stl_rotate_z(stl_file *stl, float angle);
extern void stl_mirror_xy(stl_file *stl);
extern void stl_mirror_yz(stl_file *stl);
extern void stl_mirror_xz(stl_file *stl);
extern void stl_open_merge(stl_file *stl, char *file);
extern void stl_generate_shared_vertices(stl_file *stl);
extern void stl_write_off(stl_file *stl, char *file);
extern void stl_write_dxf(stl_file *stl, char *file, char *label);
extern void stl_write_vrml(stl_file *stl, char *file);
extern void stl_calculate_normal(float normal[], stl_facet *facet);
extern void stl_normalize_vector(float v[]);
extern void stl_calculate_volume(stl_file *stl);
```

admesh.c

```
#include <stdio.h>
#include <getopt.h>
#include <stdlib.h>


#include "stl.h"

static void usage(int status, char *program_name);

void
main(int argc, char **argv)
{
  stl_file stl_in;
  int      i;
  int      last_edges_fixed = 0;
  float    tolerance = 0;
  float    increment = 0;
  float    x_trans;
  float    y_trans;
  float    z_trans;
  float    scale_factor = 0;
  float    rotate_x_angle = 0;
  float    rotate_y_angle = 0;
  float    rotate_z_angle = 0;
  int      c;
  char     *program_name;
  char     *binary_name = NULL;
  char     *ascii_name = NULL;
  char     *merge_name = NULL;
  char     *off_name = NULL;
  char     *dxf_name = NULL;
  char     *vrml_name = NULL;
  int      fixall_flag = 1;          /* Default behavior is to fix
all. */
```

```
  int      exact_flag = 0;          /* All checks turned off by
default. */
  int      tolerance_flag = 0;            /* Is tolerance specified
on cmdline */
  int      nearby_flag = 0;
  int      remove_unconnected_flag = 0;
  int      fill_holes_flag = 0;
  int      normal_directions_flag = 0;
  int      normal_values_flag = 0;
  int      reverse_all_flag = 0;
  int      write_binary_stl_flag = 0;
  int      write_ascii_stl_flag = 0;
  int      generate_shared_vertices_flag = 0;
  int      write_off_flag = 0;
  int      write_dxf_flag = 0;
  int      write_vrml_flag = 0;
  int      translate_flag = 0;
  int      scale_flag = 0;
  int      rotate_x_flag = 0;
  int      rotate_y_flag = 0;
  int      rotate_z_flag = 0;
  int      mirror_xy_flag = 0;
  int      mirror_yz_flag = 0;
  int      mirror_xz_flag = 0;
  int      merge_flag = 0;
  int      help_flag = 0;
  int      version_flag = 0;

  int      iterations = 2;          /* Default number of iterations.
*/
  int      increment_flag = 0;
  char     *input_file = NULL;

  enum {rotate_x = 1000, rotate_y, rotate_z, merge, help, version,
      mirror_xy, mirror_yz, mirror_xz, scale, translate, reverse_all,
      off_file, dxf_file, vrml_file};

  struct option long_options[] =
    {
      {"exact",              no_argument,       NULL, 'e'},
      {"nearby",             no_argument,       NULL, 'n'},
      {"tolerance",          required_argument, NULL, 't'},
      {"iterations",         required_argument, NULL, 'i'},
      {"increment",          required_argument, NULL, 'm'},
      {"remove-unconnected", no_argument,       NULL, 'u'},
      {"fill-holes",         no_argument,       NULL, 'f'},
      {"normal-directions",  no_argument,       NULL, 'd'},
      {"normal-values",      no_argument,       NULL, 'v'},
      {"no-check",           no_argument,       NULL, 'c'},
      {"reverse-all",        no_argument,       NULL, reverse_all},
      {"write-binary-stl",   required_argument, NULL, 'b'},
      {"write-ascii-stl",    required_argument, NULL, 'a'},
      {"write-off",          required_argument, NULL, off_file},
      {"write-dxf",          required_argument, NULL, dxf_file},
```

```
    {"write-vrml",          required_argument, NULL, vrml_file},
    {"translate",           required_argument, NULL, translate},
    {"scale",               required_argument, NULL, scale},
    {"x-rotate",            required_argument, NULL, rotate_x},
    {"y-rotate",            required_argument, NULL, rotate_y},
    {"z-rotate",            required_argument, NULL, rotate_z},
    {"xy-mirror",           no_argument,       NULL, mirror_xy},
    {"yz-mirror",           no_argument,       NULL, mirror_yz},
    {"xz-mirror",           no_argument,       NULL, mirror_xz},
    {"merge",               required_argument, NULL, merge},
    {"help",                no_argument,       NULL, help},
    {"version",             no_argument,       NULL, version},
    {NULL, 0, NULL, 0}
  };

program_name = argv[0];
while((c = getopt_long(argc, argv, "et:i:m:nufdcvb:a:",
                 long_options, (int *) 0)) != EOF)
  {
    switch(c)
    {
     case 0:                 /* If *flag is not null */
      break;
     case 'e':
      exact_flag = 1;
      fixall_flag = 0;
      break;
     case 'n':
      nearby_flag = 1;
      fixall_flag = 0;
      break;
     case 't':
      tolerance_flag = 1;
      tolerance = atof(optarg);
      break;
     case 'i':
      iterations = atoi(optarg);
      break;
     case 'm':
      increment_flag = 1;
      increment = atof(optarg);
      break;
     case 'u':
      remove_unconnected_flag = 1;
      fixall_flag = 0;
      break;
     case 'f':
      fill_holes_flag = 1;
      fixall_flag = 0;
      break;
     case 'd':
      normal_directions_flag = 1;
      fixall_flag = 0;
      break;
```

```
        case 'v':
          normal_values_flag = 1;
          fixall_flag = 0;
          break;
        case 'c':
          fixall_flag = 0;
          break;
        case reverse_all:
          reverse_all_flag = 1;
          fixall_flag = 0;
          break;
        case 'b':
          write_binary_stl_flag = 1;
          binary_name = optarg;        /* I'm not sure if this is safe.
*/
          break;
        case 'a':
          write_ascii_stl_flag = 1;
          ascii_name = optarg;         /* I'm not sure if this is safe.
*/
          break;
        case off_file:
          generate_shared_vertices_flag = 1;
          write_off_flag = 1;
          off_name = optarg;
          break;
        case vrml_file:
          generate_shared_vertices_flag = 1;
          write_vrml_flag = 1;
          vrml_name = optarg;
          break;
        case dxf_file:
          write_dxf_flag = 1;
          dxf_name = optarg;
          break;
        case translate:
          translate_flag = 1;
          sscanf(optarg, "%f,%f,%f", &x_trans, &y_trans, &z_trans);
          break;
        case scale:
          scale_flag = 1;
          scale_factor = atof(optarg);
          break;
        case rotate_x:
          rotate_x_flag = 1;
          rotate_x_angle = atof(optarg);
          break;
        case rotate_y:
          rotate_y_flag = 1;
          rotate_y_angle = atof(optarg);
          break;
        case rotate_z:
          rotate_z_flag = 1;
          rotate_z_angle = atof(optarg);
```

```
        break;
       case mirror_xy:
        mirror_xy_flag = 1;
        break;
       case mirror_yz:
        mirror_yz_flag = 1;
        break;
       case mirror_xz:
        mirror_xz_flag = 1;
        break;
       case merge:
        merge_flag = 1;
        merge_name = optarg;
        break;
       case help:
        help_flag = 1;
        break;
       case version:
        version_flag = 1;
        break;
       default:
        usage(1, program_name);
        break;
      }
    }

  if(help_flag)
    {
      usage(0, program_name);
    }
  if(version_flag)
    {
      printf("ADMesh - version 0.95\n");
      exit(0);
    }

  if(optind == argc)
    {
      printf("No input file name given.\n");
      usage(1, program_name);
    }
  else
    {
      input_file = argv[optind];
    }

  printf("\
ADMesh version 0.95, Copyright (C) 1995, 1996 Anthony D. Martin\n\
ADMesh comes with NO WARRANTY.  This is free software, and you are
welcome to\n\
redistribute it under certain conditions.  See the file COPYING for
details.\n");
```

```
  printf("Opening %s\n", input_file);
  stl_open(&stl_in, input_file);

  if(rotate_x_flag)
    {
      printf("Rotating about the x axis by %f degrees...\n",
rotate_x_angle);
      stl_rotate_x(&stl_in, rotate_x_angle);
    }
  if(rotate_y_flag)
    {
      printf("Rotating about the y axis by %f degrees...\n",
rotate_y_angle);
      stl_rotate_y(&stl_in, rotate_y_angle);
    }
  if(rotate_z_flag)
    {
      printf("Rotating about the z axis by %f degrees...\n",
rotate_z_angle);
      stl_rotate_z(&stl_in, rotate_z_angle);
    }
  if(mirror_xy_flag)
    {
      printf("Mirroring about the xy plane...\n");
      stl_mirror_xy(&stl_in);
    }
  if(mirror_yz_flag)
    {
      printf("Mirroring about the yz plane...\n");
      stl_mirror_yz(&stl_in);
    }
  if(mirror_xz_flag)
    {
      printf("Mirroring about the xz plane...\n");
      stl_mirror_xz(&stl_in);
    }

  if(scale_flag)
    {
      printf("Scaling by factor %f...\n", scale_factor);
      stl_scale(&stl_in, scale_factor);
    }
  if(translate_flag)
    {
      printf("Translating to %f, %f, %f ...\n", x_trans, y_trans,
z_trans);
      stl_translate(&stl_in, x_trans, y_trans, z_trans);
    }
  if(merge_flag)
    {
      printf("Merging %s with %s\n", input_file, merge_name);
      stl_open_merge(&stl_in, merge_name);
    }
```

```
  if(exact_flag || fixall_flag || nearby_flag ||
remove_unconnected_flag
     || fill_holes_flag || normal_directions_flag)
    {
      printf("Checking exact...\n");
      exact_flag = 1;
      stl_check_facets_exact(&stl_in);
      stl_in.stats.facets_w_1_bad_edge =
      (stl_in.stats.connected_facets_2_edge -
       stl_in.stats.connected_facets_3_edge);
      stl_in.stats.facets_w_2_bad_edge =
      (stl_in.stats.connected_facets_1_edge -
       stl_in.stats.connected_facets_2_edge);
      stl_in.stats.facets_w_3_bad_edge =
      (stl_in.stats.number_of_facets -
       stl_in.stats.connected_facets_1_edge);
    }

  if(nearby_flag || fixall_flag)
    {
      if(!tolerance_flag)
      {
        tolerance = stl_in.stats.shortest_edge;
      }
      if(!increment_flag)
      {
        increment = stl_in.stats.bounding_diameter / 10000.0;
      }

      if(stl_in.stats.connected_facets_3_edge <
stl_in.stats.number_of_facets)
      {
        for(i = 0; i < iterations; i++)
          {
            if(stl_in.stats.connected_facets_3_edge <
             stl_in.stats.number_of_facets)
            {
              printf("\
Checking nearby. Tolerance= %f Iteration=%d of %d...",
                     tolerance, i + 1, iterations);
              stl_check_facets_nearby(&stl_in, tolerance);
              printf("  Fixed %d edges.\n",
                     stl_in.stats.edges_fixed - last_edges_fixed);
              last_edges_fixed = stl_in.stats.edges_fixed;
              tolerance += increment;
            }
            else
            {
              printf("\
All facets connected.  No further nearby check necessary.\n");
              break;
            }
          }
      }
```

```
      else
      {
        printf("All facets connected.  No nearby check necessary.\
n");
      }
    }

  if(remove_unconnected_flag || fixall_flag || fill_holes_flag)
    {
      if(stl_in.stats.connected_facets_3_edge <
stl_in.stats.number_of_facets)
      {
        printf("Removing unconnected facets...\n");
        stl_remove_unconnected_facets(&stl_in);
      }
      else
      printf("No unconnected need to be removed.\n");
    }

  if(fill_holes_flag || fixall_flag)
    {
      if(stl_in.stats.connected_facets_3_edge <
stl_in.stats.number_of_facets)
      {
        printf("Filling holes...\n");
        stl_fill_holes(&stl_in);
      }
      else
      printf("No holes need to be filled.\n");
    }

  if(reverse_all_flag)
    {
      printf("Reversing all facets...\n");
      stl_reverse_all_facets(&stl_in);
    }

  if(normal_directions_flag || fixall_flag)
    {
      printf("Checking normal directions...\n");
      stl_fix_normal_directions(&stl_in);
    }

  if(normal_values_flag || fixall_flag)
    {
      printf("Checking normal values...\n");
      stl_fix_normal_values(&stl_in);
    }

  /* Always calculate the volume.  It shouldn't take too long */
  printf("Calculating volume...\n");
  stl_calculate_volume(&stl_in);

  if(exact_flag)
```

```
    {
      printf("Verifying neighbors...\n");
      stl_verify_neighbors(&stl_in);
    }

  if(generate_shared_vertices_flag)
    {
      printf("Generating shared vertices...\n");
      stl_generate_shared_vertices(&stl_in);
    }

  if(write_off_flag)
    {
      printf("Writing OFF file %s\n", off_name);
      stl_write_off(&stl_in, off_name);
    }

  if(write_dxf_flag)
    {
      printf("Writing DXF file %s\n", dxf_name);
      stl_write_dxf(&stl_in, dxf_name, "Created by ADMesh version
0.95");
    }

  if(write_vrml_flag)
    {
      printf("Writing VRML file %s\n", vrml_name);
      stl_write_vrml(&stl_in, vrml_name);
    }

  if(write_ascii_stl_flag)
    {
      printf("Writing ascii file %s\n", ascii_name);
      stl_write_ascii(&stl_in, ascii_name,
                    "Processed by ADMesh version 0.95");
    }
  if(write_binary_stl_flag)
    {
      printf("Writing binary file %s\n", binary_name);
      stl_write_binary(&stl_in, binary_name,
                     "Processed by ADMesh version 0.95");
    }
  if(exact_flag)
    {
      stl_stats_out(&stl_in, stdout, input_file);
    }
  stl_close(&stl_in);
  exit(0);
}

static void
usage(int status, char *program_name)
{
  if(status != 0)
```

```
    {
      fprintf(stderr, "Try '%s --help' for more information.\n",
program_name);
    }
  else
    {
      printf("\n\
ADMesh version 0.95\n\
Copyright (C) 1995, 1996  Anthony D. Martin\n\
Usage: %s [OPTION]... file\n", program_name);
      printf("\n\
     --x-rotate=angle     Rotate CCW about x-axis by angle degrees\n\
     --y-rotate=angle     Rotate CCW about y-axis by angle degrees\n\
     --z-rotate=angle     Rotate CCW about z-axis by angle degrees\n\
     --xy-mirror          Mirror about the xy plane\n\
     --yz-mirror          Mirror about the yz plane\n\
     --xz-mirror          Mirror about the xz plane\n\
     --scale=factor       Scale the file by factor (multiply by
factor)\n\
     --translate=x,y,z    Translate the file to x, y, and z\n\
     --merge=name         Merge file called name with input file\n\
 -e, --exact              Only check for perfectly matched edges\n\
 -n, --nearby             Find and connect nearby facets. Correct bad
facets\n\
 -t, --tolerance=tol      Initial tolerance to use for nearby check =
tol\n\
 -i, --iterations=i       Number of iterations for nearby check = i\
n\
 -m, --increment=inc      Amount to increment tolerance after
iteration=inc\n\
 -u, --remove-unconnected Remove facets that have 0 neighbors\n\
 -f, --fill-holes         Add facets to fill holes\n\
 -d, --normal-directions  Check and fix direction of normals(ie cw,
ccw)\n\
     --reverse-all        Reverse the directions of all facets and
normals\n\
 -v, --normal-values      Check and fix normal values\n\
 -c, --no-check           Don't do any check on input file\n\
 -b, --write-binary-stl=name   Output a binary STL file called name\
n\
 -a, --write-ascii-stl=name    Output an ascii STL file called name\
n\
     --write-off=name     Output a Geomview OFF format file called
name\n\
     --write-dxf=name     Output a DXF format file called name\n\
     --write-vrml=name    Output a VRML format file called name\n\
     --help               Display this help and exit\n\
     --version            Output version information and exit\n\
\n\
The functions are executed in the same order as the options shown
here.\n\
So check here to find what happens if, for example, --translate and
--merge\n\
```

```
options are specified together.  The order of the options specified
on the\n\
command line is not important.\n");
    }
  exit(status);
}
```

connect.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "stl.h"


static void stl_match_neighbors_exact(stl_file *stl,
                   stl_hash_edge *edge_a, stl_hash_edge *edge_b);
static void stl_match_neighbors_nearby(stl_file *stl,
                         stl_hash_edge *edge_a, stl_hash_edge
*edge_b);
static void stl_record_neighbors(stl_file *stl,
                         stl_hash_edge *edge_a, stl_hash_edge
*edge_b);
static void stl_initialize_facet_check_exact(stl_file *stl);
static void stl_initialize_facet_check_nearby(stl_file *stl);
static void stl_load_edge_exact(stl_file *stl, stl_hash_edge *edge,
                   stl_vertex *a, stl_vertex *b);
static int stl_load_edge_nearby(stl_file *stl, stl_hash_edge *edge,
                   stl_vertex *a, stl_vertex *b, float tolerance);
static void insert_hash_edge(stl_file *stl, stl_hash_edge edge,
                  void (*match_neighbors)(stl_file *stl,
                stl_hash_edge *edge_a, stl_hash_edge *edge_b));
static int stl_get_hash_for_edge(int M, stl_hash_edge *edge);
static int stl_compare_function(stl_hash_edge *edge_a, stl_hash_edge
*edge_b);
static void stl_free_edges(stl_file *stl);
static void stl_remove_facet(stl_file *stl, int facet_number);
static void stl_change_vertices(stl_file *stl, int facet_num, int
vnot,
                   stl_vertex new_vertex);
static void stl_which_vertices_to_change(stl_file *stl, stl_hash_edge
*edge_a,
                       stl_hash_edge *edge_b, int *facet1, int
*vertex1,
                       int *facet2, int *vertex2,
                       stl_vertex *new_vertex1, stl_vertex
*new_vertex2);
static void stl_remove_degenerate(stl_file *stl, int facet);
static void stl_add_facet(stl_file *stl, stl_facet *new_facet);
extern int stl_check_normal_vector(stl_file *stl,
                           int facet_num, int normal_fix_flag);
static void stl_update_connects_remove_1(stl_file *stl, int
facet_num);
```

```
void
stl_check_facets_exact(stl_file *stl)
{
/* This function builds the neighbors list.  No modifications are
made
 *  to any of the facets.  The edges are said to match only if all
six
 *  floats of the first edge matches all six floats of the second
edge.
 */

  stl_hash_edge  edge;
  stl_facet      facet;
  int            i;
  int            j;

  stl->stats.connected_edges = 0;
  stl->stats.connected_facets_1_edge = 0;
  stl->stats.connected_facets_2_edge = 0;
  stl->stats.connected_facets_3_edge = 0;

  stl_initialize_facet_check_exact(stl);

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      facet = stl->facet_start[i];

      if(   !memcmp(&facet.vertex[0], &facet.vertex[1],
                sizeof(stl_vertex))
       || !memcmp(&facet.vertex[1], &facet.vertex[2],
                sizeof(stl_vertex))
       || !memcmp(&facet.vertex[0], &facet.vertex[2],
                sizeof(stl_vertex)))
      {
        stl->stats.degenerate_facets += 1;
        stl_remove_facet(stl, i);
        i--;
        continue;

      }
      for(j = 0; j < 3; j++)
      {
        edge.facet_number = i;
        edge.which_edge = j;
        stl_load_edge_exact(stl, &edge, &facet.vertex[j],
                        &facet.vertex[(j + 1) % 3]);

        insert_hash_edge(stl, edge, stl_match_neighbors_exact);
      }
    }
  stl_free_edges(stl);
}
```

```
static void
stl_load_edge_exact(stl_file *stl, stl_hash_edge *edge,
                stl_vertex *a, stl_vertex *b)
{

  float diff_x;
  float diff_y;
  float diff_z;
  float max_diff;


  diff_x = ABS(a->x - b->x);
  diff_y = ABS(a->y - b->y);
  diff_z = ABS(a->z - b->z);
  max_diff = STL_MAX(diff_x, diff_y);
  max_diff = STL_MAX(diff_z, max_diff);
  stl->stats.shortest_edge = STL_MIN(max_diff, stl-
>stats.shortest_edge);

  if(diff_x == max_diff)
    {
      if(a->x > b->x)
      {
        memcpy(&edge->key[0], a, sizeof(stl_vertex));
        memcpy(&edge->key[3], b, sizeof(stl_vertex));
      }
      else
      {
        memcpy(&edge->key[0], b, sizeof(stl_vertex));
        memcpy(&edge->key[3], a, sizeof(stl_vertex));
        edge->which_edge += 3; /* this edge is loaded backwards */
      }
    }
  else if(diff_y == max_diff)
    {
      if(a->y > b->y)
      {
        memcpy(&edge->key[0], a, sizeof(stl_vertex));
        memcpy(&edge->key[3], b, sizeof(stl_vertex));
      }
      else
      {
        memcpy(&edge->key[0], b, sizeof(stl_vertex));
        memcpy(&edge->key[3], a, sizeof(stl_vertex));
        edge->which_edge += 3; /* this edge is loaded backwards */
      }
    }
  else
    {
      if(a->z > b->z)
      {
        memcpy(&edge->key[0], a, sizeof(stl_vertex));
        memcpy(&edge->key[3], b, sizeof(stl_vertex));
```

```
      }
      else
      {
        memcpy(&edge->key[0], b, sizeof(stl_vertex));
        memcpy(&edge->key[3], a, sizeof(stl_vertex));
        edge->which_edge += 3; /* this edge is loaded backwards */
      }
    }
}

static void
stl_initialize_facet_check_exact(stl_file *stl)
{
  int i;

  stl->stats.malloced = 0;
  stl->stats.freed = 0;
  stl->stats.collisions = 0;


  stl->M = 81397;

  for(i = 0; i < stl->stats.number_of_facets ; i++)
    {
      /* initialize neighbors list to -1 to mark unconnected edges */
      stl->neighbors_start[i].neighbor[0] = -1;
      stl->neighbors_start[i].neighbor[1] = -1;
      stl->neighbors_start[i].neighbor[2] = -1;
    }

  stl->heads = calloc(stl->M, sizeof(*stl->heads));
  if(stl->heads == NULL) perror("stl_initialize_facet_check_exact");

  stl->tail = malloc(sizeof(stl_hash_edge));
  if(stl->tail == NULL) perror("stl_initialize_facet_check_exact");

  stl->tail->next = stl->tail;

  for(i = 0; i < stl->M; i++)
    {
      stl->heads[i] = stl->tail;
    }
}

static void
insert_hash_edge(stl_file *stl, stl_hash_edge edge,
                  void (*match_neighbors)(stl_file *stl,
                stl_hash_edge *edge_a, stl_hash_edge *edge_b))
{
  stl_hash_edge *link;
  stl_hash_edge *new_edge;
  stl_hash_edge *temp;
  int            chain_number;
```

```
  chain_number = stl_get_hash_for_edge(stl->M, &edge);

  link = stl->heads[chain_number];

  if(link == stl->tail)
    {
      /* This list doesn't have any edges currently in it.  Add this
one. */
      new_edge = malloc(sizeof(stl_hash_edge));
      if(new_edge == NULL) perror("insert_hash_edge");
      stl->stats.malloced++;
      *new_edge = edge;
      new_edge->next = stl->tail;
      stl->heads[chain_number] = new_edge;
      return;
    }
  else  if(!stl_compare_function(&edge, link))
    {
      /* This is a match.  Record result in neighbors list. */
      match_neighbors(stl, &edge, link);
      /* Delete the matched edge from the list. */
      stl->heads[chain_number] = link->next;
      free(link);
      stl->stats.freed++;
      return;
    }
  else
    {                  /* Continue through the rest of the list */
      for(;;)
      {
        if(link->next == stl->tail)
          {
            /* This is the last item in the list. Insert a new edge.
*/
            new_edge = malloc(sizeof(stl_hash_edge));
            if(new_edge == NULL) perror("insert_hash_edge");
            stl->stats.malloced++;
            *new_edge = edge;
            new_edge->next = stl->tail;
            link->next = new_edge;
            stl->stats.collisions++;
            return;
          }
        else  if(!stl_compare_function(&edge, link->next))
          {
            /* This is a match.  Record result in neighbors list. */
            match_neighbors(stl, &edge, link->next);

            /* Delete the matched edge from the list. */
            temp = link->next;
            link->next = link->next->next;
            free(temp);
            stl->stats.freed++;
            return;
```

```
          }
        else
          {
            /* This is not a match.  Go to the next link */
            link = link->next;
            stl->stats.collisions++;
          }
      }
    }
}


static int
stl_get_hash_for_edge(int M, stl_hash_edge *edge)
{
  return ((edge->key[0] / 23 + edge->key[1] / 19 + edge->key[2] / 17
         + edge->key[3] /13  + edge->key[4] / 11 + edge->key[5] / 7 )
% M);
}

static int
stl_compare_function(stl_hash_edge *edge_a, stl_hash_edge *edge_b)
{
  if(edge_a->facet_number == edge_b->facet_number)
    {
      return 1;               /* Don't match edges of the same facet
*/
    }
  else
    {
      return memcmp(edge_a, edge_b, SIZEOF_EDGE_SORT);
    }
}

void
stl_check_facets_nearby(stl_file *stl, float tolerance)
{
  stl_hash_edge  edge[3];
  stl_facet      facet;
  int            i;
  int            j;


  if(   (stl->stats.connected_facets_1_edge == stl-
>stats.number_of_facets)
     && (stl->stats.connected_facets_2_edge == stl-
>stats.number_of_facets)
     && (stl->stats.connected_facets_3_edge == stl-
>stats.number_of_facets))
    {
      /* No need to check any further.  All facets are connected */
      return;
    }
```

```
  stl_initialize_facet_check_nearby(stl);

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      facet = stl->facet_start[i];
      for(j = 0; j < 3; j++)
      {
        if(stl->neighbors_start[i].neighbor[j] == -1)
          {
            edge[j].facet_number = i;
            edge[j].which_edge = j;
            if(stl_load_edge_nearby(stl, &edge[j], &facet.vertex[j],
                                  &facet.vertex[(j + 1) % 3],
                                  tolerance))
            {
              /* only insert edges that have different keys */
              insert_hash_edge(stl, edge[j],
stl_match_neighbors_nearby);
            }
          }
      }
    }

  stl_free_edges(stl);
}

static int
stl_load_edge_nearby(stl_file *stl, stl_hash_edge *edge,
                 stl_vertex *a, stl_vertex *b, float tolerance)
{
  float diff_x;
  float diff_y;
  float diff_z;
  float max_diff;
  unsigned vertex1[3];
  unsigned vertex2[3];


  diff_x = ABS(a->x - b->x);
  diff_y = ABS(a->y - b->y);
  diff_z = ABS(a->z - b->z);
  max_diff = STL_MAX(diff_x, diff_y);
  max_diff = STL_MAX(diff_z, max_diff);

  vertex1[0] = (unsigned)((a->x - stl->stats.min.x) / tolerance);
  vertex1[1] = (unsigned)((a->y - stl->stats.min.y) / tolerance);
  vertex1[2] = (unsigned)((a->z - stl->stats.min.z) / tolerance);
  vertex2[0] = (unsigned)((b->x - stl->stats.min.x) / tolerance);
  vertex2[1] = (unsigned)((b->y - stl->stats.min.y) / tolerance);
  vertex2[2] = (unsigned)((b->z - stl->stats.min.z) / tolerance);

  if(   (vertex1[0] == vertex2[0])
     && (vertex1[1] == vertex2[1])
     && (vertex1[2] == vertex2[2]))
```

```
    {
      /* Both vertices hash to the same value */
      return 0;
    }

  if(diff_x == max_diff)
    {
      if(a->x > b->x)
      {
        memcpy(&edge->key[0], vertex1, sizeof(stl_vertex));
        memcpy(&edge->key[3], vertex2, sizeof(stl_vertex));
      }
      else
      {
        memcpy(&edge->key[0], vertex2, sizeof(stl_vertex));
        memcpy(&edge->key[3], vertex1, sizeof(stl_vertex));
        edge->which_edge += 3; /* this edge is loaded backwards */
      }
    }
  else if(diff_y == max_diff)
    {
      if(a->y > b->y)
      {
        memcpy(&edge->key[0], vertex1, sizeof(stl_vertex));
        memcpy(&edge->key[3], vertex2, sizeof(stl_vertex));
      }
      else
      {
        memcpy(&edge->key[0], vertex2, sizeof(stl_vertex));
        memcpy(&edge->key[3], vertex1, sizeof(stl_vertex));
        edge->which_edge += 3; /* this edge is loaded backwards */
      }
    }
  else
    {
      if(a->z > b->z)
      {
        memcpy(&edge->key[0], vertex1, sizeof(stl_vertex));
        memcpy(&edge->key[3], vertex2, sizeof(stl_vertex));
      }
      else
      {
        memcpy(&edge->key[0], vertex2, sizeof(stl_vertex));
        memcpy(&edge->key[3], vertex1, sizeof(stl_vertex));
        edge->which_edge += 3; /* this edge is loaded backwards */
      }
    }
  return 1;
}

static void
stl_free_edges(stl_file *stl)
{
  int i;
```

```
  stl_hash_edge *temp;

  if(stl->stats.malloced != stl->stats.freed)
    {
      for(i = 0; i < stl->M; i++)
      {
        for(temp = stl->heads[i]; stl->heads[i] != stl->tail;
            temp = stl->heads[i])
          {
            stl->heads[i] = stl->heads[i]->next;
            free(temp);
            stl->stats.freed++;
          }
      }
    }
  free(stl->heads);
  free(stl->tail);
}

static void
stl_initialize_facet_check_nearby(stl_file *stl)
{
  int i;

  stl->stats.malloced = 0;
  stl->stats.freed = 0;
  stl->stats.collisions = 0;

  /*  tolerance = STL_MAX(stl->stats.shortest_edge, tolerance);*/
  /*  tolerance = STL_MAX((stl->stats.bounding_diameter / 500000.0),
tolerance);*/
  /*  tolerance *= 0.5;*/

  stl->M = 81397;

  stl->heads = calloc(stl->M, sizeof(*stl->heads));
  if(stl->heads == NULL) perror("stl_initialize_facet_check_nearby");

  stl->tail = malloc(sizeof(stl_hash_edge));
  if(stl->tail == NULL) perror("stl_initialize_facet_check_nearby");

  stl->tail->next = stl->tail;

  for(i = 0; i < stl->M; i++)
    {
      stl->heads[i] = stl->tail;
    }
}



static void
stl_record_neighbors(stl_file *stl,
```

```
                         stl_hash_edge *edge_a, stl_hash_edge
*edge_b)
{
  int i;
  int j;

  /* Facet a's neighbor is facet b */
  stl->neighbors_start[edge_a->facet_number].neighbor[edge_a-
>which_edge % 3] =
    edge_b->facet_number;     /* sets the .neighbor part */

  stl->neighbors_start[edge_a->facet_number].
    which_vertex_not[edge_a->which_edge % 3] =
      (edge_b->which_edge + 2) % 3; /* sets the .which_vertex_not
part */

  /* Facet b's neighbor is facet a */
  stl->neighbors_start[edge_b->facet_number].neighbor[edge_b-
>which_edge % 3] =
    edge_a->facet_number;     /* sets the .neighbor part */

  stl->neighbors_start[edge_b->facet_number].
    which_vertex_not[edge_b->which_edge % 3] =
      (edge_a->which_edge + 2) % 3; /* sets the .which_vertex_not
part */

  if(   ((edge_a->which_edge < 3) && (edge_b->which_edge < 3))
     || ((edge_a->which_edge > 2) && (edge_b->which_edge > 2)))
    {
      /* these facets are oriented in opposite directions.  */
      /*  their normals are probably messed up. */
      stl->neighbors_start[edge_a->facet_number].
      which_vertex_not[edge_a->which_edge % 3] += 3;
      stl->neighbors_start[edge_b->facet_number].
      which_vertex_not[edge_b->which_edge % 3] += 3;
    }


  /* Count successful connects */
  /* Total connects */
  stl->stats.connected_edges += 2;
  /* Count individual connects */
  i = ((stl->neighbors_start[edge_a->facet_number].neighbor[0] == -1)
+
       (stl->neighbors_start[edge_a->facet_number].neighbor[1] == -1)
+
       (stl->neighbors_start[edge_a->facet_number].neighbor[2] == -
1));
  j = ((stl->neighbors_start[edge_b->facet_number].neighbor[0] == -1)
+
       (stl->neighbors_start[edge_b->facet_number].neighbor[1] == -1)
+
       (stl->neighbors_start[edge_b->facet_number].neighbor[2] == -
1));
```

```
  if(i == 2)
    {
      stl->stats.connected_facets_1_edge +=1;
    }
  else if(i == 1)
    {
      stl->stats.connected_facets_2_edge +=1;
    }
  else
    {
      stl->stats.connected_facets_3_edge +=1;
    }
  if(j == 2)
    {
      stl->stats.connected_facets_1_edge +=1;
    }
  else if(j == 1)
    {
      stl->stats.connected_facets_2_edge +=1;
    }
  else
    {
      stl->stats.connected_facets_3_edge +=1;
    }
}

static void
stl_match_neighbors_exact(stl_file *stl,
                         stl_hash_edge *edge_a, stl_hash_edge
*edge_b)
{
  stl_record_neighbors(stl, edge_a, edge_b);
}

static void
stl_match_neighbors_nearby(stl_file *stl,
                         stl_hash_edge *edge_a, stl_hash_edge
*edge_b)
{
  int facet1;
  int facet2;
  int vertex1;
  int vertex2;
  int vnot1;
  int vnot2;
  stl_vertex new_vertex1;
  stl_vertex new_vertex2;

  stl_record_neighbors(stl, edge_a, edge_b);
  stl_which_vertices_to_change(stl, edge_a, edge_b, &facet1,
&vertex1,
                         &facet2, &vertex2, &new_vertex1,
&new_vertex2);
  if(facet1 != -1)
```

```
    {
      if(facet1 == edge_a->facet_number)
      {
        vnot1 = (edge_a->which_edge + 2) % 3;
      }
      else
      {
        vnot1 = (edge_b->which_edge + 2) % 3;
      }
      if(((vnot1 + 2) % 3) == vertex1)
      {
        vnot1 += 3;
      }
      stl_change_vertices(stl, facet1, vnot1, new_vertex1);
    }
  if(facet2 != -1)
    {
      if(facet2 == edge_a->facet_number)
      {
        vnot2 = (edge_a->which_edge + 2) % 3;
      }
      else
      {
        vnot2 = (edge_b->which_edge + 2) % 3;
      }
      if(((vnot2 + 2) % 3) == vertex2)
      {
        vnot2 += 3;
      }
      stl_change_vertices(stl, facet2, vnot2, new_vertex2);
    }
  stl->stats.edges_fixed += 2;
}


static void
stl_change_vertices(stl_file *stl, int facet_num, int vnot,
                   stl_vertex new_vertex)
{
  int first_facet;
  int direction;
  int next_edge;
  int pivot_vertex;

  first_facet = facet_num;
  direction = 0;

  for(;;)
    {
      if(vnot > 2)
      {
        if(direction == 0)
          {
            pivot_vertex = (vnot + 2) % 3;
```

```
            next_edge = pivot_vertex;
            direction = 1;
          }
        else
          {
            pivot_vertex = (vnot + 1) % 3;
            next_edge = vnot % 3;
            direction = 0;
          }
      }
      else
      {
        if(direction == 0)
          {
            pivot_vertex = (vnot + 1) % 3;
            next_edge = vnot;
          }
        else
          {
            pivot_vertex = (vnot + 2) % 3;
            next_edge = pivot_vertex;
          }
      }
      stl->facet_start[facet_num].vertex[pivot_vertex] = new_vertex;
      vnot = stl-
>neighbors_start[facet_num].which_vertex_not[next_edge];
      facet_num = stl-
>neighbors_start[facet_num].neighbor[next_edge];

      if(facet_num == -1)
      {
        break;
      }

      if(facet_num == first_facet)
      {
        /* back to the beginning */
        printf("\
Back to the first facet changing vertices: probably a mobius part.\n\
Try using a smaller tolerance or don't do a nearby check\n");
        exit(1);
        break;
      }
    }
}


static void
stl_which_vertices_to_change(stl_file *stl, stl_hash_edge *edge_a,
                       stl_hash_edge *edge_b, int *facet1, int
*vertex1,
                       int *facet2, int *vertex2,
                       stl_vertex *new_vertex1, stl_vertex
*new_vertex2)
```

```
{
  int v1a;              /* pair 1, facet a */
  int v1b;              /* pair 1, facet b */
  int v2a;              /* pair 2, facet a */
  int v2b;              /* pair 2, facet b */


  /* Find first pair */
  if(edge_a->which_edge < 3)
    {
      v1a = edge_a->which_edge;
      v2a = (edge_a->which_edge + 1) % 3;
    }
  else
    {
      v2a = edge_a->which_edge % 3;
      v1a = (edge_a->which_edge + 1) % 3;
    }
  if(edge_b->which_edge < 3)
    {
      v1b = edge_b->which_edge;
      v2b = (edge_b->which_edge + 1) % 3;
    }
  else
    {
      v2b = edge_b->which_edge % 3;
      v1b = (edge_b->which_edge + 1) % 3;
    }

  /* Of the first pair, which vertex, if any, should be changed */
  if(!memcmp(&stl->facet_start[edge_a->facet_number].vertex[v1a],
          &stl->facet_start[edge_b->facet_number].vertex[v1b],
          sizeof(stl_vertex)))
    {
      /* These facets are already equal.  No need to change. */
      *facet1 = -1;
    }
  else
    {
      if(   (stl->neighbors_start[edge_a->facet_number].neighbor[v1a]
== -1)
       && (stl->neighbors_start[edge_a->facet_number].
        neighbor[(v1a + 2) % 3] == -1))
      {
        /* This vertex has no neighbors.  This is a good one to
change */
        *facet1 = edge_a->facet_number;
        *vertex1 = v1a;
        *new_vertex1 = stl->facet_start[edge_b-
>facet_number].vertex[v1b];
      }
      else
      {
        *facet1 = edge_b->facet_number;
```

```
        *vertex1 = v1b;
        *new_vertex1 = stl->facet_start[edge_a-
>facet_number].vertex[v1a];
      }
    }

  /* Of the second pair, which vertex, if any, should be changed */
  if(!memcmp(&stl->facet_start[edge_a->facet_number].vertex[v2a],
         &stl->facet_start[edge_b->facet_number].vertex[v2b],
         sizeof(stl_vertex)))
    {
      /* These facets are already equal.  No need to change. */
      *facet2 = -1;
    }
  else
    {
      if(   (stl->neighbors_start[edge_a->facet_number].neighbor[v2a]
== -1)
       && (stl->neighbors_start[edge_a->facet_number].
        neighbor[(v2a + 2) % 3] == -1))
      {
        /* This vertex has no neighbors.  This is a good one to
change */
        *facet2 = edge_a->facet_number;
        *vertex2 = v2a;
        *new_vertex2 = stl->facet_start[edge_b-
>facet_number].vertex[v2b];
      }
      else
      {
        *facet2 = edge_b->facet_number;
        *vertex2 = v2b;
        *new_vertex2 = stl->facet_start[edge_a-
>facet_number].vertex[v2a];
      }
    }
}

static void
stl_remove_facet(stl_file *stl, int facet_number)
{
  int neighbor[3];
  int vnot[3];
  int i;
  int j;

  stl->stats.facets_removed += 1;
  /* Update list of connected edges */
  j = ((stl->neighbors_start[facet_number].neighbor[0] == -1) +
       (stl->neighbors_start[facet_number].neighbor[1] == -1) +
       (stl->neighbors_start[facet_number].neighbor[2] == -1));
  if(j == 2)
    {
      stl->stats.connected_facets_1_edge -= 1;
```

```
    }
  else if(j == 1)
    {
      stl->stats.connected_facets_2_edge -= 1;
      stl->stats.connected_facets_1_edge -= 1;
    }
  else if(j == 0)
    {
      stl->stats.connected_facets_3_edge -= 1;
      stl->stats.connected_facets_2_edge -= 1;
      stl->stats.connected_facets_1_edge -= 1;
    }

  stl->facet_start[facet_number] =
    stl->facet_start[stl->stats.number_of_facets - 1];
  /* I could reallocate at this point, but it is not really
necessary. */
  stl->neighbors_start[facet_number] =
    stl->neighbors_start[stl->stats.number_of_facets - 1];
  stl->stats.number_of_facets -= 1;

  for(i = 0; i < 3; i++)
    {
      neighbor[i] = stl->neighbors_start[facet_number].neighbor[i];
      vnot[i] = stl-
>neighbors_start[facet_number].which_vertex_not[i];
    }

  for(i = 0; i < 3; i++)
    {
      if(neighbor[i] != -1)
      {
        if(stl->neighbors_start[neighbor[i]].neighbor[(vnot[i] + 1)%
3] !=
           stl->stats.number_of_facets)
          {
            printf("\
in stl_remove_facet: neighbor = %d numfacets = %d this is wrong\n",
               stl->neighbors_start[neighbor[i]].neighbor[(vnot[i] +
1)% 3],
                  stl->stats.number_of_facets);
            exit(1);
          }
        stl->neighbors_start[neighbor[i]].neighbor[(vnot[i] + 1)% 3]
          = facet_number;
      }
    }
}

void
stl_remove_unconnected_facets(stl_file *stl)
{
  /* A couple of things need to be done here.  One is to remove any
*/
```

```
  /* completely unconnected facets (0 edges connected) since these
are */
  /* useless and could be completely wrong.   The second thing that
needs to */
  /* be done is to remove any degenerate facets that were created
during */
  /* stl_check_facets_nearby(). */

  int i;

  /* remove degenerate facets */
  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      if(   !memcmp(&stl->facet_start[i].vertex[0],
                &stl->facet_start[i].vertex[1], sizeof(stl_vertex))
       || !memcmp(&stl->facet_start[i].vertex[1],
                &stl->facet_start[i].vertex[2], sizeof(stl_vertex))
       || !memcmp(&stl->facet_start[i].vertex[0],
                &stl->facet_start[i].vertex[2], sizeof(stl_vertex)))
      {
        stl_remove_degenerate(stl, i);
        i--;
      }
    }

  if(stl->stats.connected_facets_1_edge < stl-
>stats.number_of_facets)
    {
      /* remove completely unconnected facets */
      for(i = 0; i < stl->stats.number_of_facets; i++)
      {
        if(   (stl->neighbors_start[i].neighbor[0] == -1)
           && (stl->neighbors_start[i].neighbor[1] == -1)
           && (stl->neighbors_start[i].neighbor[2] == -1))
          {
            /* This facet is completely unconnected.  Remove it. */
            stl_remove_facet(stl, i);
            i--;
          }
      }
    }
}

static void
stl_remove_degenerate(stl_file *stl, int facet)
{
  int edge1;
  int edge2;
  int edge3;
  int neighbor1;
  int neighbor2;
  int neighbor3;
  int vnot1;
  int vnot2;
```

```
  int vnot3;

  if(   !memcmp(&stl->facet_start[facet].vertex[0],
            &stl->facet_start[facet].vertex[1], sizeof(stl_vertex))
     && !memcmp(&stl->facet_start[facet].vertex[1],
            &stl->facet_start[facet].vertex[2], sizeof(stl_vertex)))
    {
      /* all 3 vertices are equal.  Just remove the facet.  I don't
think*/
      /* this is really possible, but just in case... */
      printf("removing a facet in stl_remove_degenerate\n");

      stl_remove_facet(stl, facet);
      return;
    }

  if(!memcmp(&stl->facet_start[facet].vertex[0],
           &stl->facet_start[facet].vertex[1], sizeof(stl_vertex)))
    {
      edge1 = 1;
      edge2 = 2;
      edge3 = 0;
    }
  else if(!memcmp(&stl->facet_start[facet].vertex[1],
              &stl->facet_start[facet].vertex[2],
sizeof(stl_vertex)))
    {
      edge1 = 0;
      edge2 = 2;
      edge3 = 1;
    }
  else if(!memcmp(&stl->facet_start[facet].vertex[2],
              &stl->facet_start[facet].vertex[0],
sizeof(stl_vertex)))
    {
      edge1 = 0;
      edge2 = 1;
      edge3 = 2;
    }
  else
    {
      /* No degenerate. Function shouldn't have been called. */
      return;
    }
  neighbor1 = stl->neighbors_start[facet].neighbor[edge1];
  neighbor2 = stl->neighbors_start[facet].neighbor[edge2];

  if(neighbor1 == -1)
    {
      stl_update_connects_remove_1(stl, neighbor2);
    }
  if(neighbor2 == -1)
    {
      stl_update_connects_remove_1(stl, neighbor1);
```

```
    }

  neighbor3 = stl->neighbors_start[facet].neighbor[edge3];
  vnot1 = stl->neighbors_start[facet].which_vertex_not[edge1];
  vnot2 = stl->neighbors_start[facet].which_vertex_not[edge2];
  vnot3 = stl->neighbors_start[facet].which_vertex_not[edge3];

  stl->neighbors_start[neighbor1].neighbor[(vnot1 + 1) % 3] =
neighbor2;
  stl->neighbors_start[neighbor2].neighbor[(vnot2 + 1) % 3] =
neighbor1;
  stl->neighbors_start[neighbor1].which_vertex_not[(vnot1 + 1) % 3] =
vnot2;
  stl->neighbors_start[neighbor2].which_vertex_not[(vnot2 + 1) % 3] =
vnot1;

  stl_remove_facet(stl, facet);

  if(neighbor3 != -1)
    {
      stl_update_connects_remove_1(stl, neighbor3);
      stl->neighbors_start[neighbor3].neighbor[(vnot3 + 1) % 3] = -1;
    }
}

void
stl_update_connects_remove_1(stl_file *stl, int facet_num)
{
  int j;

  /* Update list of connected edges */
  j = ((stl->neighbors_start[facet_num].neighbor[0] == -1) +
       (stl->neighbors_start[facet_num].neighbor[1] == -1) +
       (stl->neighbors_start[facet_num].neighbor[2] == -1));
  if(j == 0)                          /* Facet has 3 neighbors */
    {
      stl->stats.connected_facets_3_edge -= 1;
    }
  else if(j == 1)              /* Facet has 2 neighbors */
    {
      stl->stats.connected_facets_2_edge -= 1;
    }
  else if(j == 2)              /* Facet has 1 neighbor  */
    {
      stl->stats.connected_facets_1_edge -= 1;
    }
}

void
stl_fill_holes(stl_file *stl)
{
  stl_facet facet;
  stl_facet new_facet;
```

```
int neighbors_initial[3];
stl_hash_edge edge;
int first_facet;
int direction;
int facet_num;
int vnot;
int next_edge;
int pivot_vertex;
int next_facet;
int i;
int j;
int k;

/* Insert all unconnected edges into hash list */
stl_initialize_facet_check_nearby(stl);
for(i = 0; i < stl->stats.number_of_facets; i++)
  {
    facet = stl->facet_start[i];
    for(j = 0; j < 3; j++)
    {
      if(stl->neighbors_start[i].neighbor[j] != -1) continue;
      edge.facet_number = i;
      edge.which_edge = j;
      stl_load_edge_exact(stl, &edge, &facet.vertex[j],
                          &facet.vertex[(j + 1) % 3]);

      insert_hash_edge(stl, edge, stl_match_neighbors_exact);
    }
  }

for(i = 0; i < stl->stats.number_of_facets; i++)
  {
    facet = stl->facet_start[i];
    neighbors_initial[0] = stl->neighbors_start[i].neighbor[0];
    neighbors_initial[1] = stl->neighbors_start[i].neighbor[1];
    neighbors_initial[2] = stl->neighbors_start[i].neighbor[2];
    first_facet = i;
    for(j = 0; j < 3; j++)
    {
      if(stl->neighbors_start[i].neighbor[j] != -1) continue;

      new_facet.vertex[0] = facet.vertex[j];
      new_facet.vertex[1] = facet.vertex[(j + 1) % 3];
      if(neighbors_initial[(j + 2) % 3] == -1)
        {
          direction = 1;
        }
      else
        {
          direction = 0;
        }

      facet_num = i;
      vnot = (j + 2) % 3;
```

```
        for(;;)
          {
            if(vnot > 2)
            {
              if(direction == 0)
                {
                  pivot_vertex = (vnot + 2) % 3;
                  next_edge = pivot_vertex;
                  direction = 1;
                }
              else
                {
                  pivot_vertex = (vnot + 1) % 3;
                  next_edge = vnot % 3;
                  direction = 0;
                }
            }
            else
            {
              if(direction == 0)
                {
                  pivot_vertex = (vnot + 1) % 3;
                  next_edge = vnot;
                }
              else
                {
                  pivot_vertex = (vnot + 2) % 3;
                  next_edge = pivot_vertex;
                }
            }
            next_facet = stl-
>neighbors_start[facet_num].neighbor[next_edge];

            if(next_facet == -1)
            {
              new_facet.vertex[2] = stl->facet_start[facet_num].
                vertex[vnot % 3];
              stl_add_facet(stl, &new_facet);
              for(k = 0; k < 3; k++)
                {
                  edge.facet_number = stl->stats.number_of_facets -
1;

                  edge.which_edge = k;
                  stl_load_edge_exact(stl, &edge,
&new_facet.vertex[k],
                                &new_facet.vertex[(k + 1) % 3]);

                  insert_hash_edge(stl, edge,
stl_match_neighbors_exact);
                }
              break;
            }
            else
```

```
            {
              vnot = stl->neighbors_start[facet_num].
                which_vertex_not[next_edge];
              facet_num = next_facet;
            }

            if(facet_num == first_facet)
            {
              /* back to the beginning */
              printf("\
Back to the first facet filling holes: probably a mobius part.\n\
Try using a smaller tolerance or don't do a nearby check\n");
              exit(1);
              break;
            }
          }
      }
    }
}

static void
stl_add_facet(stl_file *stl, stl_facet *new_facet)
{
  stl->stats.facets_added += 1;
  if(stl->stats.facets_malloced < stl->stats.number_of_facets + 1)
    {
      stl->facet_start = realloc(stl->facet_start,
             (sizeof(stl_facet) * (stl->stats.facets_malloced +
256)));
      if(stl->facet_start == NULL) perror("stl_add_facet");
      stl->neighbors_start = realloc(stl->neighbors_start,
             (sizeof(stl_neighbors) * (stl->stats.facets_malloced +
256)));
      if(stl->neighbors_start == NULL) perror("stl_add_facet");
      stl->stats.facets_malloced += 256;
    }
  stl->facet_start[stl->stats.number_of_facets] = *new_facet;

  /* note that the normal vector is not set here, just initialized to
0 */
  stl->facet_start[stl->stats.number_of_facets].normal.x = 0.0;
  stl->facet_start[stl->stats.number_of_facets].normal.y = 0.0;
  stl->facet_start[stl->stats.number_of_facets].normal.z = 0.0;

  stl->neighbors_start[stl->stats.number_of_facets].neighbor[0] = -1;
  stl->neighbors_start[stl->stats.number_of_facets].neighbor[1] = -1;
  stl->neighbors_start[stl->stats.number_of_facets].neighbor[2] = -1;
  stl->stats.number_of_facets += 1;
}
```

normals.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <math.h>

#include "stl.h"

static void stl_reverse_facet(stl_file *stl, int facet_num);
/* static float stl_calculate_area(stl_facet *facet); */
static void stl_reverse_vector(float v[]);
int stl_check_normal_vector(stl_file *stl, int facet_num, int
normal_fix_flag);

static void
stl_reverse_facet(stl_file *stl, int facet_num)
{
  stl_vertex tmp_vertex;
  /*  int tmp_neighbor;*/
  int neighbor[3];
  int vnot[3];

  stl->stats.facets_reversed += 1;

  neighbor[0] = stl->neighbors_start[facet_num].neighbor[0];
  neighbor[1] = stl->neighbors_start[facet_num].neighbor[1];
  neighbor[2] = stl->neighbors_start[facet_num].neighbor[2];
  vnot[0] = stl->neighbors_start[facet_num].which_vertex_not[0];
  vnot[1] = stl->neighbors_start[facet_num].which_vertex_not[1];
  vnot[2] = stl->neighbors_start[facet_num].which_vertex_not[2];

  /* reverse the facet */
  tmp_vertex = stl->facet_start[facet_num].vertex[0];
  stl->facet_start[facet_num].vertex[0] =
    stl->facet_start[facet_num].vertex[1];
  stl->facet_start[facet_num].vertex[1] = tmp_vertex;

  /* fix the vnots of the neighboring facets */
  if(neighbor[0] != -1)
  stl->neighbors_start[neighbor[0]].which_vertex_not[(vnot[0] + 1) %
3] =
    (stl->neighbors_start[neighbor[0]].
     which_vertex_not[(vnot[0] + 1) % 3] + 3) % 6;
  if(neighbor[1] != -1)
  stl->neighbors_start[neighbor[1]].which_vertex_not[(vnot[1] + 1) %
3] =
    (stl->neighbors_start[neighbor[1]].
     which_vertex_not[(vnot[1] + 1) % 3] + 4) % 6;
  if(neighbor[2] != -1)
  stl->neighbors_start[neighbor[2]].which_vertex_not[(vnot[2] + 1) %
3] =
    (stl->neighbors_start[neighbor[2]].
     which_vertex_not[(vnot[2] + 1) % 3] + 2) % 6;

  /* swap the neighbors of the facet that is being reversed */
  stl->neighbors_start[facet_num].neighbor[1] = neighbor[2];
  stl->neighbors_start[facet_num].neighbor[2] = neighbor[1];
```

```
  /* swap the vnots of the facet that is being reversed */
  stl->neighbors_start[facet_num].which_vertex_not[1] = vnot[2];
  stl->neighbors_start[facet_num].which_vertex_not[2] = vnot[1];

  /* reverse the values of the vnots of the facet that is being
reversed */
  stl->neighbors_start[facet_num].which_vertex_not[0] =
    (stl->neighbors_start[facet_num].which_vertex_not[0] + 3) % 6;
  stl->neighbors_start[facet_num].which_vertex_not[1] =
    (stl->neighbors_start[facet_num].which_vertex_not[1] + 3) % 6;
  stl->neighbors_start[facet_num].which_vertex_not[2] =
    (stl->neighbors_start[facet_num].which_vertex_not[2] + 3) % 6;
}

void
stl_fix_normal_directions(stl_file *stl)
{
  char *norm_sw;
  /*  int edge_num;*/
  /*  int vnot;*/
  int checked = 0;
  int facet_num;
  /*  int next_facet;*/
  int i;
  int j;
  int checked_before = 0;
  struct stl_normal
  {
    int                facet_num;
    struct stl_normal *next;
  };
  struct stl_normal *head;
  struct stl_normal *tail;
  struct stl_normal *new;
  struct stl_normal *temp;


  /* Initialize linked list. */
  head = malloc(sizeof(struct stl_normal));
  if(head == NULL) perror("stl_fix_normal_directions");
  tail = malloc(sizeof(struct stl_normal));
  if(tail == NULL) perror("stl_fix_normal_directions");
  head->next = tail;
  tail->next = tail;

  /* Initialize list that keeps track of already fixed facets. */
  norm_sw = calloc(stl->stats.number_of_facets, sizeof(char));
  if(norm_sw == NULL) perror("stl_fix_normal_directions");


  facet_num = 0;
  if(stl_check_normal_vector(stl, 0, 0) == 2)
    stl_reverse_facet(stl, 0);
```

```
  norm_sw[facet_num] = 1;
  /*  edge_num = 0;
      vnot = stl->neighbors_start[0].which_vertex_not[0];
      */
  checked++;

  for(;;)
    {
      /* Add neighbors_to_list. */
      for(j = 0; j < 3; j++)
      {
        /* Reverse the neighboring facets if necessary. */
        if(stl->neighbors_start[facet_num].which_vertex_not[j] > 2)
          {
            if(stl->neighbors_start[facet_num].neighbor[j] != -1)
            {
              stl_reverse_facet
                (stl, stl->neighbors_start[facet_num].neighbor[j]);
            }
          }
        if(stl->neighbors_start[facet_num].neighbor[j] != -1)
          {
            if(norm_sw[stl->neighbors_start[facet_num].neighbor[j]] !
= 1)
            {
              /* Add node to beginning of list. */
              new = malloc(sizeof(struct stl_normal));
              if(new == NULL) perror("stl_fix_normal_directions");
              new->facet_num = stl-
>neighbors_start[facet_num].neighbor[j];
              new->next = head->next;
              head->next = new;
            }
          }
      }
      /* Get next facet to fix from top of list. */
      if(head->next != tail)
      {
        facet_num = head->next->facet_num;
        if(norm_sw[facet_num] != 1) /* If facet is in list mutiple
times */
          {
            norm_sw[facet_num] = 1; /* Record this one as being
fixed. */
            checked++;
          }
        temp = head->next;    /* Delete this facet from the list. */
        head->next = head->next->next;
        free(temp);
      }
      else
      {
        /* All of the facets in this part have been fixed. */
        stl->stats.number_of_parts += 1;
```

```
        /* There are (checked-checked_before) facets */
        /* in part stl->stats.number_of_parts */
        checked_before = checked;
        if(checked == stl->stats.number_of_facets)
          {
            /* All of the facets have been checked.  Bail out. */
            break;
          }
        else
          {
            /* There is another part here.  Find it and continue. */
            for(i = 0; i < stl->stats.number_of_facets; i++)
            {
              if(norm_sw[i] == 0)
                {   /* This is the first facet of the next part. */
                  facet_num = i;
                  if(stl_check_normal_vector(stl, i, 0) == 2)
                  {
                    stl_reverse_facet(stl, i);
                  }

                  norm_sw[facet_num] = 1;
                  checked++;
                  break;
                }
            }
          }
      }
    }
  free(head);
  free(tail);
  free(norm_sw);
}

int
stl_check_normal_vector(stl_file *stl, int facet_num, int
normal_fix_flag)
{
  /* Returns 0 if the normal is within tolerance */
  /* Returns 1 if the normal is not within tolerance, but direction
is OK */
  /* Returns 2 if the normal is not within tolerance and backwards */
  /* Returns 4 if the status is unknown. */

  float normal[3];
  float test_norm[3];
  stl_facet *facet;

  facet = &stl->facet_start[facet_num];

  stl_calculate_normal(normal, facet);
  stl_normalize_vector(normal);

  if(   (ABS(normal[0] - facet->normal.x) < 0.001)
```

```
     && (ABS(normal[1] - facet->normal.y) < 0.001)
     && (ABS(normal[2] - facet->normal.z) < 0.001))
    {
      /* It is not really necessary to change the values here */
      /* but just for consistency, I will. */
      facet->normal.x = normal[0];
      facet->normal.y = normal[1];
      facet->normal.z = normal[2];
      return 0;
    }

  test_norm[0] = facet->normal.x;
  test_norm[1] = facet->normal.y;
  test_norm[2] = facet->normal.z;

  stl_normalize_vector(test_norm);
  if(   (ABS(normal[0] - test_norm[0]) < 0.001)
     && (ABS(normal[1] - test_norm[1]) < 0.001)
     && (ABS(normal[2] - test_norm[2]) < 0.001))
    {
      if(normal_fix_flag)
      {
        facet->normal.x = normal[0];
        facet->normal.y = normal[1];
        facet->normal.z = normal[2];
        stl->stats.normals_fixed += 1;
      }
      return 1;
    }

  stl_reverse_vector(test_norm);
  if(   (ABS(normal[0] - test_norm[0]) < 0.001)
     && (ABS(normal[1] - test_norm[1]) < 0.001)
     && (ABS(normal[2] - test_norm[2]) < 0.001))
    {
      /* Facet is backwards. */
      if(normal_fix_flag)
      {
        facet->normal.x = normal[0];
        facet->normal.y = normal[1];
        facet->normal.z = normal[2];
        stl->stats.normals_fixed += 1;
      }
      return 2;
    }
  if(normal_fix_flag)
    {
      facet->normal.x = normal[0];
      facet->normal.y = normal[1];
      facet->normal.z = normal[2];
      stl->stats.normals_fixed += 1;
    }
  return 4;
}
```

```
static void
stl_reverse_vector(float v[])
{
  v[0] *= -1;
  v[1] *= -1;
  v[2] *= -1;
}


void
stl_calculate_normal(float normal[], stl_facet *facet)
{
  float v1[3];
  float v2[3];

  v1[0] = facet->vertex[1].x - facet->vertex[0].x;
  v1[1] = facet->vertex[1].y - facet->vertex[0].y;
  v1[2] = facet->vertex[1].z - facet->vertex[0].z;
  v2[0] = facet->vertex[2].x - facet->vertex[0].x;
  v2[1] = facet->vertex[2].y - facet->vertex[0].y;
  v2[2] = facet->vertex[2].z - facet->vertex[0].z;

  normal[0] = (float)((double)v1[1] * (double)v2[2]) - ((double)v1[2]
* (double)v2[1]);
  normal[1] = (float)((double)v1[2] * (double)v2[0]) - ((double)v1[0]
* (double)v2[2]);
  normal[2] = (float)((double)v1[0] * (double)v2[1]) - ((double)v1[1]
* (double)v2[0]);
}

/*
static float
stl_calculate_area(stl_facet *facet)
{
  float cross[3][3];
  float sum[3];
  float normal[3];
  float area;
  int i;

  for(i = 0; i < 3; i++)
    {
      cross[i][0] = ((facet->vertex[i].y * facet->vertex[(i + 1) %
3].z) -
                 (facet->vertex[i].z * facet->vertex[(i + 1) %
3].y));
      cross[i][1] = ((facet->vertex[i].z * facet->vertex[(i + 1) %
3].x) -
                 (facet->vertex[i].x * facet->vertex[(i + 1) %
3].z));
      cross[i][2] = ((facet->vertex[i].x * facet->vertex[(i + 1) %
3].y) -
```

```
                 (facet->vertex[i].y * facet->vertex[(i + 1) %
3].x));
    }

  sum[0] = cross[0][0] + cross[1][0] + cross[2][0];
  sum[1] = cross[0][1] + cross[1][1] + cross[2][1];
  sum[2] = cross[0][2] + cross[1][2] + cross[2][2];

  stl_calculate_normal(normal, facet);
  stl_normalize_vector(normal);
  area = 0.5 * (normal[0] * sum[0] + normal[1] * sum[1] +
            normal[2] * sum[2]);
  return ABS(area);
}
*/

void stl_normalize_vector(float v[])
{
  double length;
  double factor;
  float min_normal_length;

  length = sqrt((double)v[0] * (double)v[0] + (double)v[1] *
(double)v[1] + (double)v[2] * (double)v[2]);
  min_normal_length = 0.000000000001;
  if(length < min_normal_length)
    {
      v[0] = 1.0;
      v[1] = 0.0;
      v[2] = 0.0;
      return;
    }
  factor = 1.0 / length;
  v[0] *= factor;
  v[1] *= factor;
  v[2] *= factor;
}

void
stl_fix_normal_values(stl_file *stl)
{
  int i;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      stl_check_normal_vector(stl, i, 1);
    }
}

void
stl_reverse_all_facets(stl_file *stl)
{
  int i;
  float normal[3];
```

```
  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      stl_reverse_facet(stl, i);
      stl_calculate_normal(normal, &stl->facet_start[i]);
      stl_normalize_vector(normal);
      stl->facet_start[i].normal.x = normal[0];
      stl->facet_start[i].normal.y = normal[1];
      stl->facet_start[i].normal.z = normal[2];
    }
}
```

chared.c

```
#include <stdlib.h>

#include "stl.h"


void
stl_generate_shared_vertices(stl_file *stl)
{
  int i;
  int j;
  int first_facet;
  int direction;
  int facet_num;
  int vnot;
  int next_edge;
  int pivot_vertex;
  int next_facet;
  int reversed;

  stl->v_indices =
    calloc(stl->stats.number_of_facets, sizeof(v_indices_struct));
  if(stl->v_indices == NULL) perror("stl_generate_shared_vertices");
  stl->v_shared =
    calloc((stl->stats.number_of_facets / 2), sizeof(stl_vertex));
  if(stl->v_shared == NULL) perror("stl_generate_shared_vertices");
  stl->stats.shared_malloced = stl->stats.number_of_facets / 2;
  stl->stats.shared_vertices = 0;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      stl->v_indices[i].vertex[0] = -1;
      stl->v_indices[i].vertex[1] = -1;
      stl->v_indices[i].vertex[2] = -1;
    }


  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      first_facet = i;
```

```
      for(j = 0; j < 3; j++)
      {
        if(stl->v_indices[i].vertex[j] != -1)
          {
            continue;
          }
        if(stl->stats.shared_vertices == stl->stats.shared_malloced)
          {
            stl->stats.shared_malloced += 1024;
            stl->v_shared = realloc(stl->v_shared,
                     stl->stats.shared_malloced *
sizeof(stl_vertex));
            if(stl->v_shared == NULL)
perror("stl_generate_shared_vertices");
          }

        stl->v_shared[stl->stats.shared_vertices] =
          stl->facet_start[i].vertex[j];

        direction = 0;
        reversed = 0;
        facet_num = i;
        vnot = (j + 2) % 3;

        for(;;)
          {
            if(vnot > 2)
            {
              if(direction == 0)
                {
                  pivot_vertex = (vnot + 2) % 3;
                  next_edge = pivot_vertex;
                  direction = 1;
                }
              else
                {
                  pivot_vertex = (vnot + 1) % 3;
                  next_edge = vnot % 3;
                  direction = 0;
                }
            }
            else
            {
              if(direction == 0)
                {
                  pivot_vertex = (vnot + 1) % 3;
                  next_edge = vnot;
                }
              else
                {
                  pivot_vertex = (vnot + 2) % 3;
                  next_edge = pivot_vertex;
                }
            }
```

```
            stl->v_indices[facet_num].vertex[pivot_vertex] =
            stl->stats.shared_vertices;

            next_facet = stl-
>neighbors_start[facet_num].neighbor[next_edge];
            if(next_facet == -1)
            {
              if(reversed)
                {
                  break;
                }
              else
                {
                  direction = 1;
                  vnot = (j + 1) % 3;
                  reversed = 1;
                  facet_num = first_facet;
                }
            }
            else if(next_facet != first_facet)
            {
              vnot = stl->neighbors_start[facet_num].
                which_vertex_not[next_edge];
              facet_num = next_facet;
            }
            else
            {
              break;
            }
          }
        stl->stats.shared_vertices += 1;
      }
    }
}

void
stl_write_off(stl_file *stl, char *file)
{
  int i;
  FILE      *fp;
  char      *error_msg;


  /* Open the file */
  fp = fopen(file, "w");
  if(fp == NULL)
    {
      error_msg =
      malloc(81 + strlen(file)); /* Allow 80 chars+file size for
message */
      sprintf(error_msg, "stl_write_ascii: Couldn't open %s for
writing",
            file);
      perror(error_msg);
```

```
      free(error_msg);
      exit(1);
    }

  fprintf(fp, "OFF\n");
  fprintf(fp, "%d %d 0\n",
        stl->stats.shared_vertices, stl->stats.number_of_facets);

  for(i = 0; i < stl->stats.shared_vertices; i++)
    {
      fprintf(fp, "\t%f %f %f\n",
            stl->v_shared[i].x, stl->v_shared[i].y, stl-
>v_shared[i].z);
    }
  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      fprintf(fp, "\t3 %d %d %d\n", stl->v_indices[i].vertex[0],
            stl->v_indices[i].vertex[1], stl-
>v_indices[i].vertex[2]);
    }
  fclose(fp);
}

void
stl_write_vrml(stl_file *stl, char *file)
{
  int i;
  FILE      *fp;
  char      *error_msg;


  /* Open the file */
  fp = fopen(file, "w");
  if(fp == NULL)
    {
      error_msg =
      malloc(81 + strlen(file)); /* Allow 80 chars+file size for
message */
      sprintf(error_msg, "stl_write_ascii: Couldn't open %s for
writing",
            file);
      perror(error_msg);
      free(error_msg);
      exit(1);
    }

  fprintf(fp, "#VRML V1.0 ascii\n\n");
  fprintf(fp, "Separator {\n");
  fprintf(fp, "\tDEF STLShape ShapeHints {\n");
  fprintf(fp, "\t\tvertexOrdering COUNTERCLOCKWISE\n");
  fprintf(fp, "\t\tfaceType CONVEX\n");
  fprintf(fp, "\t\tshapeType SOLID\n");
  fprintf(fp, "\t\tcreaseAngle 0.0\n");
  fprintf(fp, "\t}\n");
```

```
  fprintf(fp, "\tDEF STLModel Separator {\n");
  fprintf(fp, "\t\tDEF STLColor Material {\n");
  fprintf(fp, "\t\t\temissiveColor 0.700000 0.700000 0.000000\n");
  fprintf(fp, "\t\t}\n");
  fprintf(fp, "\t\tDEF STLVertices Coordinate3 {\n");
  fprintf(fp, "\t\t\tpoint [\n");

  for(i = 0; i < (stl->stats.shared_vertices - 1); i++)
    {
      fprintf(fp, "\t\t\t\t%f %f %f,\n",
            stl->v_shared[i].x, stl->v_shared[i].y, stl-
>v_shared[i].z);
    }
  fprintf(fp, "\t\t\t\t%f %f %f]\n",
        stl->v_shared[i].x, stl->v_shared[i].y, stl->v_shared[i].z);
  fprintf(fp, "\t\t}\n");
  fprintf(fp, "\t\tDEF STLTriangles IndexedFaceSet {\n");
  fprintf(fp, "\t\t\tcoordIndex [\n");

  for(i = 0; i < (stl->stats.number_of_facets - 1); i++)
    {
      fprintf(fp, "\t\t\t\t%d, %d, %d, -1,\n", stl-
>v_indices[i].vertex[0],
            stl->v_indices[i].vertex[1], stl-
>v_indices[i].vertex[2]);
    }
  fprintf(fp, "\t\t\t\t%d, %d, %d, -1]\n", stl-
>v_indices[i].vertex[0],
        stl->v_indices[i].vertex[1], stl->v_indices[i].vertex[2]);
  fprintf(fp, "\t\t}\n");
  fprintf(fp, "\t}\n");
  fprintf(fp, "}\n");
  fclose(fp);
}
```

stl_io.c

```
#include <stdlib.h>
#include "stl.h"

#if !defined(SEEK_SET)
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
#endif

static void stl_put_little_int(FILE *fp, int value);
static void stl_put_little_float(FILE *fp, float value_in);

void
stl_print_edges(stl_file *stl, FILE *file)
{
  int i;
  int edges_allocated;
```

```
  edges_allocated = stl->stats.number_of_facets * 3;
  for(i = 0; i < edges_allocated; i++)
    {
      fprintf(file, "%d, %f, %f, %f, %f, %f, %f\n",
            stl->edge_start[i].facet_number,
            stl->edge_start[i].p1.x, stl->edge_start[i].p1.y,
            stl->edge_start[i].p1.z, stl->edge_start[i].p2.x,
            stl->edge_start[i].p2.y, stl->edge_start[i].p2.z);
    }
}


void
stl_stats_out(stl_file *stl, FILE *file, char *input_file)
{
  fprintf(file, "\n\
================= Results produced by ADMesh version 0.95
================\n");
  fprintf(file, "\
Input file         : %s\n", input_file);
  if(stl->stats.type == binary)
    {
      fprintf(file, "\
File type          : Binary STL file\n");
    }
  else
    {
      fprintf(file, "\
File type          : ASCII STL file\n");
    }
  fprintf(file, "\
Header             : %s\n", stl->stats.header);
  fprintf(file, "============== Size ==============\n");
  fprintf(file, "Min X = % f, Max X = % f\n",
        stl->stats.min.x, stl->stats.max.x);
  fprintf(file, "Min Y = % f, Max Y = % f\n",
        stl->stats.min.y, stl->stats.max.y);
  fprintf(file, "Min Z = % f, Max Z = % f\n",
        stl->stats.min.z, stl->stats.max.z);

  fprintf(file, "\
========= Facet Status ========== Original ============ Final ====\
n");
  fprintf(file, "\
Number of facets                 : %5d               %5d\n",
        stl->stats.original_num_facets, stl->stats.number_of_facets);
  fprintf(file, "\
Facets with 1 disconnected edge  : %5d               %5d\n",
        stl->stats.facets_w_1_bad_edge, stl-
>stats.connected_facets_2_edge -
        stl->stats.connected_facets_3_edge);
  fprintf(file, "\
Facets with 2 disconnected edges : %5d               %5d\n",
```

```
        stl->stats.facets_w_2_bad_edge, stl-
>stats.connected_facets_1_edge -
        stl->stats.connected_facets_2_edge);
  fprintf(file, "\
Facets with 3 disconnected edges : %5d               %5d\n",
        stl->stats.facets_w_3_bad_edge, stl->stats.number_of_facets -
        stl->stats.connected_facets_1_edge);
  fprintf(file, "\
Total disconnected facets        : %5d               %5d\n",
        stl->stats.facets_w_1_bad_edge + stl-
>stats.facets_w_2_bad_edge +
        stl->stats.facets_w_3_bad_edge, stl->stats.number_of_facets -
        stl->stats.connected_facets_3_edge);

  fprintf(file,
"=== Processing Statistics ===     ===== Other Statistics =====\n");
  fprintf(file, "\
Number of parts       : %5d        Volume   : % f\n",
        stl->stats.number_of_parts, stl->stats.volume);
  fprintf(file, "\
Degenerate facets     : %5d\n", stl->stats.degenerate_facets);
  fprintf(file, "\
Edges fixed           : %5d\n", stl->stats.edges_fixed);
  fprintf(file, "\
Facets removed        : %5d\n", stl->stats.facets_removed);
  fprintf(file, "\
Facets added          : %5d\n", stl->stats.facets_added);
  fprintf(file, "\
Facets reversed       : %5d\n", stl->stats.facets_reversed);
  fprintf(file, "\
Backwards edges       : %5d\n", stl->stats.backwards_edges);
  fprintf(file, "\
Normals fixed         : %5d\n", stl->stats.normals_fixed);
}

void
stl_write_ascii(stl_file *stl, char *file, char *label)
{
  int       i;
  FILE      *fp;
  char      *error_msg;


  /* Open the file */
  fp = fopen(file, "w");
  if(fp == NULL)
    {
      error_msg =
      malloc(81 + strlen(file)); /* Allow 80 chars+file size for
message */
      sprintf(error_msg, "stl_write_ascii: Couldn't open %s for
writing",
            file);
      perror(error_msg);
```

```
      free(error_msg);
      exit(1);
    }

  fprintf(fp, "solid  %s\n", label);

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      fprintf(fp, "  facet normal % .8E % .8E % .8E\n",
            stl->facet_start[i].normal.x, stl-
>facet_start[i].normal.y,
            stl->facet_start[i].normal.z);
      fprintf(fp, "    outer loop\n");
      fprintf(fp, "      vertex % .8E % .8E % .8E\n",
            stl->facet_start[i].vertex[0].x, stl-
>facet_start[i].vertex[0].y,
            stl->facet_start[i].vertex[0].z);
      fprintf(fp, "      vertex % .8E % .8E % .8E\n",
            stl->facet_start[i].vertex[1].x, stl-
>facet_start[i].vertex[1].y,
            stl->facet_start[i].vertex[1].z);
      fprintf(fp, "      vertex % .8E % .8E % .8E\n",
            stl->facet_start[i].vertex[2].x, stl-
>facet_start[i].vertex[2].y,
            stl->facet_start[i].vertex[2].z);
      fprintf(fp, "    endloop\n");
      fprintf(fp, "  endfacet\n");
    }

  fprintf(fp, "endsolid  %s\n", label);

  fclose(fp);
}

void
stl_print_neighbors(stl_file *stl, char *file)
{
  int i;
  FILE *fp;
  char *error_msg;

  /* Open the file */
  fp = fopen(file, "w");
  if(fp == NULL)
    {
      error_msg =
      malloc(81 + strlen(file)); /* Allow 80 chars+file size for
message */
      sprintf(error_msg, "stl_print_neighbors: Couldn't open %s for
writing",
            file);
      perror(error_msg);
      free(error_msg);
      exit(1);
```

```
    }

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      fprintf(fp, "%d, %d,%d, %d,%d, %d,%d\n",
            i,
            stl->neighbors_start[i].neighbor[0],
            (int)stl->neighbors_start[i].which_vertex_not[0],
            stl->neighbors_start[i].neighbor[1],
            (int)stl->neighbors_start[i].which_vertex_not[1],
            stl->neighbors_start[i].neighbor[2],
            (int)stl->neighbors_start[i].which_vertex_not[2]);
    }
}

static void
stl_put_little_int(FILE *fp, int value_in)
{
  int new_value;
  union
    {
      int  int_value;
      char char_value[4];
    } value;

  value.int_value = value_in;

  new_value  = value.char_value[0] & 0xFF;
  new_value |= (value.char_value[1] & 0xFF) << 0x08;
  new_value |= (value.char_value[2] & 0xFF) << 0x10;
  new_value |= (value.char_value[3] & 0xFF) << 0x18;
  fwrite(&new_value, sizeof(int), 1, fp);
}

static void
stl_put_little_float(FILE *fp, float value_in)
{
  int new_value;
  union
    {
      float float_value;
      char  char_value[4];
    } value;

  value.float_value = value_in;

  new_value  = value.char_value[0] & 0xFF;
  new_value |= (value.char_value[1] & 0xFF) << 0x08;
  new_value |= (value.char_value[2] & 0xFF) << 0x10;
  new_value |= (value.char_value[3] & 0xFF) << 0x18;
  fwrite(&new_value, sizeof(int), 1, fp);
}
```

```
void
stl_write_binary(stl_file *stl, char *file, char *label)
{
  FILE      *fp;
  int       i;
  char      *error_msg;


  /* Open the file */
  fp = fopen(file, "w");
  if(fp == NULL)
    {
      error_msg =
      malloc(81 + strlen(file)); /* Allow 80 chars+file size for
message */
      sprintf(error_msg, "stl_write_binary: Couldn't open %s for
writing",
            file);
      perror(error_msg);
      free(error_msg);
      exit(1);
    }

  fprintf(fp, "%s", label);
  for(i = strlen(label); i < LABEL_SIZE; i++) putc(0, fp);

  fseek(fp, LABEL_SIZE, SEEK_SET);

  stl_put_little_int(fp, stl->stats.number_of_facets);

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      stl_put_little_float(fp, stl->facet_start[i].normal.x);
      stl_put_little_float(fp, stl->facet_start[i].normal.y);
      stl_put_little_float(fp, stl->facet_start[i].normal.z);
      stl_put_little_float(fp, stl->facet_start[i].vertex[0].x);
      stl_put_little_float(fp, stl->facet_start[i].vertex[0].y);
      stl_put_little_float(fp, stl->facet_start[i].vertex[0].z);
      stl_put_little_float(fp, stl->facet_start[i].vertex[1].x);
      stl_put_little_float(fp, stl->facet_start[i].vertex[1].y);
      stl_put_little_float(fp, stl->facet_start[i].vertex[1].z);
      stl_put_little_float(fp, stl->facet_start[i].vertex[2].x);
      stl_put_little_float(fp, stl->facet_start[i].vertex[2].y);
      stl_put_little_float(fp, stl->facet_start[i].vertex[2].z);
      fputc(stl->facet_start[i].extra[0], fp);
      fputc(stl->facet_start[i].extra[1], fp);
    }

  fclose(fp);
}

void
stl_write_vertex(stl_file *stl, int facet, int vertex)
{
```

```
  printf("  vertex %d/%d % .8E % .8E % .8E\n", vertex, facet,
       stl->facet_start[facet].vertex[vertex].x,
       stl->facet_start[facet].vertex[vertex].y,
       stl->facet_start[facet].vertex[vertex].z);
}

void
stl_write_facet(stl_file *stl, char *label, int facet)
{
  printf("facet (%d)/ %s\n", facet, label);
  stl_write_vertex(stl, facet, 0);
  stl_write_vertex(stl, facet, 1);
  stl_write_vertex(stl, facet, 2);
}

void
stl_write_edge(stl_file *stl, char *label, stl_hash_edge edge)
{
  printf("edge (%d)/(%d) %s\n", edge.facet_number, edge.which_edge,
label);
  if(edge.which_edge < 3)
    {
      stl_write_vertex(stl, edge.facet_number, edge.which_edge % 3);
      stl_write_vertex(stl, edge.facet_number, (edge.which_edge + 1)
% 3);
    }
  else
    {
      stl_write_vertex(stl, edge.facet_number, (edge.which_edge + 1)
% 3);
      stl_write_vertex(stl, edge.facet_number, edge.which_edge % 3);
    }
}

void
stl_write_neighbor(stl_file *stl, int facet)
{
  printf("Neighbors %d: %d, %d, %d ;  %d, %d, %d\n", facet,
       stl->neighbors_start[facet].neighbor[0],
       stl->neighbors_start[facet].neighbor[1],
       stl->neighbors_start[facet].neighbor[2],
       stl->neighbors_start[facet].which_vertex_not[0],
       stl->neighbors_start[facet].which_vertex_not[1],
       stl->neighbors_start[facet].which_vertex_not[2]);
}

void
stl_write_quad_object(stl_file *stl, char *file)
{
  FILE      *fp;
  int       i;
  int       j;
  char      *error_msg;
  stl_vertex connect_color;
```

```
  stl_vertex uncon_1_color;
  stl_vertex uncon_2_color;
  stl_vertex uncon_3_color;
  stl_vertex color;

  /* Open the file */
  fp = fopen(file, "w");
  if(fp == NULL)
    {
      error_msg =
      malloc(81 + strlen(file)); /* Allow 80 chars+file size for
message */
      sprintf(error_msg, "stl_write_quad_object: Couldn't open %s for
writing",
            file);
      perror(error_msg);
      free(error_msg);
      exit(1);
    }

  connect_color.x = 0.0;
  connect_color.y = 0.0;
  connect_color.z = 1.0;
  uncon_1_color.x = 0.0;
  uncon_1_color.y = 1.0;
  uncon_1_color.z = 0.0;
  uncon_2_color.x = 1.0;
  uncon_2_color.y = 1.0;
  uncon_2_color.z = 1.0;
  uncon_3_color.x = 1.0;
  uncon_3_color.y = 0.0;
  uncon_3_color.z = 0.0;

  fprintf(fp, "CQUAD\n");
  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      j = ((stl->neighbors_start[i].neighbor[0] == -1) +
         (stl->neighbors_start[i].neighbor[1] == -1) +
         (stl->neighbors_start[i].neighbor[2] == -1));
      if(j == 0)
      {
        color = connect_color;
      }
      else if(j == 1)
      {
        color = uncon_1_color;
      }
      else if(j == 2)
      {
        color = uncon_2_color;
      }
      else
      {
        color = uncon_3_color;
```

```
      }
      fprintf(fp, "%f %f %f    %1.1f %1.1f %1.1f 1\n",
            stl->facet_start[i].vertex[0].x,
            stl->facet_start[i].vertex[0].y,
            stl->facet_start[i].vertex[0].z, color.x, color.y,
color.z);
      fprintf(fp, "%f %f %f    %1.1f %1.1f %1.1f 1\n",
            stl->facet_start[i].vertex[1].x,
            stl->facet_start[i].vertex[1].y,
            stl->facet_start[i].vertex[1].z, color.x, color.y,
color.z);
      fprintf(fp, "%f %f %f    %1.1f %1.1f %1.1f 1\n",
            stl->facet_start[i].vertex[2].x,
            stl->facet_start[i].vertex[2].y,
            stl->facet_start[i].vertex[2].z, color.x, color.y,
color.z);
      fprintf(fp, "%f %f %f    %1.1f %1.1f %1.1f 1\n",
            stl->facet_start[i].vertex[2].x,
            stl->facet_start[i].vertex[2].y,
            stl->facet_start[i].vertex[2].z, color.x, color.y,
color.z);
    }
  fclose(fp);
}

void
stl_write_dxf(stl_file *stl, char *file, char *label)
{
  int       i;
  FILE      *fp;
  char      *error_msg;

  /* Open the file */
  fp = fopen(file, "w");
  if(fp == NULL)
    {
      error_msg =
      malloc(81 + strlen(file)); /* Allow 80 chars+file size for
message */
      sprintf(error_msg, "stl_write_ascii: Couldn't open %s for
writing",
            file);
      perror(error_msg);
      free(error_msg);
      exit(1);
    }
  fprintf(fp, "999\n%s\n", label);
  fprintf(fp, "0\nSECTION\n2\nHEADER\n0\nENDSEC\n");
  fprintf(fp, "0\nSECTION\n2\nTABLES\n0\nTABLE\n2\nLAYER\n70\n1\n\
0\nLAYER\n2\n0\n70\n0\n62\n7\n6\nCONTINUOUS\n0\nENDTAB\n0\nENDSEC\
n");
  fprintf(fp, "0\nSECTION\n2\nBLOCKS\n0\nENDSEC\n");
  fprintf(fp, "0\nSECTION\n2\nENTITIES\n");
  for(i = 0; i < stl->stats.number_of_facets; i++)
```

```
    {
      fprintf(fp, "0\n3DFACE\n8\n0\n");
      fprintf(fp, "10\n%f\n20\n%f\n30\n%f\n",
            stl->facet_start[i].vertex[0].x, stl-
>facet_start[i].vertex[0].y,
            stl->facet_start[i].vertex[0].z);
      fprintf(fp, "11\n%f\n21\n%f\n31\n%f\n",
            stl->facet_start[i].vertex[1].x, stl-
>facet_start[i].vertex[1].y,
            stl->facet_start[i].vertex[1].z);
      fprintf(fp, "12\n%f\n22\n%f\n32\n%f\n",
            stl->facet_start[i].vertex[2].x, stl-
>facet_start[i].vertex[2].y,
            stl->facet_start[i].vertex[2].z);
      fprintf(fp, "13\n%f\n23\n%f\n33\n%f\n",
            stl->facet_start[i].vertex[2].x, stl-
>facet_start[i].vertex[2].y,
            stl->facet_start[i].vertex[2].z);
    }
  fprintf(fp, "0\nENDSEC\n0\nEOF\n");
  fclose(fp);
}
```

stlinit.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "stl.h"

#if !defined(SEEK_SET)
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
#endif

static void stl_initialize(stl_file *stl, char *file);
static void stl_allocate(stl_file *stl);
static void stl_read(stl_file *stl, int first_facet, int first);
static void stl_reallocate(stl_file *stl);
static int stl_get_little_int(FILE *fp);
static float stl_get_little_float(FILE *fp);

void
stl_open(stl_file *stl, char *file)
{
  stl_initialize(stl, file);
  stl_allocate(stl);
  stl_read(stl, 0, 1);
  fclose(stl->fp);
}

static int
```

```
stl_get_little_int(FILE *fp)
{
  int value;
  value  =  fgetc(fp) & 0xFF;
  value |= (fgetc(fp) & 0xFF) << 0x08;
  value |= (fgetc(fp) & 0xFF) << 0x10;
  value |= (fgetc(fp) & 0xFF) << 0x18;
  return(value);
}

static float
stl_get_little_float(FILE *fp)
{
  union
    {
      int   int_value;
      float float_value;
    } value;

  value.int_value  =  fgetc(fp) & 0xFF;
  value.int_value |= (fgetc(fp) & 0xFF) << 0x08;
  value.int_value |= (fgetc(fp) & 0xFF) << 0x10;
  value.int_value |= (fgetc(fp) & 0xFF) << 0x18;
  return(value.float_value);
}


static void
stl_initialize(stl_file *stl, char *file)
{
  long           file_size;
  int            header_num_facets;
  int            num_facets;
  int            i, j;
  unsigned char  chtest[128];
  int            num_lines = 1;
  char           *error_msg;

  stl->stats.degenerate_facets = 0;
  stl->stats.edges_fixed  = 0;
  stl->stats.facets_added = 0;
  stl->stats.facets_removed = 0;
  stl->stats.facets_reversed = 0;
  stl->stats.normals_fixed = 0;
  stl->stats.number_of_parts = 0;
  stl->stats.original_num_facets = 0;
  stl->stats.number_of_facets = 0;
  stl->stats.volume = -1.0;

  stl->neighbors_start = NULL;
  stl->facet_start = NULL;
  stl->v_indices = NULL;
  stl->v_shared = NULL;
```

```
  /* Open the file */
  stl->fp = fopen(file, "r");
  if(stl->fp == NULL)
    {
      error_msg = 
      malloc(81 + strlen(file)); /* Allow 80 chars+file size for 
message */
      sprintf(error_msg, "stl_initialize: Couldn't open %s for 
reading",
            file);
      perror(error_msg);
      free(error_msg);
      exit(1);
    }
  /* Find size of file */
  fseek(stl->fp, 0, SEEK_END);
  file_size = ftell(stl->fp);

  /* Check for binary or ASCII file */
  fseek(stl->fp, HEADER_SIZE, SEEK_SET);
  fread(chtest, sizeof(chtest), 1, stl->fp);
  stl->stats.type = ascii;
  for(i = 0; i < sizeof(chtest); i++)
    {
      if(chtest[i] > 127)
      {
        stl->stats.type = binary;
        break;
      }
    }
  rewind(stl->fp);

  /* Get the header and the number of facets in the .STL file */
  /* If the .STL file is binary, then do the following */
  if(stl->stats.type == binary)
    {
      /* Test if the STL file has the right size  */
      if(((file_size - HEADER_SIZE) % SIZEOF_STL_FACET != 0)
       || (file_size < STL_MIN_FILE_SIZE))
      {
        fprintf(stderr, "The file %s has the wrong size.\n", file);
        exit(1);
      }
      num_facets = (file_size - HEADER_SIZE) / SIZEOF_STL_FACET;

      /* Read the header */
      fread(stl->stats.header, LABEL_SIZE, 1, stl->fp);
      stl->stats.header[80] = '\0';

      /* Read the int following the header.  This should contain # of 
facets */
      header_num_facets = stl_get_little_int(stl->fp);
      if(num_facets != header_num_facets)
```

```
      {
        fprintf(stderr,
        "Warning: File size doesn't match number of facets in the
header\n");
      }
    }
  /* Otherwise, if the .STL file is ASCII, then do the following */
  else
    {
      /* Find the number of facets */
      j = 0;
      for(i = 0; i < file_size ; i++)
      {
        j++;
        if(getc(stl->fp) == '\n')
          {
            if(j > 4) /* don't count short lines */
            {
              num_lines++;
            }
            j = 0;
          }
      }
      rewind(stl->fp);

      /* Get the header */
      for(i = 0;
        (i < 80) && (stl->stats.header[i] = getc(stl->fp)) != '\n';
i++);
      stl->stats.header[i] = '\0'; /* Lose the '\n' */
      stl->stats.header[80] = '\0';

      num_facets = num_lines / ASCII_LINES_PER_FACET;
    }
  stl->stats.number_of_facets += num_facets;
  stl->stats.original_num_facets = stl->stats.number_of_facets;
}

static void
stl_allocate(stl_file *stl)
{
  /*  Allocate memory for the entire .STL file */
  stl->facet_start = calloc(stl->stats.number_of_facets,
                      sizeof(stl_facet));
  if(stl->facet_start == NULL) perror("stl_initialize");
  stl->stats.facets_malloced = stl->stats.number_of_facets;

  /* Allocate memory for the neighbors list */
  stl->neighbors_start =
    calloc(stl->stats.number_of_facets, sizeof(stl_neighbors));
  if(stl->facet_start == NULL) perror("stl_initialize");
}

void
```

```
stl_open_merge(stl_file *stl, char *file)
{
  int first_facet;

  first_facet = stl->stats.number_of_facets;
  stl_initialize(stl, file);
  stl_reallocate(stl);
  stl_read(stl, first_facet, 0);
}

static void
stl_reallocate(stl_file *stl)
{
  /*  Reallocate more memory for the .STL file(s) */
  stl->facet_start = realloc(stl->facet_start, stl-
>stats.number_of_facets *
                       sizeof(stl_facet));
  if(stl->facet_start == NULL) perror("stl_initialize");
  stl->stats.facets_malloced = stl->stats.number_of_facets;

  /* Reallocate more memory for the neighbors list */
  stl->neighbors_start =
    realloc(stl->neighbors_start, stl->stats.number_of_facets *
          sizeof(stl_neighbors));
  if(stl->facet_start == NULL) perror("stl_initialize");
}

static void
stl_read(stl_file *stl, int first_facet, int first)
{
  stl_facet facet;
  int   i;
  float diff_x;
  float diff_y;
  float diff_z;
  float max_diff;


  if(stl->stats.type == binary)
    {
      fseek(stl->fp, HEADER_SIZE, SEEK_SET);
    }
  else
    {
      rewind(stl->fp);
      /* Skip the first line of the file */
      while(getc(stl->fp) != '\n');
    }

  for(i = first_facet; i < stl->stats.number_of_facets; i++)
    {
      if(stl->stats.type == binary)
      /* Read a single facet from a binary .STL file */
      {
```

```
    facet.normal.x = stl_get_little_float(stl->fp);
    facet.normal.y = stl_get_little_float(stl->fp);
    facet.normal.z = stl_get_little_float(stl->fp);
    facet.vertex[0].x = stl_get_little_float(stl->fp);
    facet.vertex[0].y = stl_get_little_float(stl->fp);
    facet.vertex[0].z = stl_get_little_float(stl->fp);
    facet.vertex[1].x = stl_get_little_float(stl->fp);
    facet.vertex[1].y = stl_get_little_float(stl->fp);
    facet.vertex[1].z = stl_get_little_float(stl->fp);
    facet.vertex[2].x = stl_get_little_float(stl->fp);
    facet.vertex[2].y = stl_get_little_float(stl->fp);
    facet.vertex[2].z = stl_get_little_float(stl->fp);
    facet.extra[0] = fgetc(stl->fp);
    facet.extra[1] = fgetc(stl->fp);
  }
  else
  /* Read a single facet from an ASCII .STL file */
  {
          fscanf(stl->fp, "%*s %*s %f %f %f\n", &facet.normal.x,
         &facet.normal.y, &facet.normal.z);
    fscanf(stl->fp, "%*s %*s");
    fscanf(stl->fp, "%*s %f %f %f\n", &facet.vertex[0].x,
         &facet.vertex[0].y,  &facet.vertex[0].z);
    fscanf(stl->fp, "%*s %f %f %f\n", &facet.vertex[1].x,
         &facet.vertex[1].y,  &facet.vertex[1].z);
    fscanf(stl->fp, "%*s %f %f %f\n", &facet.vertex[2].x,
         &facet.vertex[2].y,  &facet.vertex[2].z);
    fscanf(stl->fp, "%*s");
    fscanf(stl->fp, "%*s");
  }
  /* Write the facet into memory. */
  stl->facet_start[i] = facet;

  /* while we are going through all of the facets, let's find the
*/
  /* maximum and minimum values for x, y, and z  */

  /* Initialize the max and min values the first time through*/
  if(first)
  {
    stl->stats.max.x = facet.vertex[0].x;
    stl->stats.min.x = facet.vertex[0].x;
    stl->stats.max.y = facet.vertex[0].y;
    stl->stats.min.y = facet.vertex[0].y;
    stl->stats.max.z = facet.vertex[0].z;
    stl->stats.min.z = facet.vertex[0].z;

    diff_x = ABS(facet.vertex[0].x - facet.vertex[1].x);
    diff_y = ABS(facet.vertex[0].y - facet.vertex[1].y);
    diff_z = ABS(facet.vertex[0].z - facet.vertex[1].z);
    max_diff = STL_MAX(diff_x, diff_y);
    max_diff = STL_MAX(diff_z, max_diff);
    stl->stats.shortest_edge = max_diff;
```

```
        first = 0;
      }
      /* now find the max and min values */
      stl->stats.max.x = STL_MAX(stl->stats.max.x,
facet.vertex[0].x);
      stl->stats.min.x = STL_MIN(stl->stats.min.x,
facet.vertex[0].x);
      stl->stats.max.y = STL_MAX(stl->stats.max.y,
facet.vertex[0].y);
      stl->stats.min.y = STL_MIN(stl->stats.min.y,
facet.vertex[0].y);
      stl->stats.max.z = STL_MAX(stl->stats.max.z,
facet.vertex[0].z);
      stl->stats.min.z = STL_MIN(stl->stats.min.z,
facet.vertex[0].z);

      stl->stats.max.x = STL_MAX(stl->stats.max.x,
facet.vertex[1].x);
      stl->stats.min.x = STL_MIN(stl->stats.min.x,
facet.vertex[1].x);
      stl->stats.max.y = STL_MAX(stl->stats.max.y,
facet.vertex[1].y);
      stl->stats.min.y = STL_MIN(stl->stats.min.y,
facet.vertex[1].y);
      stl->stats.max.z = STL_MAX(stl->stats.max.z,
facet.vertex[1].z);
      stl->stats.min.z = STL_MIN(stl->stats.min.z,
facet.vertex[1].z);

      stl->stats.max.x = STL_MAX(stl->stats.max.x,
facet.vertex[2].x);
      stl->stats.min.x = STL_MIN(stl->stats.min.x,
facet.vertex[2].x);
      stl->stats.max.y = STL_MAX(stl->stats.max.y,
facet.vertex[2].y);
      stl->stats.min.y = STL_MIN(stl->stats.min.y,
facet.vertex[2].y);
      stl->stats.max.z = STL_MAX(stl->stats.max.z,
facet.vertex[2].z);
      stl->stats.min.z = STL_MIN(stl->stats.min.z,
facet.vertex[2].z);
    }
  stl->stats.size.x = stl->stats.max.x - stl->stats.min.x;
  stl->stats.size.y = stl->stats.max.y - stl->stats.min.y;
  stl->stats.size.z = stl->stats.max.z - stl->stats.min.z;
  stl->stats.bounding_diameter =
    sqrt(stl->stats.size.x * stl->stats.size.x +
       stl->stats.size.y * stl->stats.size.y +
       stl->stats.size.z * stl->stats.size.z);
}


void
stl_close(stl_file *stl)
```

```
{
    if(stl->neighbors_start != NULL)
      free(stl->neighbors_start);
    if(stl->facet_start != NULL)
      free(stl->facet_start);
    if(stl->v_indices != NULL)
      free(stl->v_indices);
    if(stl->v_shared != NULL)
      free(stl->v_shared);
}
```

util.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "stl.h"

#define M_PI 3.14159265359
static void stl_rotate(float *x, float *y, float angle);
static void stl_get_size(stl_file *stl);
static float get_area(stl_facet *facet);
static float get_volume(stl_file *stl);


void
stl_verify_neighbors(stl_file *stl)
{
  int i;
  int j;
  stl_edge edge_a;
  stl_edge edge_b;
  int neighbor;
  int vnot;

  stl->stats.backwards_edges = 0;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        edge_a.p1 = stl->facet_start[i].vertex[j];
        edge_a.p2 = stl->facet_start[i].vertex[(j + 1) % 3];
        neighbor = stl->neighbors_start[i].neighbor[j];
        vnot = stl->neighbors_start[i].which_vertex_not[j];

        if(neighbor == -1)
          continue;           /* this edge has no neighbor...
Continue. */
        if(vnot < 3)
          {
```

```
            edge_b.p1 = stl->facet_start[neighbor].vertex[(vnot + 2)
% 3];
            edge_b.p2 = stl->facet_start[neighbor].vertex[(vnot + 1)
% 3];
          }
        else
          {
            stl->stats.backwards_edges += 1;
            edge_b.p1 = stl->facet_start[neighbor].vertex[(vnot + 1)
% 3];
            edge_b.p2 = stl->facet_start[neighbor].vertex[(vnot + 2)
% 3];
          }
        if(memcmp(&edge_a, &edge_b, SIZEOF_EDGE_SORT) != 0)
          {
            /* These edges should match but they don't.  Print
results. */
            printf("edge %d of facet %d doesn't match edge %d of
facet %d\n",
                   j, i, vnot + 1, neighbor);
            stl_write_facet(stl, "first facet", i);
            stl_write_facet(stl, "second facet", neighbor);
          }
      }
    }
}

void
stl_translate(stl_file *stl, float x, float y, float z)
{
  int i;
  int j;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        stl->facet_start[i].vertex[j].x -= (stl->stats.min.x - x);
        stl->facet_start[i].vertex[j].y -= (stl->stats.min.y - y);
        stl->facet_start[i].vertex[j].z -= (stl->stats.min.z - z);
      }
    }
  stl->stats.max.x -= (stl->stats.min.x - x);
  stl->stats.max.y -= (stl->stats.min.y - y);
  stl->stats.max.z -= (stl->stats.min.z - z);
  stl->stats.min.x = x;
  stl->stats.min.y = y;
  stl->stats.min.z = z;
}

void
stl_scale(stl_file *stl, float factor)
{
  int i;
```

```
  int j;

  stl->stats.min.x *= factor;
  stl->stats.min.y *= factor;
  stl->stats.min.z *= factor;
  stl->stats.max.x *= factor;
  stl->stats.max.y *= factor;
  stl->stats.max.z *= factor;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        stl->facet_start[i].vertex[j].x *= factor;
        stl->facet_start[i].vertex[j].y *= factor;
        stl->facet_start[i].vertex[j].z *= factor;
      }
    }
}

static void calculate_normals(stl_file *stl)
{
      long i;
      float normal[3];

      for(i = 0; i < stl->stats.number_of_facets; i++){
            stl_calculate_normal(normal, &stl->facet_start[i]);
            stl_normalize_vector(normal);
            stl->facet_start[i].normal.x = normal[0];
            stl->facet_start[i].normal.y = normal[1];
            stl->facet_start[i].normal.z = normal[2];
      }
}

void
stl_rotate_x(stl_file *stl, float angle)
{
  int i;
  int j;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        stl_rotate(&stl->facet_start[i].vertex[j].y,
                   &stl->facet_start[i].vertex[j].z, angle);
      }
    }
  stl_get_size(stl);
      calculate_normals(stl);
}

void
stl_rotate_y(stl_file *stl, float angle)
```

```
{
  int i;
  int j;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        stl_rotate(&stl->facet_start[i].vertex[j].z,
                   &stl->facet_start[i].vertex[j].x, angle);
      }
    }
  stl_get_size(stl);
      calculate_normals(stl);
}

void
stl_rotate_z(stl_file *stl, float angle)
{
  int i;
  int j;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        stl_rotate(&stl->facet_start[i].vertex[j].x,
                   &stl->facet_start[i].vertex[j].y, angle);
      }
    }
  stl_get_size(stl);
      calculate_normals(stl);
}



static void
stl_rotate(float *x, float *y, float angle)
{
  double r;
  double theta;
  double radian_angle;

  radian_angle = (angle / 180.0) * M_PI;

  r = sqrt((*x * *x) + (*y * *y));
  theta = atan2(*y, *x);
  *x = r * cos(theta + radian_angle);
  *y = r * sin(theta + radian_angle);
}

static void
stl_get_size(stl_file *stl)
{
```

```
  int i;
  int j;

  stl->stats.min.x = stl->facet_start[0].vertex[0].x;
  stl->stats.min.y = stl->facet_start[0].vertex[0].y;
  stl->stats.min.z = stl->facet_start[0].vertex[0].z;
  stl->stats.max.x = stl->facet_start[0].vertex[0].x;
  stl->stats.max.y = stl->facet_start[0].vertex[0].y;
  stl->stats.max.z = stl->facet_start[0].vertex[0].z;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        stl->stats.min.x = STL_MIN(stl->stats.min.x,
                                   stl->facet_start[i].vertex[j].x);
        stl->stats.min.y = STL_MIN(stl->stats.min.y,
                                   stl->facet_start[i].vertex[j].y);
        stl->stats.min.z = STL_MIN(stl->stats.min.z,
                                   stl->facet_start[i].vertex[j].z);
        stl->stats.max.x = STL_MAX(stl->stats.max.x,
                                   stl->facet_start[i].vertex[j].x);
        stl->stats.max.y = STL_MAX(stl->stats.max.y,
                                   stl->facet_start[i].vertex[j].y);
        stl->stats.max.z = STL_MAX(stl->stats.max.z,
                                   stl->facet_start[i].vertex[j].z);
      }
    }
}

void
stl_mirror_xy(stl_file *stl)
{
  int i;
  int j;
  float temp_size;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        stl->facet_start[i].vertex[j].z *= -1.0;
      }
    }
  temp_size = stl->stats.min.z;
  stl->stats.min.z = stl->stats.max.z;
  stl->stats.max.z = temp_size;
  stl->stats.min.z *= -1.0;
  stl->stats.max.z *= -1.0;
}

void
stl_mirror_yz(stl_file *stl)
{
```

```
  int i;
  int j;
  float temp_size;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        stl->facet_start[i].vertex[j].x *= -1.0;
      }
    }
  temp_size = stl->stats.min.x;
  stl->stats.min.x = stl->stats.max.x;
  stl->stats.max.x = temp_size;
  stl->stats.min.x *= -1.0;
  stl->stats.max.x *= -1.0;
}

void
stl_mirror_xz(stl_file *stl)
{
  int i;
  int j;
  float temp_size;

  for(i = 0; i < stl->stats.number_of_facets; i++)
    {
      for(j = 0; j < 3; j++)
      {
        stl->facet_start[i].vertex[j].y *= -1.0;
      }
    }
  temp_size = stl->stats.min.y;
  stl->stats.min.y = stl->stats.max.y;
  stl->stats.max.y = temp_size;
  stl->stats.min.y *= -1.0;
  stl->stats.max.y *= -1.0;
}

static float get_volume(stl_file *stl)
{
      long i;
      stl_vertex p0;
      stl_vertex p;
      stl_normal n;
      float height;
      float area;
      float volume = 0.0;

      /* Choose a point, any point as the reference */
      p0.x = stl->facet_start[0].vertex[0].x;
      p0.y = stl->facet_start[0].vertex[0].y;
      p0.z = stl->facet_start[0].vertex[0].z;
```

```
      for(i = 0; i < stl->stats.number_of_facets; i++){
            p.x = stl->facet_start[i].vertex[0].x - p0.x;
            p.y = stl->facet_start[i].vertex[0].y - p0.y;
            p.z = stl->facet_start[i].vertex[0].z - p0.z;
            /* Do dot product to get distance from point to plane */
            n = stl->facet_start[i].normal;
            height = (n.x * p.x) + (n.y * p.y) + (n.z * p.z);
            area = get_area(&stl->facet_start[i]);
            volume += (area * height) / 3.0;
      }
      return volume;
}

void stl_calculate_volume(stl_file *stl)
{
      stl->stats.volume = get_volume(stl);
      if(stl->stats.volume < 0.0){
            stl_reverse_all_facets(stl);
            stl->stats.volume = -stl->stats.volume;
      }
}

static float get_area(stl_facet *facet)
{
      float cross[3][3];
      float sum[3];
      float n[3];
      float area;
      int i;

      for(i = 0; i < 3; i++){
          cross[i][0]=((facet->vertex[i].y * facet->vertex[(i + 1) %
3].z) -
                   (facet->vertex[i].z * facet->vertex[(i + 1) %
3].y));
          cross[i][1]=((facet->vertex[i].z * facet->vertex[(i + 1) %
3].x) -
                   (facet->vertex[i].x * facet->vertex[(i + 1) %
3].z));
          cross[i][2]=((facet->vertex[i].x * facet->vertex[(i + 1) %
3].y) -
                   (facet->vertex[i].y * facet->vertex[(i + 1) %
3].x));
      }

      sum[0] = cross[0][0] + cross[1][0] + cross[2][0];
      sum[1] = cross[0][1] + cross[1][1] + cross[2][1];
      sum[2] = cross[0][2] + cross[1][2] + cross[2][2];
      /* This should already be done.  But just in case, let's do it
again */
      stl_calculate_normal(n, facet);
      stl_normalize_vector(n);

      area = 0.5 * (n[0] * sum[0] + n[1] * sum[1] + n[2] * sum[2]);
```

```
    return area;
}
```

Prepared by Anthony D. Martin

APPENDIX B

STL PROCESSING RESULTS

Output Summary For cubetol.stl

```
ADMesh version 0.95, Copyright (C) 1995, 1996 Anthony D. Martin
ADMesh comes with NO WARRANTY.  This is free software, and you are
welcome to
redistribute it under certain conditions.  See the file COPYING for
details.
Opening cubetol.stl
Checking exact...
Checking nearby. Tolerance= 0.800000 Iteration=1 of 2...  Fixed 10
edges.
All facets connected.  No further nearby check necessary.
No unconnected need to be removed.
No holes need to be filled.
Checking normal directions...
Checking normal values...
Calculating volume...
Verifying neighbors...
Writing ascii file cubetolfix.stl

================= Results produced by ADMesh version 0.95
================
Input file         : cubetol.stl
File type          : ASCII STL file
Header             : solid cubetol.stl
============== Size ==============
Min X =  0.000000, Max X =  1.000000
Min Y =  0.000000, Max Y =  1.000000
Min Z =  0.000000, Max Z =  1.000000
========= Facet Status ========== Original ============ Final ====
Number of facets                 :    12                  12
Facets with 1 disconnected edge  :     0                   0
Facets with 2 disconnected edges :     5                   0
Facets with 3 disconnected edges :     0                   0
Total disconnected facets        :     5                   0
=== Processing Statistics ===     ===== Other Statistics =====
Number of parts       :     1        Volume   :  1.000000
Degenerate facets     :     0
Edges fixed           :    10
Facets removed        :     0
Facets added          :     0
Facets reversed       :     0
Backwards edges       :     0
Normals fixed         :     0
```

cubetol.stl

```
solid cubetol.stl
  facet normal  0.00000000E+000  0.00000000E+000  1.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000 -1.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000 -1.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal -1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal -1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.80000000E+000  0.90000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.90000000E+000  0.80000000E+000
    endloop
```

```
  endfacet
  facet normal  0.00000000E+000 -1.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000 -1.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  1.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.80000000E+000  1.00000000E+000  0.90000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  1.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.90000000E+000  1.00000000E+000  0.80000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000  1.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
    endloop
  endfacet
endsolid  cubetol.stl
```

cubetolfix.stl

```
solid  Processed by ADMesh version 0.95
  facet normal  0.00000000E+000  0.00000000E+000  1.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000 -1.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000 -1.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal -1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal -1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
    endloop
```

```
  endfacet
  facet normal  0.00000000E+000 -1.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000 -1.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  1.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  1.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000  1.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
    endloop
  endfacet
endsolid  Processed by ADMesh version 0.95
```

Output Summary For cubehole.stl

```
ADMesh version 0.95, Copyright (C) 1995, 1996 Anthony D. Martin
ADMesh comes with NO WARRANTY.  This is free software, and you are
welcome to
redistribute it under certain conditions.  See the file COPYING for
details.
Opening cubehole.stl
Checking exact...
Checking nearby. Tolerance= 1.000000 Iteration=1 of 2...  Fixed 0
edges.
Checking nearby. Tolerance= 1.000173 Iteration=2 of 2...  Fixed 0
edges.
Removing unconnected facets...
Filling holes...
Checking normal directions...
Checking normal values...
Calculating volume...
Verifying neighbors...
Writing ascii file cubeholefix.stl

================= Results produced by ADMesh version 0.95
================
Input file         : cubehole.stl
File type          : ASCII STL file
Header             : solid cubehole.stl
============== Size ==============
Min X =  0.000000, Max X =  1.000000
Min Y =  0.000000, Max Y =  1.000000
Min Z =  0.000000, Max Z =  1.000000
========= Facet Status ========== Original ============ Final ====
Number of facets                 :    10                 12
Facets with 1 disconnected edge  :     4                  0
Facets with 2 disconnected edges :     0                  0
Facets with 3 disconnected edges :     0                  0
Total disconnected facets        :     4                  0
=== Processing Statistics ===     ===== Other Statistics =====
Number of parts       :     1        Volume   :  1.000000
Degenerate facets     :     0
Edges fixed           :     0
Facets removed        :     0
Facets added          :     2
Facets reversed       :     2
Backwards edges       :     0
Normals fixed         :     2
```

cubehole.stl

```
solid cubehole.stl
  facet normal  0.00000000E+000  0.00000000E+000 -1.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000 -1.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal -1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal -1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000 -1.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
```

```
  endfacet
  facet normal  0.00000000E+000 -1.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  1.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  1.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
endsolid cubehole.stl
```

cubeholefix.stl

```
solid  Processed by ADMesh version 0.95
  facet normal  0.00000000E+000  0.00000000E+000 -1.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000 -1.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal -1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal -1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  1.00000000E+000  0.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000 -1.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
    endloop
```

```
  endfacet
  facet normal  0.00000000E+000 -1.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  0.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  1.00000000E+000  0.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  1.00000000E+000  0.00000000E+000
    outer loop
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  0.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  0.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000  1.00000000E+000
    outer loop
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
    endloop
  endfacet
  facet normal  0.00000000E+000  0.00000000E+000  1.00000000E+000
    outer loop
      vertex  1.00000000E+000  0.00000000E+000  1.00000000E+000
      vertex  1.00000000E+000  1.00000000E+000  1.00000000E+000
      vertex  0.00000000E+000  1.00000000E+000  1.00000000E+000
    endloop
  endfacet
endsolid  Processed by ADMesh version 0.95
```

REFERENCES

REFERENCES

Bell, G., Parisi, A., and Pesce, M.,1996, "VRML 1.0 Specification," Silicon Graphics, Mountain View, CA.

Glassner, A. S., 1990, "Graphics Gems," Academic Press, Cambridge, MA, pp. 539-547.

Phillips, M., Levy, S., and Munzer, T., 1996, "Geomview Manual," Geometry Center, Minneapolis, MN.

Sedgewick, R., 1990, "Algorithms in C," Addison-Wesley Publishing Company, Reading, MA, pp. 232-236.

CONTENTS