Mapping a 3-D Model into Abstract Cellular Complex Format

Varakorn Ungvichian[1] and Pizzanu Kanongchaiyos[2]

[1]Chulalongkorn University, ungvichian@thaimail.com
[2] Chulalongkorn University, pizzanu@cp.eng.chula.ac.th

ABSTRACT

This paper describes an algorithm for converting a three-dimensional object in wireframe format into the Abstract Cellular Complex format described by Kovalevsky, a format designed to efficiently represent topological data in 3-D images. The algorithm is partially based on previously-developed algorithms which convert a two-dimensional wireframe drawing into a three-dimensional object, adapted here for processing three-dimensional wireframes to find their faces. Other steps of the algorithm determine the volumes which are bounded by the faces, and the relations between each edge and its adjacent entities. With further refinements, this work will be useful to work that utilizes computer graphics and would benefit from topological data, such as computer-assisted design (CAD).

Keywords: Topology, data representation, cellular complexes

1. INTRODUCTION

Presently, computers are used in many aspects of everyday life, and computers in general have been improved, both in efficiency and capability. Among the various abilities that have been developed for computers is computer graphics, which has developed from two-dimensional to three-dimensional, allowing for realistic display of three-dimensional objects.

Using CG in three dimensions results in a new problem: There are many ways to represent a three-dimensional object in CG, for example: wireframe, solid model, surface model, and winged-edge. Different representations are not interchangeable, due to the differences between the data retained in each representation. Also, in a given line of work, such as product design, more than one representation may be needed. Therefore, a method to convert between representations is required.

Among the many representations of three-dimensional CG that have been developed is Kovalevsky's "Abstract Cellular Complex" format, which was described as a structure with the property of efficient representation of topological data. This is the structure that will be the output of this work.

The expected benefits of this study are 1) To learn about finding topological data from wireframe information, and 2) To provide a method for converting wireframe information into Abstract Cellular Complex format.

2. PREVIOUS LITERATURE

2.1 The Abstract Cellular Complex

Kovalevsky [1] explained the requirements for a data structure to efficiently represent topological data thusly:

- The data structure must contain sufficiently complete topological data to get knowledge about topological relations among parts of a 3-D scene without searching.
- The data structure must represent "non-proper" complexes correctly.

Kovalevsky proposed a data structure that satisfies these requirements in two dimensions, the "cell list", and also proposed a cell list structure for three dimensions. The three-dimensional cell list is comprised of lists of vertices, edges, faces and volumes, along with the relations between various elements, for example, the edges that enter/exit each vertex.

In this data structure, the topological information of a 3-D scene can be found without a search, and the structure also features other properties as described by Kovalevsky.

The only notable flaw in Kovalevsky's structure is that in order to allow for finding topological information in a 3-D scene without searching, the structure contains some redundancy. For example, the vertices come with information on where edges enter and exit, and the edges come with information about the start and end points. Thus, the relations between edges and vertices are featured at least twice in the structure. However, for our purposes, this flaw will not be considered.

2.2 Converting Wireframe Data into a 3-D Image

Shpiltani and Lipson's research [4] was among the first to describe a method to convert a two-dimensional wireframe drawing into a three-dimensional object. This method has since been improved by two pieces of subsequent research:

1. Liu and Lee [2]: Liu and Lee devised a "depth first search" algorithm for finding minimal faces, as well as an algorithm for finding the optimal arrangement of faces based on algorithms for finding the maximum weight clique in a graph. It was proved that the results of this new algorithm are identical to those obtained from Shpiltani and Lipson's algorithm.
2. Oh and Kim [3]: Oh and Kim reduced the time required to find the two-dimensional faces by organizing the circuits found into three different classes: "basis faces", "minimal faces", and "implausible faces", with the basis faces absolutely certain of being the object's true faces. Also, "sketch order analysis" was also used in reconstructing the object.

2.3 3D Shape Recognition

Much research on 3D shape recognition has focused on re-constructing 3D shapes from 2D data, such as projections [5] or images from cameras [6], [7]. This research focuses on obtaining a 3D shape from given 3D data.

Hofer et al. [8] describe a method for detecting the surface of a 3D shape from given 3D data, based on line geometry, while Mello et al. [9] describe a general method to determine the in/out function of a surface represented by a sample of points. However, those research works cover different scopes from the present research. Hofer's research deals with surface-like point clouds and triangle meshes and is only interested in finding the surface of an object, while Mello's research deals with finding a surface that best fits a given sample of points. The present research deals with a given set of faces (described by their vertices and edges), and is interested in finding the manifolds within a 3D shape (although finding the surface of the shape is part of the algorithm). The method can also be generalized to multiple objects in a single image as well.

2.4 Topology Extraction

Steiner and Fischer [10] describe a method of extracting the topological graph from a wireframe mesh. However, the mesh used in this research is represented with a boundary representation, with all relations between facets, edges and point elements already provided. This current research extracts topological data from a simple wireframe with only its edges identified.

3. PURPOSE AND LIMITATIONS OF THE STUDY

3.1 Purpose

The purpose of this program is to find a method to convert a 3-D image in wireframe format into Abstract Cellular Complex format. The main difference between this research and the research cited in the previous section is that the previous research describes methods to convert *2-D* images of wireframes into regular 3-D images.

3.2 Limitations

The program will directly convert a wireframe into Abstract Cellular Complex format, with all the edges in the wireframe being straight edges. Also, it is assumed that all objects in the given file have no holes, and that the objects do not overlap each other (each object will be considered individually).

4. THE ALGORITHM

The algorithm to convert from wireframe into Abstract Cellular Format is as follows:

- Read the information on each edge and find each individual vertex
- Determine the faces from the edges and vertices given, using a breadth-first search to trace out faces
- Determining the volumes from the faces found
- Finding the relations between each edge and its adjacent volumes and faces
- Assembling the data together

The language used for the implementation is Microsoft Visual Basic .NET. The steps of the implementation and details of each step are described below.

5. STEPS

5.1 Reading in information on each vertex from the wireframe information

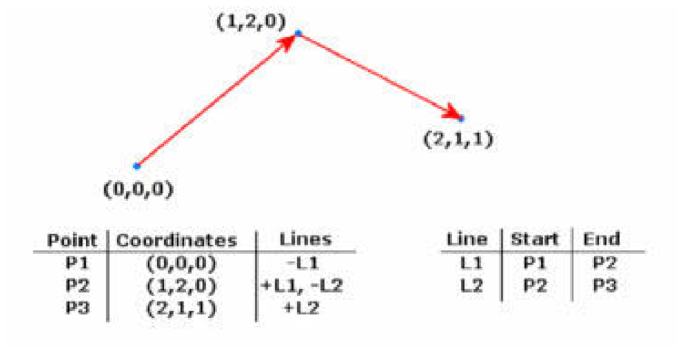
The format of the input is a file with the edges in the format (x_0, y_0, z_0, x_1, y_1, z_1), with each edge starting from the coordinates marked by x_0, y_0, z_0 (therefore setting the direction).

While reading in each edge from the file, the program compares each read vertex with previously read vertices, which are stored in a hash table with linear probing, and stores the vertices if needed. The hash table is expanded when it is full.

After all the edges have been read, the program sorts the hash table by the x, y and z coordinates (respectively) of each vertex. For example, (0, 0, 0) comes before (0, 0, 1) (0, 1, 0) and (1, 0, 0).

5.2 Reading in information on each edge from the wireframe information

After sorting the table of vertices, the data on the edges is re-read, using a binary search (as the vertices have been sorted) to find the vertices in the table. When found, edge and direction information will be stored in an array corresponding to each vertex, and the points found for each edge will be stored in another array.



Point	Coordinates	Lines
P1	(0,0,0)	-L1
P2	(1,2,0)	+L1, -L2
P3	(2,1,1)	+L2

Line	Start	End
L1	P1	P2
L2	P2	P3

Fig. 1. A sample result obtained from the first two steps.

The information on the vertices and information on which edges are adjacent to each vertex are stored in separate arrays in the actual implementation, with the data in the two arrays corresponding to each other.

5.3 Finding Faces in the Wireframe

After obtaining information on each edge, the next step is to figure out the faces of the object, by tracing along each edge.

Tracing along each edge begins by checking how many edges extend from either end. The trace will begin from the direction with the lesser amount of edges (when equal, the trace will start from the edge's normal direction), so as to minimize the time needed to trace the edges.

The program traces by creating a path consisting of the first edge and the chosen direction and placing it in an array. At each step of a trace, an intermediate array is created to store each currently traced path. During each step, the program reads the last edge and direction of each path in the array to determine the last vertex of the current path. The path is then appended with the other edges attached to the last vertex, and then stored in the new array. Creating this intermediate array is comparable to running a breadth first search. See Fig. 2.

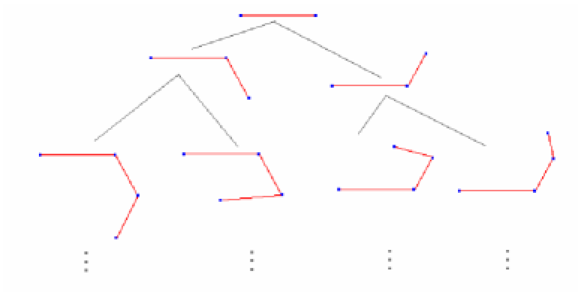


Fig. 2. Face tracing.

After each step of creating the intermediate array, the array's size is reduced, to reduce the time required for tracing. Paths removed from the array fulfill one of these properties (see Fig. 3):

- Has a circuit *within* the path (as opposed to the whole path being a circuit), for example, a path that runs through vertices v_1, v_6, v_3, v_5, and v_6 respectively.
- Its first two edges are the same as a path that is a circuit. For example, if a path of comprising of edges e_1, e_2, e_3, and e_4 is a circuit, a path comprising of edges e_1, e_2, e_5, and e_6 will be removed. This corresponds to Shpiltani and Lipson's face adjacency theorem [4].

Also, if a path in the array is a circuit, no edges will be added to the end. For example, if a path in the current array comprising of edges e_1, e_2, e_3, and e_4 is a circuit, there will not be a path comprising of edges e_1, e_2, e_3, e_4, and e_5 in the next array, but rather just e_1, e_2, e_3, and e_4.

This intermediate array will then be used to form subsequent arrays, so as to trace various paths, until the new array is identical to the previous. At this point, the array is comprised entirely of circuits, and then output to a file. This is done for each edge, with the overall result being the first stage of potential faces.

The potential faces this step obtains are not all the faces of the shape in some cases (namely, when there are many edges in one of the actual faces, such as in the case of a cylinder), and this will be rectified in a later step.

5.4 Circuit Organization

Two traces starting from different edges may produce identical loops. Therefore, after the initial list of potential faces has been obtained, the list will be reduced to unique paths, by using a hash table to store each found path.

After each path is read in, a hash value is calculated from the edges in the path, and then used to search for the path in the table. Paths are compared first by looking at their lengths. If equal, the program searches for the edge used for the "start" of the first path in the other path. If that edge is found, both paths are traversed according to the appropriate direction. If no different edges are found, the paths are equivalent.

As is the case with the hash table in the vertices reading step, if the table is full, a rehash is executed. After all the circuits are read, only the distinct potential faces from the previous step remain.

5.5 Reducing the Initial Potential Faces Down to the Actual Faces

In finding the actual faces from the initial batch of potential faces, one of the steps used is finding the area of each face, by rotating each face by using well-defined matrix transforms so that:

- the first two edges of each path lie on the x-z plane (that is, $y = 0$),
- the path starts at the origin point (0, 0, 0), and

- the first edge of the path runs along the x-axis

After transforming the path, lines are "drawn" from the origin point (the first vertex in the path) to other vertices in the path. If none of the lines drawn are found to intersect any edges (considering the x-y plane only) or pass outside of the circuit, the face's area will be measured by splitting it into several triangles, each with the origin point as a vertex, finding the area using known formulas, and adding them together.

If there is at least one line that intersects an edge, the face is initially split into 2 or 3 parts (some parts may need to be split even further), and their areas are then calculated and added up. (In some cases, the face will be rotated by transformation to make it possible to find the area. In testing, a few cases have been found where it is not possible for the program to calculate the face area, in which cases these faces are discarded.)

Another step in finding the most likely faces is checking the flatness of the faces, by calculating the normal vector from the cross product ($\vec{a} \times \vec{b}$) of the two edges that meet at each vertex, and then calculating the cosine of the angle between each of these normal vectors by using the dot product (as $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}|\cos\theta$, where θ is the angle between the two vectors). The smallest absolute cosine between any pair of normal vectors is used as the flatness value, which thus takes a range between 0 and 1, with 1 being a flat face.

Also, the program searches for common edges between various faces. If two faces have a common edge, there may be a face comprised of these two faces (or even these two faces plus another) in the list. In the case that such a face is found, that face is then removed and no longer considered, so as to speed up the process (this step of the algorithm is the most time-consuming). This step is equivalent to removing the faces with internal edges in Oh and Kim's study [3]. Faces that share at least two edges with another smaller face (or a smaller sum of two faces) are designated *secondary faces*, while the other faces are designated *primary faces*.

Other methods in this step include finding smooth entity chains (due to the limitations of this study, the program is interested only in series of edges which comprise a single straight line), calculating the maximum rank of each edge and vertex using Shpitalni and Lipson's equations (and considering the smooth entity chains found), and recording which faces are adjacent to each edge and vertex, for use in the next step.

5.6 Arranging the Initially Found Faces into an Initial Object

In selecting various faces from the ones previously determined, the first faces selected are the smallest primary faces, with a flatness value of more than 0.9, adjacent to each vertex and edge. If no such face exists for a given edge, the smallest secondary face with a flatness value of more than 0.9 will be selected instead (again, if such a face exists).

After the selection is done, the ranks of each edge and vertex are calculated and compared with the ranks previously calculated. If any part of the picture has a rank higher than the calculated rank, the largest faces (with any of the parts with excess rank) are unselected until no excess ranks remain.

After removing excess faces, the remaining flat and unselected primary faces are then selected face-by-face, with ranks being checked after each face. If after adding a face, there are excess ranks, that face is unselected. If the added face shares more than one edge with an already selected face, the larger face is unselected.

In most tests, the expected actual faces (among the initial potential faces) are the exact faces selected. Exceptional cases are when the expected faces are those that do not have at least 0.9 flatness (however, it is expected that in practice, such faces will not be created), and certain 2-D images, as this algorithm is mainly designed for 3-D (for example, it treats a flat square with edges between its center and each of its corners the same as a pyramid, producing four triangles and one square in both cases). The latter case will be handled in the next step.

5.7 Finding Undiscovered Faces

As already mentioned earlier, there are some cases where not all faces have been found. This step is used to account for the unfound faces.

In the winged-edge structure, each edge has two faces adjacent to it. With this in mind, the algorithm searches for all edges with actual rank of 0 or 1, except for faces with both calculated and actual rank of 1 each. These edges are used to form possible extra faces, by using the same tracing method used to find the initial faces.

The faces found are then analyzed and added where applicable, by considering edge and vertex ranks and whether each face shares more than one edge with a previously selected face.

Also, overlapping faces are cut by finding two faces that produce a flat face when added together, finding a face bigger than these two faces and sharing more than one edge with the sum, and comparing the sum's center with the boundaries of the larger face. If the center of the sum is found to be inside the larger face, the larger face is considered to be overlapping, and therefore discarded.

5.8 Creating the Finished Object and Identifying Manifolds

After the final faces are obtained for each object, the objects are then traced out, starting at a random edge, adding faces adjacent to the edge, tracing the edges in each face, and then adding faces adjacent to those edges recursively until no more faces or edges are added.

Each object may consist of several manifolds (for instance, two cubes that are attached so that a face is shared between the cubes trace out as one object). Manifolds are traced in a different manner. First, after making copies of the lists of edges, faces and points that make up the object to work from, the program starts finding manifolds by putting an "available" (i.e., has not been removed from the list copy) face with the smallest (x, y, then z) coordinates of its first two vertices in the face list, and adding its edges to the edge list. Faces that are adjacent to edges that are adjacent to two faces (which are available to be added) are repeatedly added, along with the faces' edges and vertices, to their respective lists until no such faces remain to be added.

If the currently selected faces do not correspond to the Euler-Poincare theorem (in its most simplified form, $V - E + F = 2$, where V is the number of vertices, E the number of edges, and F the number of faces), and there are faces remaining to be added, the program looks for edges which only have one selected face attached. When such an edge is found, it checks how many available faces are adjacent to the edge. (The count also includes the already-selected face.) If there are just two faces, it will simply add the not-yet selected face (along with its edges and vertices). However, if there are at least three faces, the program calculates vectors which lie on the same plane as each face (i.e., perpendicular to the face's normal), and are perpendicular with the edge in question. The program then traces edges with only one selected face attached, starting from the current edge in question, to create a chain of edges (see Fig. 3 for an example), before finding the average of the vertices in the chain. The program calculates the vector between the center of the current edge and the average of the vertices, and modifies the vector to be perpendicular to the edge while lying on the same plane as the original vector (if needed). This is used to determine the proper turn direction, and thus the face with the smallest turn angle in the proper direction, which is then added.

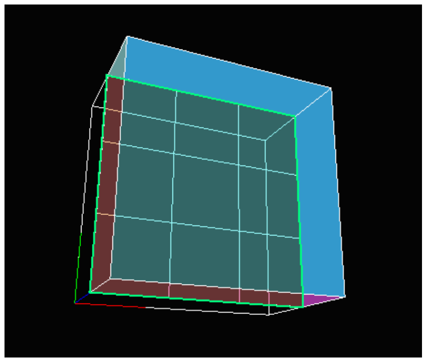


Fig. 3. Example, with the edge chain marked in a thick line, and the selected faces shaded.

As an example, consider Fig. 4 (left). Here, 4 faces (represented by the edges marked *a*, *b*, *c*, and *d*) are adjacent to an edge (represented with vertex *A*). Face *a* has already been selected. The program will either select face *b* or *d*, depending on the vector between the center of edge *A* and the average of the vertices in a chain that starts with the same edge. Possible results are represented here with α and β. Using α as the result, the program finds that the face with the least turn angle (in the direction of α) from *a* is *d*, and thus picks that face. However, using β as the result, the program will select *b* instead, since it is the face with the least turn angle in the same direction as β. Using the sample shape, Fig. 4 (right) illustrates the selection method (using many of the same labels).

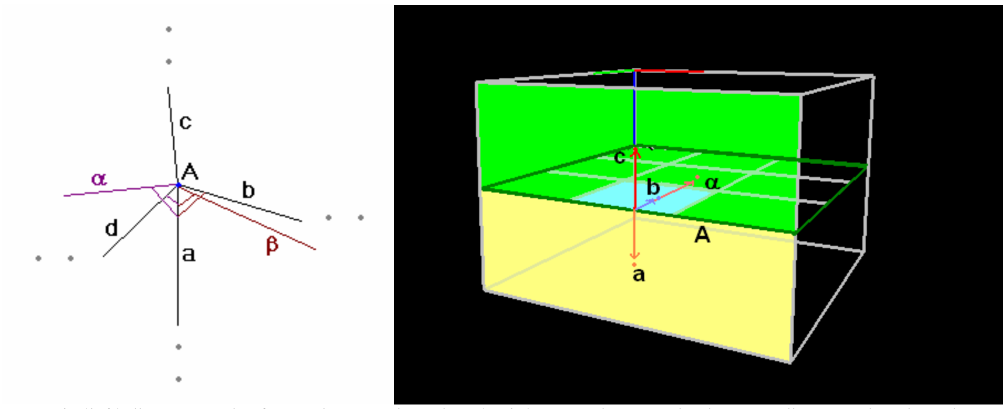


Fig. 4. *(left)* Illustrating the face selecting algorithm *(right)* Using the sample shape to illustrate the algorithm.

This process is then repeated until the faces correspond to the Euler-Poincare theorem, thus comprising a closed volume, or until there are no more faces remaining in the object, in which case the selected faces represent an open (i.e., not closed) volume.

In order to determine which faces to remove from future consideration, the outside surface of the whole object has to be traced. Tracing the surface uses the same algorithm as that used to found the manifolds, except that faces with the *highest* turn angle are added (using Fig. 4 as an example, α results in the program selecting *b*, and β results in the program selecting *d*). The list of faces in the manifold is compared with the list of faces in on the surface. The list of faces in the manifold which are not on the outside surface of the object is classified as the split plane.

A potential flaw in both the manifold and surface finding algorithms is that there is the possibility that the program cannot determine the proper turn direction. Currently, the program solves this issue by trying different edges instead, and if the proper turn direction cannot be obtained from any edge (which is most possible when the selected faces are part of a single plane), the program modifies the vector which held the result of the difference between the center of the edge and center of the edge chain, by adding to the x (and y, if necessary) coordinate values of that difference. The use of this ad hoc solution is due to how both finding algorithms start at faces with the smallest coordinates.
The faces that are not in the split plane list (i.e., are on the object's outside surface) are removed from the copy of the list of faces in the object. The process then repeats from the face selection onwards, until the copy of the face list is empty, and by this point, the program has identified all manifolds (representing both closed and open volumes) in the image.

5.9 Identifying Face-volume Relationships

As part of converting the object into Abstract Cellular Complex format, the relationships between the faces and their respective volumes need to be found.

The first step is to look for the vertex with the smallest coordinates (least x, then least y, then least z) in the current volume being considered. After finding the vertex, the program then picks the edge with that vertex (that is part of the volume) that makes the least angle with the z-axis. After selecting the edge, the program then picks from the faces adjacent to that edge the face where the cross product between the edge and its normal vector has the least angle with the y-axis. The program checks the x-coordinate of the picked face's normal vector. If the x-coordinate is positive, the face is facing towards the volume, and if the x-coordinate is negative, the face is facing away from the volume. Depending on the result, the vector that points away from the volume is stored in an array (either the normal vector or its opposite).

The program finds an unconsidered face in the volume that is attached to a previously considered face. The vector between the center of the currently considered face and the center of the previously considered face is calculated. The program calculates the dot product between the vector that points away from the volume from the previous face (*a* in Fig. 5) and the vector between the centers of the two faces (*c*), and then the dot product between the currently considered face's normal vector (*b*) and the vector between the centers. If both products have the same sign, the currently considered face is facing towards the volume, while if the two products have opposite signs, the new face is facing away from the volume. (In the case of the first dot product being 0, the dot product of the two normal vectors is considered instead.) This process repeats itself until each face in the volume has been considered.

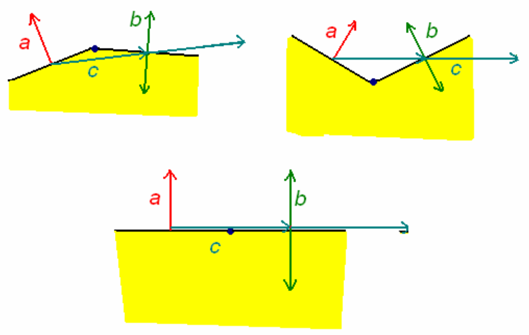


Fig. 5. (*upper left*) $a \bullet c < 0 \rightarrow b \bullet c > 0$, (*upper right*) $a \bullet c > 0 \rightarrow b \bullet c < 0$, (*bottom*) $a \bullet c = 0 \rightarrow a \bullet b > 0$.

5.10 Edge Analysis

The final step before assembling the Abstract Cellular Complex model of the objects is analyzing each edge. In Kovalevsky's description, each edge is also described with the faces and volumes it is adjacent to, in a right-hand rotation order.

To start this step, the program searches, in a copy of the list of the edge's adjacent faces, for a face that is not already adjacent to two volumes, that is, at least one face is "open" (if there is no open face, the program just picks a random face). The faces and volumes are traced out in order, either by measuring angles between faces or by finding entities (faces or volumes) adjacent to the most recently added entity. After all the adjacent faces have been exhausted from the copied list, the program checks if there is a volume (other than the first volume, in the case of having just two adjacent faces) between the last face and first face of the ordered list. If so, such a volume is added, along with the first face of the ordered list.

After the ordered list has been obtained, the program calculates the cross product of the vectors between the center of the edge and centers of the first two faces in the list. If the result is in the opposite direction of the edge (i.e., a negative dot product is obtained from these vectors), the list is then reversed. Thus, a list in the right-hand rotation order is obtained.

The last step is to determine the direction of the right-hand rotation in relation to the normal of each face, by calculating the cross product of the vector from the center of the edge to the center of the face and the face normal. The face is given a negative direction if the edge and normal are in opposite directions and positive otherwise.

5.11 Converting the Object into Abstract Cellular Complex Format

With all the data obtained from all the previous steps now sufficient to create the Abstract Cellular Complex of the original wireframe, the data is compiled into a single file, in this logical order:

- For each vertex: its coordinates, and the edges that enter and exit the vertex (with directions)
- For each edge: its start and end vertices, and the faces and closed volumes it is adjacent to (in right-hand rotation order)
- For each face: the volumes it is adjacent to, and its vertices and edges (in right-hand rotation order around its normal vector)
- For each volume: the faces adjacent to it

When reading the object in the new format, the program simply reads in the data in the same order it was output in the previous step, and displays it on the screen accordingly.

6. RESULTS

Performance-wise, the results (tested on a 996 MHz Intel Pentium) obtained at each step of the implementation for three sample shapes (see Fig. 6) are given in Tab. 1. All times are averaged from 4 runs.

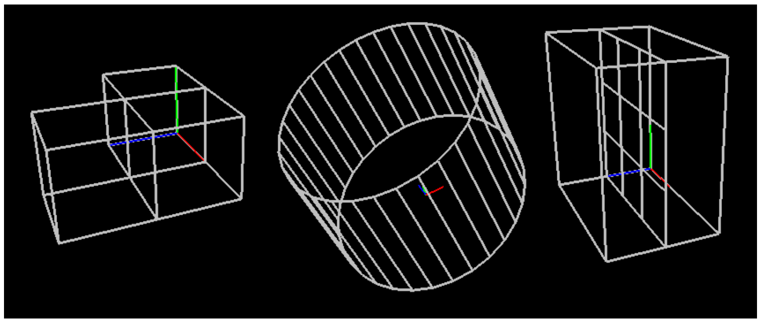
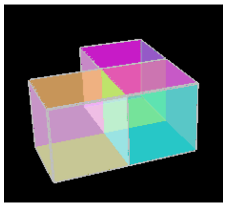
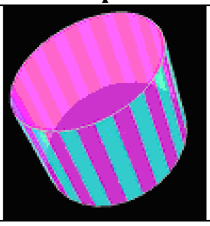
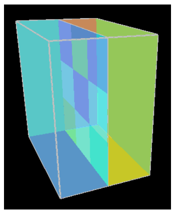


Fig. 6. The three sample shapes.

	Shape 1	**Shape 2**	**Shape 3**
An OpenGL rendering of the object and its obtained faces			
Finding vertices and edges	2.405 s, V=16, E=28	7.388 s, V=64, E=96	3.430 s, V=24, E=40
Finding faces	4.428 s	20.687 s	6.619 s
Finding unique faces	0.558 s, F=16	0.916 s, F=64	0.513 s, F=23
Reducing initial faces	0.726 s, F=16	2.523 s, F=32	1.091 s, F=19
Initial face fitting	0.335 s, F=16	0.343 s, F=32	0.343 s, F=19
Analyzing initial faces	0.243 s, F=16	6.206 s, F=34	0.578 s, F=19
Finding and analyzing manifolds	0.741 s, M=3	0.335 s, M=1	0.944 s, M=2
Processing manifolds	7.618 s	19.646 s	12.272 s
Assembling the ACC	0.188 s	0.964 s	0.313 s

Tab. 1. Experimental results.

The faces obtained according to the algorithm described above are, in most cases, equivalent to the most likely results that a person may obtain from inspecting the wireframe. However, there are also many exceptions. These include cases where the faces that are obtained are not flat (in the algorithm, with a calculated flatness of less than 0.9), and especially when complex images are processed.

7. FUTURE IMPROVEMENTS

Significant improvements to be made in the future include:

- Improving the manifold and surface finding algorithms to use a more elegant and all-purpose solution to its main flaw than the ad hoc solution described in section 5.8, and possibly to improve accuracy.
- Improving the face selection algorithm to be more accurate, especially where complex structures are concerned.
- Creating a program to enable actual editing of a file in Abstract Cellular Complex format. Editing an ACC is not as trivial as editing a wireframe, because, as mentioned earlier, so as to be able to obtain knowledge of topological relations among parts of a 3-D scene without searching, the structure contains some redundancy. Editing the structure in one portion therefore requires changes in other portions as well.

8. CONCLUSION

In conclusion, this paper describes a method for converting the wireframe data of an object into an Abstract Cellular Complex of the same object, by using many of the same methods used by Shpitalni and Lipson [4] to convert 2-D wireframe drawings into proper 3-D objects. It is believed that this method will prove useful in computer-assisted design.

9. REFERENCES

[1] Kovalevsky, V., Algorithms and Data Structures for Computer Topology, *Digital and image geometry: advanced lectures*, Springer-Verlag New York, Inc., 2001.

[2] Liu, J. and Lee, Y. T., A Graph-Based Method for Face Identification from a Single 2D Line Drawing. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. 23, No. 10, 2001, pp 1106 – 1119.

[3] Oh, B.-S. and Kim, C.-H., Progressive reconstruction of 3D objects from a single free-hand line drawing. Department of Computer Science and Engineering, Korea University, 2003.

[4] Shpitalni, M. and Lipson, H., Identification of Faces in a 2D Line Drawing Projection of a Wireframe Object. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. 18, No. 10, 1996, pp 1000 – 1012.

[5] Mohammad-Djafari, A., Sauer, K., Khagu, Y. and Cano E., Reconstruction of the shape of a compact object from few projections. *Proceedings of the 1997 International Conference on Image Processing*, 1997, pp 165-168.

[6] Pribanic, T., Cifrek, M. and Tonkovic, S., A simple method for 3D shape reconstruction, Proc. IASTED International Conference on Biomedical Engineering, 2004, pp 151-156.

[7] Shimanuki, M., Sato, H. and Akatsuka, T., Three-dimensional shape reconstruction from two-dimensional images using symmetric camera location, Proc. SPIE - The International Society for Optical Engineering Vol. 3313, 2005, pp 34-45.

[8] Hofer, M., Odehnal, B., Pottmann H., Steiner, T. and Wallner, J., 3D Shape Recognition and Reconstruction Based on Line Element Geometry. *Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05)*, Vol. 2, 2005, pp 1532-1538.

[9] Mello, V., Velho, L. and Taubin, G., Estimating the in/out function of a surface represented by points. *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, 2003, pp 108-114.

[10] Steiner, D. and Fischer, A., Topology Recognition of 3D Closed Freeform Objects Based on Topological Graphs. *Proceedings of the Symposium on Solid Modeling and Applications*, 2001, pp 305-306.