Finding Thin Points in an Abstract Cellular Complex

Varakorn Ungvichian
Department of Computer Engineering,
Chulalongkorn University
Bangkok, Thailand
ungvichian@thaimail.com

Pizzanu Kanongchaiyos
Department of Computer Engineering,
Chulalongkorn University
Bangkok, Thailand
pizzanu@cp.eng.chula.ac.th

ABSTRACT

This research describes an algorithm to find "thin points"' in a solid represented as an Abstract Cellular Complex. The algorithm is mostly iterative, adding one face at a time to a set of previously selected faces, and choosing the selections that produce the loops with the shortest lengths for the next iteration. The output from the algorithm is a set of loops that indicate the thinnest portions of the solid. As implemented, the algorithm allows a threshold to be set to limit the number of loops that are selected in each iteration. The results indicate that, while it does occasionally produce errors, the algorithm is mostly accurate, and a lower threshold increases its speed, without negatively affecting its accuracy.

Keywords

Geometry, computer graphics, Abstract Cellular Complex, topology

1. INTRODUCTION

Our algorithm focuses on finding "thin points"' in a solid represented as an Abstract Cellular Complex. "Thin points" refers to areas where the solid is narrowest in the current locality. Mathematically defined, a "thin point" is a point p on the surface of a solid where the radius r of the largest sphere that can be inscribed within the figure and be tangent to that point is the smallest in the current locality, that is, $r'(p) = 0$, $r'(p - \partial p) < 0$ and $r'(p + \partial p) < 0$. One potential application for finding the thin points of a solid would be to find the weakest points of a structure.

In the program, we will actually search for loops, comprised of a set of adjacent edges where the length of the loop is the smallest, which should provide a reasonable approximation. The only potential difference is when the loop at that point is a highly-concave figure.

2. PREVIOUS LITERATURE

Previous methods to find the "thin points" in images have used Hessian matrices. For example, Sukanya et al. [Suk96] describe a new operator for image structure analysis based on Hessian matrices, which describes the shape of each pixel. The paper also describes 4 types of surface shape: Dale, valley, ridge, and hill.

Florack and Kuijper [Flo00] use the "catastrophe theory", also based on Hessian matrices and critical points, to find "extrema" and "saddles" in localised portions of images.

Danielsson and Lin [Dan01] also use the Hessian, along with spherical harmonics, for shape detection in both 2D and 3D. However, instead of searching for "thin points", the 3D portion of the research concerns detecting "strings", i.e., long curvilinear shapes. Similarly, Koller et al. [Kol95] concentrate on finding "line-like" structures. Both pieces of research cite finding blood vessels in MRI data as applications for their algorithms. It should be noted that most existing research designed for 3D images concentrates on finding lines, generally with medical applications. However, Danielsson and Lin's research is derived from a more general overview [Dan98] of a derotation algorithm for 3D segmentation.

Here, we will describe a method to find the 3D equivalent of a saddle in a 3D solid. Also, the method used in this algorithm is somewhat axis-independent, depending mostly on the adjacency between the faces, and the lengths of the edges.

The data structure used to represent the solids (the input to the algorithm) is the Abstract Cellular

Complex, devised by Kovalevsky [Kov01]. It is designed to efficiently represent topological data, by describing, for each element of the solid, which other elements of different dimensions it is adjacent to (e.g., faces adjacent to a given edge, edges adjacent to a given vertex, etc.), so that relations between parts of the image can be found with limited searching. In particular, it explicitly identifies which faces are adjacent to a given edge, and what edges comprise a given face. This makes the algorithm efficient.

3. THE ALGORITHM

The structure of the algorithm is thus:

1. Determining the axis of the shape (optional) and selecting a face
2. Creating new selections by adding faces to the current selection, and finding the selection with the smallest total length
3. Limiting the number of selections created according to the length ("thresholding")
4. Processing the selections to determine the thinnest points

While loading the Abstract Cellular Complex data, we pre-calculate the length of each edge. After the data has been loaded there is an optional "axis detection" step. This step is designed to find a good face to start the next step from. This step begins with averaging the coordinates of the vertices of the volume to find an approximate center. We also average the coordinates of the vertices of each face to obtain each face's center as well.

Next, we split the volume into sets of vertices by the *x*-value of each vertex. In our algorithm, we have chosen to split them into 20 sets. We find the averages of the coordinates of each vertex in each set, and use linear regression to determine the line that best fits the 20 given averages.

We repeat this process using the *y*- and *z*-values instead of the *x*-value. In Figure 1, we show the results for the *x* and *y* axes for one of the example shapes we will be using. (For the illustration, we have also simplified it down to just 10 sets of vertices.)

Having obtained three different lines, we determine which of them has the best fit, by summing the distances between the average of each set and the line. In the case of Figure 1, the line obtained after splitting the shape by the *x*-axis has a better fit than that obtained after splitting the shape by the y-axis.

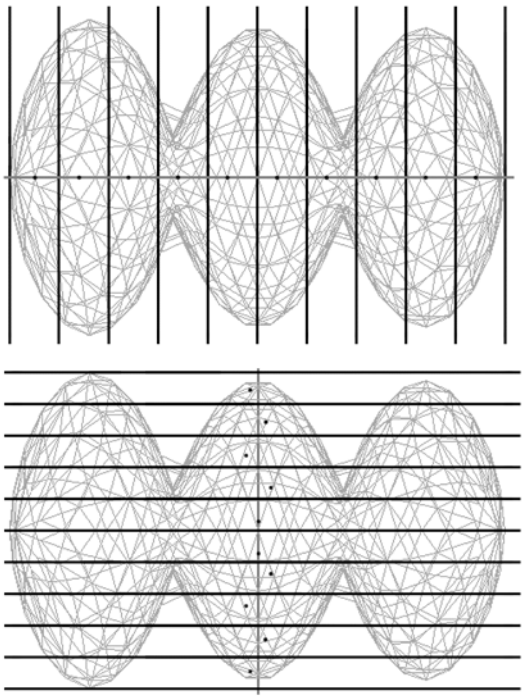

Figure 1. Initial splitting.

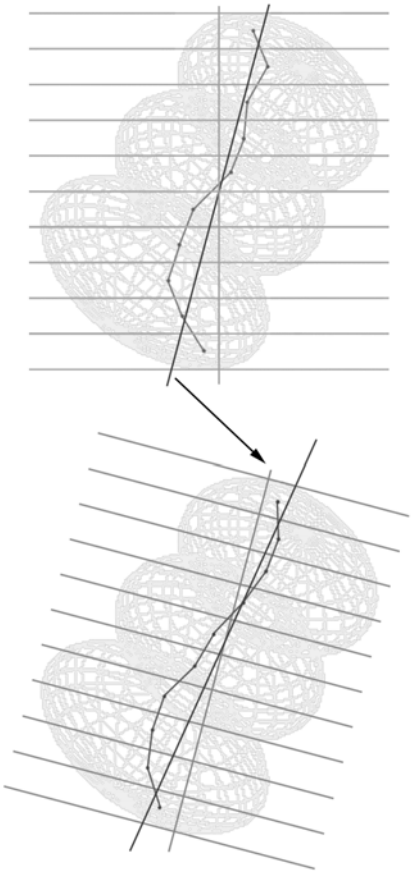

Figure 2. Iterative splitting.

After obtaining the best-fitting line, we then split the vertices into sets again, using this line as the axis. Once again, we find the averages of the coordinates of each vertex in each set, and use linear regression to determine the best-fitting axis. We then repeat this step with the newly-determined axis until the new axis deviates from the old axis by less than 0.1°. See Figure 2, as done with another example shape. If the axis does not converge after a certain number of steps, we average the most recent axes obtained and use that as the axis.

We determine the plane that passes through the approximate center and is normal to the best-fitting line, and find the face whose center is furthest from the plane. We select this face, and sum the lengths of its edges, and store the total length in an array.

(If axis detection is not used, or produces an error, the program simply takes the first face listed for the volume. In our earlier work [Ung06], which this research augments, this face contains the vertex with the smallest x-value.)

The next step of this algorithm is iterative. We find all the faces that are adjacent to a face in each current selection. A "selection" is the set of faces in the solid that have been selected, and a selection is represented as a simple Boolean array, with each value in the array corresponding to whether the corresponding face has been selected. In the first step, the first face picked is the only current selection.

For each face that is adjacent to a previously-selected face in the selection, we add the face to the selection. To save time, we only select faces that are adjacent to the most number of faces of the current selection. In other words, if there are faces that are adjacent to 3 faces of the current selection, and none that are adjacent to at least 4, the program will add only those faces adjacent to 3 faces of the current selection. In Figure 3, there are three faces that are attached to 3 selected faces, one attached to 2 selected faces, and five attached to one selected face. The three faces (as highlighted in the figure) will be added.

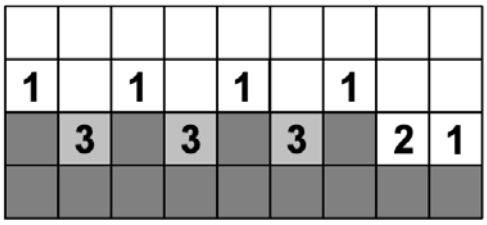

Figure 3. Face preference.

Having added a face to the selection, we test to see if it is a duplicate of a previous selection by running an exclusive-or operation:

$$\bigvee_{i=0}^{n-1}\left(\mathrm{sel}_a(i) \oplus \mathrm{sel}_b(i)\right) \rightarrow \neg\left(\mathrm{sel}_a(i) \sim \mathrm{sel}_b(i)\right)$$

If it is not a duplicate, we sum the lengths of the edges with one selected face attached. (We have tested other heuristics, and found that this one produces more accurate results than with the others.)

Our algorithm uses other "shortcuts" to determine which edges have just one selected face attached and to sum the lengths of those edges. First, in each selection, we maintain two lists, a list of edges with one selected face attached (L_1) and a list of edges with (at least) two selected faces attached (L_2). As we read each edge in, if it is not in either list, we add it to L_1. If it is already in L_1, it is moved to L_2.

To sum the length of the edges in L_1, we also maintain an array of the total lengths that were calculated in the previous step. As we add the face to the selection, we also determine which faces are being added or removed from L_1, before adding and subtracting the lengths of those edges from the previously-calculated length to determine the new total length. Directly summing all the edges in L_1 could become unwieldy, especially when there are hundreds of edges in the list.

After processing all the selections and their adjacent faces, we determine which selection has the least total length of edges in L_1. To save processing time, we then remove selections whose total edge length is higher than 1.1 times that least total length. Further limitations may be made as necessary. (More will be explained in Thresholding.) The remaining selections (and their total lengths) are stored in an array for the next round of processing. We also store the least total length obtained for the current iteration in an array (along with a list of the edges that produces the result, i.e., the L_1 of the selection).

The number of iterations for this step is the same as the number of faces in the solid. After the iterative step is complete, we adjust the values, so as to favor selections from "midway" through the process, rather than those at the beginning and end. We do this by using this equation:

$$v_i = \frac{\min(i, f-i)}{l_i}$$

where i is the number of the current loop, f is the total number of faces in the solid, and l_i is the least total length of the current loop.

After passing the values through the equation, the most preferable values will have a high amount instead.

If necessary, we then clean up the values further by averaging adjacent values. This is done when the values tend to alternate between going up and down (e.g., in a solid with triangular faces).

Next, we calculate upward spikes in the values with another equation:

$$s_i = \begin{cases} (v_i - v_{i-1}) \times (v_i - v_{i+1}) & \text{if } v_i \geq v_{i-1} \\ -(v_i - v_{i-1}) \times (v_i - v_{i+1}) & \text{if } v_i < v_{i-1} \end{cases}$$

The result of this equation produces positive values at upward spikes, which correspond to small loop sizes.

Next, we measure the "distance" (i.e., the number of loops) between the current loop and the closest loop that produces a higher value. For example, if s_{120} = 200, s_{100} = 250, s_{150} = 300, and $\forall 100 < x < 150 : s_x < 250$, then the distance is 20, as $|120-100| = 20 < 30 = |150-120|$.

Therefore, we multiply $s_{100}$by 20 to produce the final value of 250 × 20 = 5000.

We then output the final values that are larger than $\frac{1}{50}$ of the largest final value, as the thin points of the solid.

4. THRESHOLDING

As implemented, the program allows its user to select the threshold of number of selections that remain after each loop of the iterative step (min. 10, max. 1000). When the number of selections that remain, after removing those with a total length more than 1.1 times the least total length, still exceeds the threshold, the program sets a new length limit:

$$t_0 = 0.1 \times \min(\frac{n_t}{n_c}, 0.95)$$

where n_t is the threshold, and n_c is the actual number of selections. The program then removes selection with a total length more than (1+t_0) times the least total length. If necessary, the program adjusts the length limit further:

$$t_i = t_{i-1} \times \min(\frac{n_t}{n_c}, 0.95)$$

If, after 20 iterations of adjusting the length limit (i>20), the number of selections that remain still exceeds the threshold, the program uses a sorting algorithm to find the selections with the smallest total lengths, up to the threshold (e.g., if the threshold is 40, we take the 40 selections with the smallest total lengths).

The pros and cons of high and low thresholds are thus:

- *High threshold (more selections to consider)*

  Pro: More accurate

  Con: Takes more time

- *Low threshold (less selections to consider)*

  Pro: Takes less time

  Con: Potentially less accurate

5. EXPERIMENTAL RESULTS

The algorithm was run on a set of examples, as illustrated in the following Figures.

Each figure was made in two versions: one with quadrilateral faces, and another with mostly triangle faces.

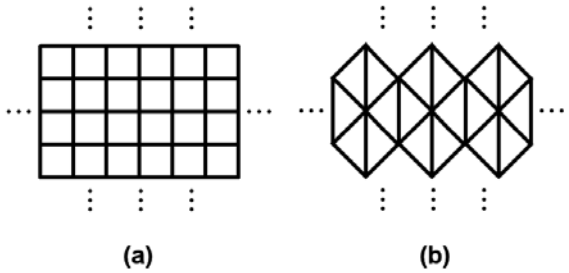

Figure 6. (a) Quadrilateral faces (b) Triangle faces

The results obtained from these examples (with a 40 threshold) are shown in the following Figures 7 to 10, with the loops obtained highlighted in black.

The results show that the algorithm, while it has some degree of accuracy, still needs improvement. In particular, the algorithm produces spurious results in two examples, and it completely misses another thin point in one of the other examples.

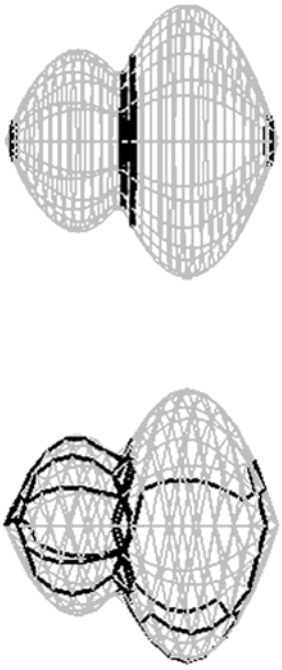

Figure 6. Example 1 results.

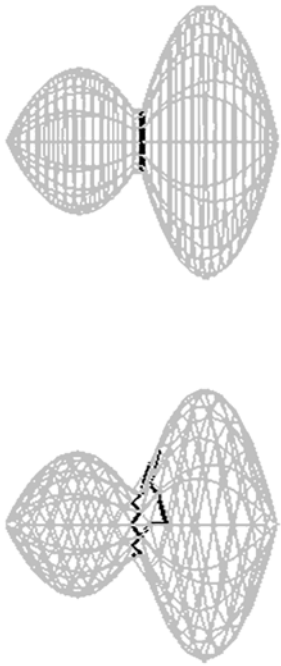

Figure 7. Example 2 results.

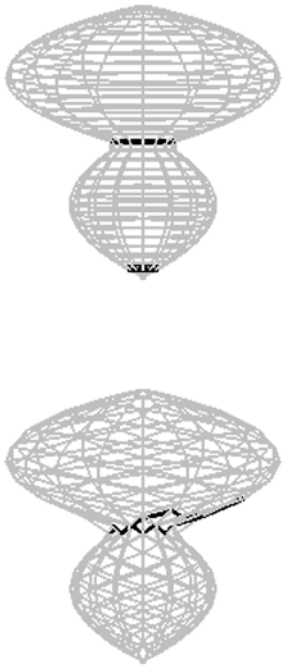

Figure 8. Example 3 results.

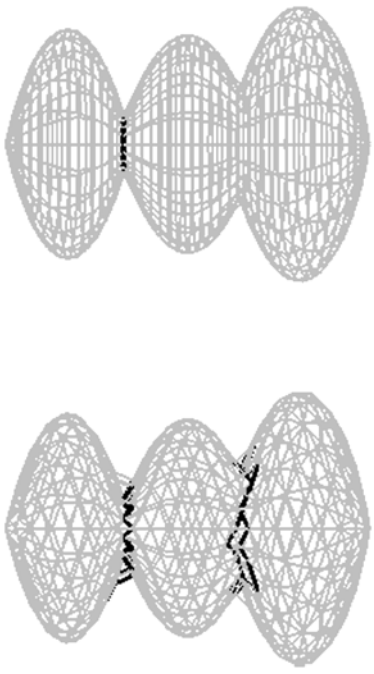

Figure 9. Example 4 results.

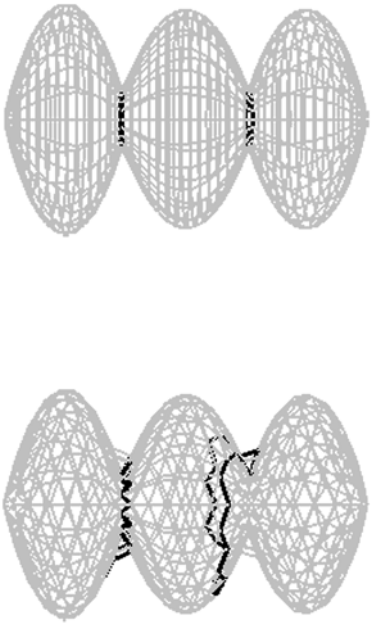

Figure 10. Example 5 results.

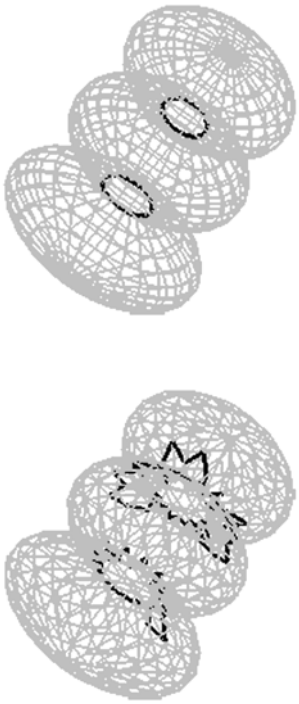

Figure 11. Example 6 results.

The execution times are shown in Table 1. The results indicate a much longer execution time for solids with triangular faces. The reason this is so is that quadrilateral faces take advantage of the face preference time-saving measure, while the triangular faces do not.

Ex.	Faces (Quad.)	Time (Quad., s)	Faces (Tri.)	Time (Tri., s)
1	480	19.598	448	95.858
2	480	22.162	448	136.556
3	480	22.282	448	134.103
4	960	73.946	912	1009.011
5	960	72.534	912	802.224
6	960	70.031	912	587.915

Table 1. Execution time

Due to the lengthy execution times for the solids with triangular faces, the experiment then looked into the effects of lowering the threshold.

This experiment was done on the third example, and the results are in Table 2.

Threshold	**10 (min.)**	**40**
Time	14:30.442 s	26:24.408 s
Threshold	**80**	**120**
Time	1:01:19.721 s	2:00:51.497 s

Table 2. Execution time for different thresholds

The results were the same as the 40 threshold for the 80 and 120 thresholds, but different for the 10 threshold. This result shows that while lowering the threshold also lowers the execution time, it does not significantly adversely affect the accuracy.

6. CONCLUSIONS

As currently implemented, the algorithm that has been described in this paper is effective in finding the "thin points" in an Abstract Cellular Complex. This algorithm has potential applications where finding the thinnest part of a structure represented as a 3D mesh is necessary, for example, to find a possible weak spot in a structure, or a potential spot to "chop" the structure into two parts or more.

Current drawbacks of the algorithm are that it has limited sensitivity, that it assumes that the shortest loop is also the smallest loop (an assumption that does not hold when the smallest loop is very concave) and that, in the worst case, it takes exponential time to discover the thin points. Future potential improvements include finding better heuristics to determine the smallest loops, increasing the speed and efficiency of the algorithm, and increasing the sensitivity while decreasing the spurious results obtained with the algorithm.

7. REFERENCES

[Dan98] Danielsson, P.-E., Lin., Q., and Ye, Q.-Z. Segmentation of 3D-volumes Using Second Derivatives. Proc. 14th Int. Conf. on Pattern Recognition, pp. 248-251, 1998.

[Dan01] Danielsson, P.-E., and Lin., Q. Efficient detection of second-degree variations on 2D and 3D images. Journal of Visual Communication and Image Representation, Vol. 12, pp. 255-305, 2001.

[Flo00] Florack, L., and Kuijper, A. The topological structure of scale-space images. Journal of Mathematical Imaging and Vision, Vol. 12, pp.65-79, 2000.

[Kol95] Koller, T., Gerig, G., Székely, G., and Dettwiler, D. Multiscale Detection of Curvilinear Structures in 2-D and 3-D Image Data, 5th International Conference on Computer Vision, pp. 864-869, 1995.

[Kov01] Kovalevsky, V. Algorithms and data structures for computer topology. Digital and image geometry: advanced lectures, pp. 38-58, 2001.

[Suk96] Sukanya, P., Takamatsu, R., and Sato., M. A new operator for image structure analysis. Proceedings of the International Conference on Image Processing, Vol. 3, pp 618-658, 1996.

[Ung06] Ungvichian, V. and Kanongchaiyos, P. Mapping A 3-D Model into Abstract Cellular Complex Format. Computer-Aided Design and Applications Journal, Vol. 3, pp. 395-404, 2006.

Columns on Last Page Should Be Made As Close As Possible to Equal Length