An Implementation of Chen & Han's Shortest Paths Algorithm

Biliana Kaneva Joseph O'Rourke*

Abstract

In 1990 Chen and Han proposed a quadratic algorithm for finding the shortest paths from one source point to all vertices on a polyhedral surface. We report on a C++ implementation of their algorithm, to our knowledge the first publicly available implementation of the complete algorithm.

1 Introduction

In 1990 Chen and Han proposed a quadratic algorithm for finding the shortest paths from one source point to all vertices on a polyhedral surface [CH90a, CH96]. Let a polyhedral surface P have n vertices. P may be convex or nonconvex, with or without boundary, i.e., it is a piecewise-linear 2-manifold. Let s and t be two points on P. The problem is to find a shortest path on the surface from s to t; call this the *point-pair shortest path* problem. Often one needs the shortest paths from s (the source point) to all other points on the surface. Knowing a shortest path to each vertex suffices to determine the paths to all other points. Call the more general problem the *single-source shortest paths* problem on a polyhedral surface. Existing algorithms solve the point-pair problem by solving the single-source problem.

After an early $O(n^5)$ algorithm [OSB85], an $O(n^2 \log n)$ algorithm was developed using what the authors called the "continuous Dijskstra" technique [MMP87], which propagates wavefronts of points equidistant from the source. This remained the best asymptotic bound until Chen and Han's rather different $O(n^2)$ algorithm [CH96].

Although one cannot improve upon quadratic if the n paths are represented by explicit segments, one per crossed face, there were two reasons to suspect the result could be improved: the point-pair problem has output size only $O(n)$; and even for the single-source problem, it is possible to compress the output below quadratic [CH90b].

During the 1990's, Chen & Han's algorithm was superceded in two ways. First, a series of sophisticated approximation algorithms were developed [LMS97, AHPSV97, VA97, HP99]. Second, Kapoor finally broke the quadratic barrier with a wavefront propagation algorithm of $O(n \log^2 n)$ time complexity [Kap99][O'R99].

Despite these advances, Chen & Han's algorithm remains the only currently feasible method of computing the exact shortest paths on a polyhedral surface: the approximate algorithms are not exact, and Kapoor's algorithm is complex.

Although the Chen & Han algorithm is conceptually simple, and has been available for over a decade, implementation is not straightforward. To our knowledge, only two implementations have been attempted: one by the second author and Xu [Xu96], and one by Lanthier et al. [LMS97]. The former only implements the easier, convex part of Chen & Han's algorithm, and is not entirely robust even in that case. The latter is a full implementation, but the code is proprietary and its performance has never been discussed in print.

We offer in this paper a description of a C++ implementation of the full Chen & Han algorithm, and discuss its performance. The code is publicly available.[1]

2 Summary of Results

1. Roundoff error is not a serious issue for this algorithm, within the range we were able to test.[2]

2. The time used by our code does increase at what appears to be a quadratic rate with respect to n, the number of vertices.

3. Memory usage grows at closer to a linear rate with n, even without implementing the memory conservation steps necessary to ensure $O(n)$ growth.

4. Nevertheless, we found memory to be the limiting factor. We could not feasibly run examples with more than $n \approx 10^4$ vertices (2×10^4 faces).

*Dept. Comput. Sci., Smith College, Northampton, MA 01063, USA. {bkaneva,orourke}@cs.smith.edu. Supported by NSF grant CCR-9731804.

[1] `http://cs.smith.edu/~orourke/code.html`.

[2] This agrees with the experience of [LMS97] (Jörg Sack, personal communication, Sep. 1997).

5. We found it essential to supplement the Chen & Han algorithm with shortest paths clipping, which in cases produced a 20-fold reduction in memory requirements.

6. Extensions of the implementation from convex to nonconvex surfaces, and to surfaces with boundary, are nontrivial.

7. Collinear vertices on nonconvex surfaces demand careful coding to avoid exponential explosion.

3 Algorithm Description

Chen & Han's algorithm is notable for *not* following the Dijkstra paradigm so ubiquitous in shortest-path algorithms. At the top level is a loop over all F faces. We assume the surface is triangulated, so $F = 2n - 4$, where n is the number of vertices. The algorithm builds a "sequence tree," each node of which represents a set of shortest paths, angularly contiguous at the source s, all crossing a particular face f, and all having the same *edge sequence*: the ordered sequence of polyhedron edges crossed. We call each such bundle of shortest paths a *cone*, and the structure the *cone tree*.[3] A cone that has crossed ℓ faces is said to be at *level* ℓ.

Each iteration of the loop selects a leaf of the cone tree for "rolling out" over one face f, across the last edge e of its edge sequence. The extended cone may encompass the vertex v of f opposite e. In this case, the cone splits to two children. Exponential explosion is avoided by Chen & Han's key "one-split" lemma: If two cones occupy v from the same edge e, then at most one cone has two children that include shortest paths.

Let vertex v of the polyhedron be incident to t triangles. Then at any one time, v may have as many as t *occupiers*, that is, cones that occupy it over the opposite edges of the incident triangles. Although there can be many occupiers for a vertex, across each edge e there is just one, representing the shortest path to v whose last edge crossed is e. Although not explicit in [CH96], these occupiers must be maintained for each vertex. Then when a rolled-out cone newly occupies a vertex v via e, it is compared to the current occupier across that edge e, and the one-split rule applied. If the new occupier "wins" (i.e., has a shorter path to v), then one of the old occupier's children is deleted. Note that this child could represent the root of a substantial cone subtree. If the old occupier wins, then the new occupier has only one child.

After the main loop has iterated F times, the best occupier of each vertex represents the shortest path to that vertex. Note that the paths do not grow in shortest-first fashion; rather the frontier represents all cones that have crossed ℓ faces, regardless of their length.

Extension to nonconvex surfaces is accomplished by treating each negative-curvature vertex (one whose total incident face angle exceeds 2π) as a new "pseudo-source" point.

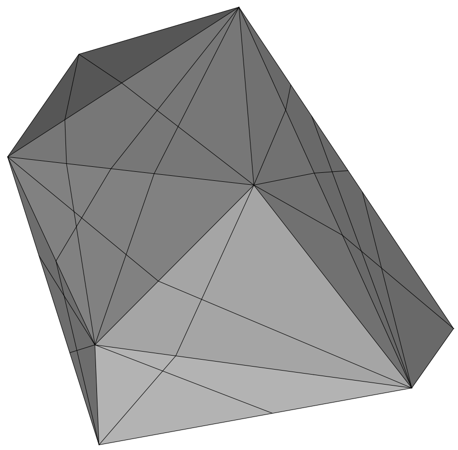

Figure 1: The shortest paths between all pairs of vertices.

4 Implementation Description

We concentrate on what we consider the notable aspects of our implementation.

4.1 Source Point

Although it is easiest to restrict the source point to be interior to a (triangle) face, for then it always has just three incident level-0 cones, our implementation permits the source to be at a vertex or on an edge. Many applications demand this flexibility. For example, our work on Cauchy rigidity [O'R00] requires the shortest paths between every pair of vertices; see Fig. 1.

4.2 Cones

Chen & Han take the viewpoint of fixing a face f and its apex v, and arranging "source images" around f, so that a segment from v to a source image represents an unrolling of the path backwards from v to s. This viewpoint is useful when performing the one-split computations. However, we decided to reorient the entire algorithm on the basis of a fixed source s. Of course the two viewpoints are geometrically equivalent, but the choice affects many aspects of the implementation.

With a fixed source s, the shortest paths are circularly arrayed around s in cones, at all times covering 2π. Suppose s coincides with a vertex of degree d. Initially there are d cones, one for each triangle face f

[3] We will use our own terminology rather than that in [CH96].

incident to s, covering the edge of f not incident to s. This is level 0. Level 1 rolls out those cones across the adjacent faces. And so on. Any particular vertex of the polyhedron will have many "images" in the plane, representing rollouts along different routes to that vertex. See Fig. 2.

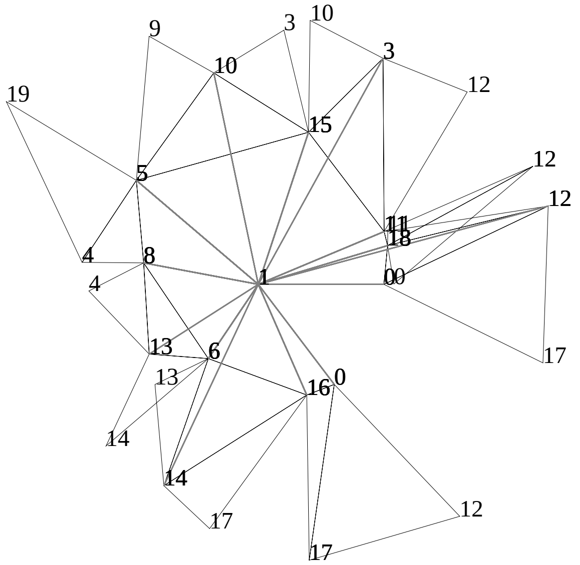

Figure 2: Rollout of cones to level $\ell = 3$ for a small polytope about the source 1. Vertex 19 has only one image at this level; vertex 12 has four images. Cone edges are light; triangle edges dark.

We represent each cone by its source point, and its left and right *bounding vertices*, which are planar images of rolled-out polyhedron vertices.

4.3 Rolling Out

It is natural to perform the rolling out of the faces crossed by a path via composition of rotation matrices [NF93], which leads to coordinates expressed in the form

$$x_i = \sum_{k=0}^{m} c_k \underbrace{\cos\alpha_{i_0} \ldots \cos\alpha_{i_{k-1}}}_{k} \underbrace{\sin\alpha_{i_k} \ldots \sin\alpha_{i_{m-1}}}_{m-k}$$

where the α_{i_k} are various triangle angles along the edge sequence crossing the m faces $i_0, \ldots, i_{m-1}$ rolled out. However, in our earlier experience [Xu96], this led to robustness issues. We therefore treat the input as entirely two-dimensional: a collection of triangles and edge gluing rules. The orientation of a rolled out face f in the plane with respect to the fixed source is then determined by a simple sum of angles at one or the other endpoint of the edges crossed by the cone to f. Which endpoint is determined by which side of each triangle's apex the cone passes:

$$\begin{aligned} x_i = &\quad a_k \cos(\alpha_{i_0} + \cdots + \alpha_{i_{m-1}}) \\ + &\quad b_k \sin(\alpha_{i_0} + \cdots + \alpha_{i_{m-1}}) \end{aligned}$$

Note the sum of products of m trigonometric terms from the rotation matrix multiplication has been replaced by two trigonometric functions applied to a sum of m angles. The orientation (as computed above) together with the coordinates of one vertex (from the previous face rollout) determine the layout of f in the plane. We have observed no numerical instabilities in this computation as far as we tested it, to $n \approx 10^5$. See Fig. 3.

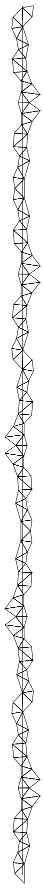

Figure 3: Rollout of a geodesic on a pyramid.

4.4 Clipping Cones

Because of repeated cone splitting, as the algorithm progresses, the cones become angularly thin. These thin cones continue to rollout across faces, rarely encountering a vertex, and even then often losing to the previous occupier. Recall that Chen & Han's algorithm pushes each cone to level $\ell = F$ regardless of the length of its contained paths.

We found it essential to supplement the algorithm with an additional clipping module, which operates as follows. For each face f, for each of its edge pairs, we maintain a list of cones that cross its interior from edge-to-edge. When a cone c is newly rolled out across f, if it does not cover the opposite vertex, it crosses from edge-to-edge. The relevant oppositely crossing cones are retrieved, and a pairwise comparison is performed on each such cone c'. If either c dominates c' in the sense that it is better to shortcut from c' into c, or vice versa, then the appropriate cone (and the subtree it roots) is clipped from the cone tree. This produced a 20-fold reduction in memory requirements on a convex polytope of 500 vertices. All further results we mention include this modification to the basic algorithm.

4.5 Nonconvex Complications

When shortest paths wrap around a negative-curvature vertex v (henceforth, *negative* vertices), v acts like a new source vertex, a *pseudo-source*, potentially with as many children as there are faces incident to v. Thus the one-split rule does not hold. We implement cone splitting at negative vertices as follows. Suppose a cone c newly occupies a negative vertex v. Rather than splitting to at most two children, c may have a left child, a right child, and a series of *middle children* to cover the faces more than π counterclockwise, and more than π clockwise, from the central segment $s'v$ from the previous (pseudo-)source s' to v. There is no limit to the angular extent covered by the middle children, as a negative vertex can have arbitarily large total incident face angle.

4.6 Boundary Vertices

Vertices on the boundary of a polyhedral manifold are special in two ways: they are not entirely surrounded by faces; and even if they have positive curvature (total incident face angle $< 2\pi$), they may serve as pseudo-sources. Rather than implement the considerable special-case code these would require, we implemented only *terminating boundary vertices*, which do not split cones. We then surround a given terrain with a rectangle whose four corners are terminating boundary vertices. See Fig. 11. Extending our code to work on terrains with interior boundaries, (e.g., surrounding a lake) would require a more general approach.

4.7 Collinearities

A somewhat subtle problem can arise from collinear negative vertices, a situation not infrequent in terrains defined over grids. These naturally lead to cones with collinear left and right bounding points, i.e., cones of zero width. When two such overlapping "needle cones" approach a vertex from the same direction, they are of course tied for distance to s. If tied cones both have two children, exponential explosion results. If collinear tied cones are selected randomly for continuance, gaps in coverage may result, as some of these cones are needed to create new pseudo-sources. We found it a delicate matter to handle tied collinear cones appropriately.[4]

5 Results in Detail

Our implementation, excluding graphics, is 2446 lines of C++ code. We ran all our test examples on a Linux PC with a Pentium III 600 MHz processor and 256MB RAM.

5.1 Correctness

Evidence for correctness was obtained in two ways: by verifying that no two shortest paths cross, and with a collection of special examples where at least some shortest paths are known exactly. The former is a more stringent test than one might initially suppose: even one incorrect path forces reorganization of many paths to avoid crossings. In our experience, every error during development led to crossing paths. We did not, however, implement an automatic check for the noncrossing condition, although it would be useful to do so; rather we relied on visual inspection.

We used four special classes of examples with known shortest paths for testing correctness:

1. Flat, rectangular terrains sprinkled with interior vertices. Here the shortest paths are simply straight lines from the source to each vertex.
2. Simple orthogonal polyhedra, e.g., Fig. 4, where the shortest paths are obvious.

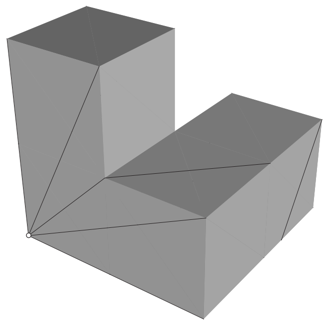

Figure 4: A polyhedron of $n = 20$ vertices ($F = 36$).

3. "Snake terrains" (e.g., Fig. 5) where the shortest path travels down a deep, snaking channel surrounded by high wall.

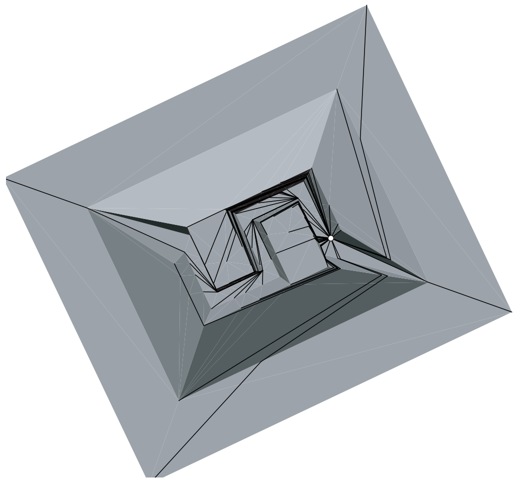

Figure 5: Snake terrain, $n = 26$, $F = 46$.

4. Cylinders and cones, which when sufficiently tall have a clear pattern of shortest paths. Fig. 6 shows

[4] This does not seem to be addressed in [CH96, p.136].: "If there are several nodes which tie on the same angle we store the two outermost tying nodes for the angle."

a squat cylinder with a more complex pattern of shortest paths, which is nevertheless a good test. From the source s on the top rim to vertices on the bottom rim and roughly diametrically opposite s, there are two equal-length shortest paths: a chord on the top face and down the back band, or down the front band and a congruent chord on the bottom face. In the figure these ties are broken arbitrarily, resulting in a random choice between the two routes.

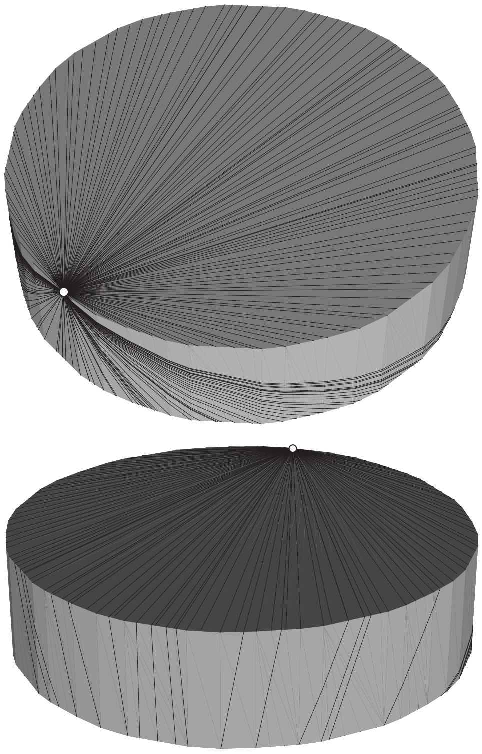

Figure 6: Two views of a cylinder whose height is half its radius; $n = 196$ vertices ($F = 388$).

5.2 Convex Polytopes

We tested performance on three types of polyhedral surfaces: convex polytopes, terrains, and orthogonal polyhedra.

Fig. 7 shows the paths on an ellipsoidal polytope of about $n = 100$ vertices, and Fig. 8 shows the paths on a spherical polytope of about $n = 500$ vertices on the surface of a sphere.[5]

The largest convex example we were able to run has $n = 10473$ vertices and $F = 20942$ faces. This took 90 minutes, and required 281MB of memory. Because this exceeded our machine's 256MB, swapping ensued, and the time result is correspondingly skewed. The

[5]The polytopes were generated with `sphere.c` and `chull.c` from [O'R98].

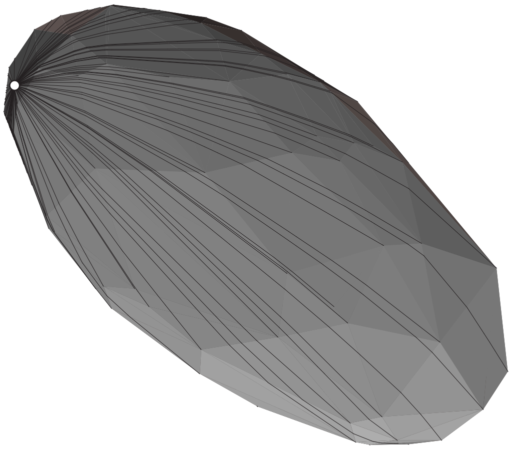

Figure 7: An ellipsoid of $n = 99$ vertices ($F = 194$); axis lengths 100, 50, and 25.

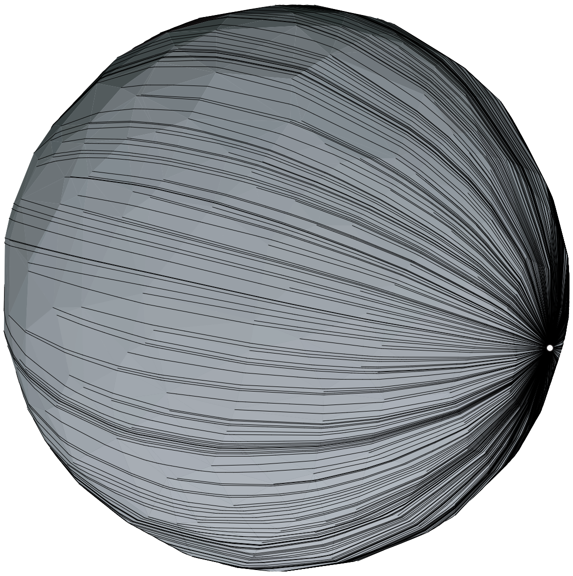

Figure 8: Convex polytope of $n = 493$ vertices and $F = 982$ faces.

times for the smaller examples shown in Fig. 9 are more representative.

Two points to note. First, the graph shows clear superlinear growth, which is not surprising because Chen & Han's algorithm is fundamentally quadratic. Second, even for the smaller sizes, the time requirements are substantial, e.g., 27 minutes for $n = 7660$. However, in this latter example, the shortest paths to these 7660 vertices are composed of 664571 segments, all of which are stored and printed by the code. This amounts to $1620/664571 \approx 0.002$, i.e, 2 milliseconds per segment.

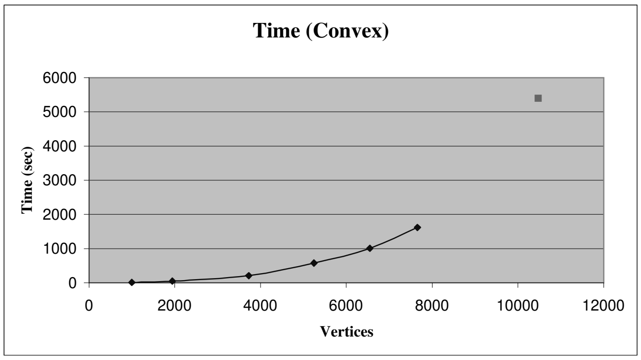

Figure 9: Time for convex polytopes. The data point for $n \approx 10^5$ includes swapping time.

Now we remark on comparison to the work Lanthier et al. [LMS97]. Their implementation of Chen & Han's algorithm is incidental to their work, and consequently they report only one aspect of its performance: a time of 164 seconds to compute a path on a terrain of $n =$ 5044 vertices ($F = 10082$) [their Fig. 7]. This data point is obviously greatly at odds with our own timing results. We feel that in fact no direct comparison can be made, for the following reasons:

1. Their timing reflected the computation of a path between a source s and one particular destination t, not between s and all vertices.
2. The (s, t) pair was chosen randomly, so even on a large terrain, they may have been quite close.
3. They stopped the Chen & Han propogation when t was reached; so the whole tree was not computed.[6] (Consequently, they may not have found the true shortest path.)

5.3 Nonconvex Terrians

We have only explored terrains[7] with a rectangular boundary. We generated random terrains, with a variety of degrees of "roughness."[8] Fig. 10 shows a rough terrain of about 1000 vertices. Fig. 11 shows an overhead view of a smoother terrain with about twice as many vertices.

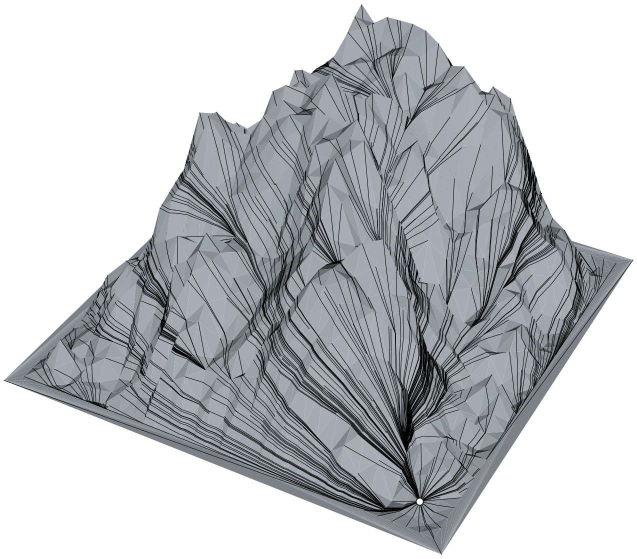

Figure 10: Rough terrain, $n = 1093$, $F = 2180$.

The largest terrain we ran had $n = 2597$ vertices and $F = 5188$ faces. Again we were memory limited. Fig. 12 shows the peak memory usage for the mild terrains.

We did not implement the memory efficiencies necessary to guarantee that the algorithm uses only $O(n)$ memory. These efficiencies include retaining "only the leaves and the interior nodes which have more than one child in the current tree" [CH96, p.136]. Although this is easily done, the path of segments for each shortest path is then no longer explicitly stored in the cone tree, and would have to be regenerated in a post-processing phase. More subtle is an issue only indirectly addressed in [CH96]: our direct implementation of the handling of negative vertices permits such a vertex of degree d, occupied by $d/2$ cones, each to generate $d/2$ middle children cones. This quadratic behavior could be skirted by recognizing and throttling the duplication on-the-fly during the generation phase. Nevertheless, Fig. 12 shows that the growth of memory is approximately linear.

Nor did we attempt to conserve space on each cone record; ours consumed 216 bytes each. Thus the 196MB data point in Fig. 12 represents 810676 nodes in the cone tree, which accounts for 175MB of the observed memory usage. (Other data structures consume the remainder: lists of vertex coordinates, lists of occupiers for each vertex, lists of crossing cones for each face, etc.)

[6] Mark Lanthier, personal communication, Mar. 2000.

[7] A *terrain* is a surface that meets each vertical line in at most one point.

[8] We used the midpoint-displacement method; see, e.g., [HB97, p. 373ff].

Figure 11: Overhead view of a terrain of $n = 2140$ vertices and $F = 4274$ faces.

Because any full implementation of Chen & Han's algorithm must construct the cone tree (if not retain it in its entirety), we expect that even with effort to reduce space, the algorithm will remain memory-limited, albeit for somewhat larger n.

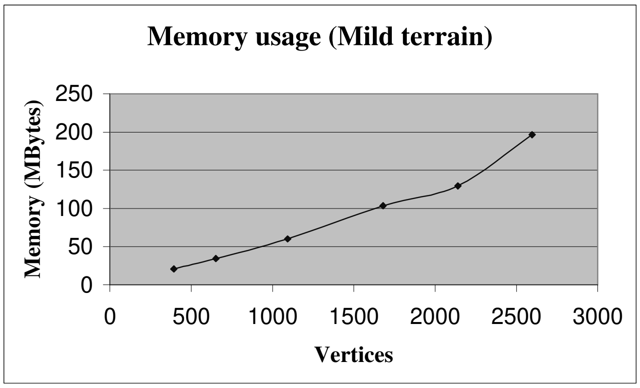

Figure 12: Peak memory usage in MB.

5.4 Orthogonal Polyhedra

The final class of examples we explored are (closed) orthogonal polyhedra, with all faces meeting at right angles. We generated our orthogonal polyhedra by gluing triangulated unit cubes together. These examples constitute a stringent test for collinearities, for any edge of length L contains $L-1$ interior collinear points. For example, in Fig. 13, the longest edges include 10 vertices: the two endpoints and 8 interior points.

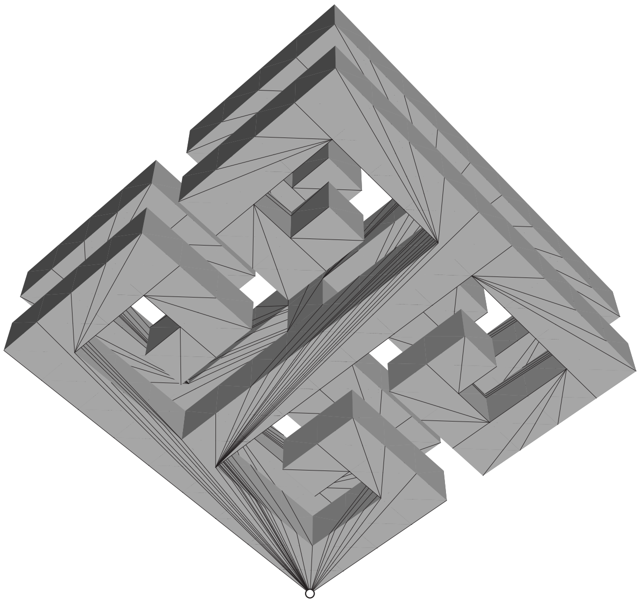

Figure 13: A simple orthogonal polyhedron of $n = 400$ vertices ($F = 796$) (Based on Fig. 7.10 in [O'R98, p.251]).

Finally we remark that there is no restriction that the polyhedron have genus zero. Fig. 14 shows an example of genus 6. Indeed, because our program is based entirely on a user-supplied two-dimensional complex, the input need not represent a surface embeddable in E^3 without self-intersections; nor need it be orientable.

Acknowledgement. We thank Veronica Morales for help generating the figures.

References

[AHPSV97] P. K. Agarwal, S. Har-Peled, M. Sharir, and K. R. Varadarajan. Approximate shortest paths on a convex polytope in three dimensions. *J. ACM*, 44:567–584, 1997.

[CH90a] J. Chen and Y. Han. Shortest paths on a polyhedron. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 360–369, 1990.

[CH90b] J. Chen and Y. Han. Shortest paths on a polyhedron, Part II: Storing shortest paths. Technical Report 161-90, Comput. Sci. Dept., Univ. Kentucky, Lexington, KY, February 1990.

[CH96] J. Chen and Y. Han. Shortest paths on a polyhedron. *Internat. J. Comput. Geom. Appl.*, 6:127–144, 1996.

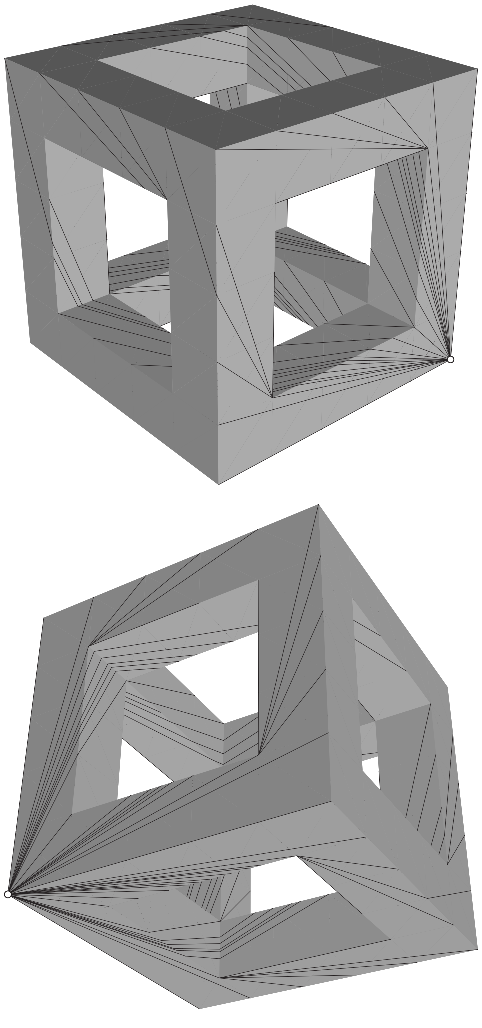

Figure 14: Two views of a wireframe cube: $n = 160$ ($F = 336$).

[HB97] D. Hearn and M. P. Baker. *Computer Graphics: C Version.* Prentice Hall, Second Edition, 1997.

[HP99] S. Har-Peled. Constructing approximate shortest path maps in three dimensions. *SIAM J. Comput.*, 28:1182–1197, 1999.

[Kap99] S. Kapoor. Efficient computation of geodesic shortest paths. In *Proc. 32nd Annu. ACM Sympos. Theory Comput.*, pages 770–779, 1999.

[LMS97] M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating weighted shortest paths on polyhedral surfaces. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 274–283, 1997.

[MMP87] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM J. Comput.*, 16:647–668, 1987.

[NF93] M. Namiki and K. Fukuda. Unfolding 3-dimensional convex polytopes: A package for Mathematica 1.2 or 2.0. Mathematica Notebook, Univ. of Tokyo, 1993.

[O'R98] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 2nd edition, 1998.

[O'R99] J. O'Rourke. Computational geometry column 35. *Internat. J. Comput. Geom. Appl.*, 9:513–515, 1999. Also in *SIGACT News*, 30(2):31-32 (1999), Issue 111.

[O'R00] J. O'Rourke. Folding and unfolding in computational geometry. In, *Lecture Notes Comput. Sci.*, Vol. 1763, J. Akiyama, M. Kano, M. Urabe, Eds. Springer-Verlag, Berlin, 2000, pp. 258–266.

[OSB85] J. O'Rourke, S. Suri, and H. Booth. Shortest paths on polyhedral surfaces. In *Proc. 2nd Sympos. Theoret. Aspects Comput. Sci.*, volume 182 of *Lecture Notes Comput. Sci.*, pages 243–254. Springer-Verlag, 1985.

[VA97] K. R. Varadarajan and P. Agarwal. Approximating shortest paths on an nonconvex polyhedron. In *Proc. 38th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 182–191, Miami Beach, Florida, 20-22 October 1997.

[Xu96] D. Xu. Shortest paths on polytopes. Undergraduate thesis, Smith College, 1996.