"Meshsweeper": From Closest Point to Hausdorff Distances Between Meshes

A. Guéziec† Multigen-Paradigm, Inc., a Computer Associates company

Abstract

We introduce a new algorithm for computing the distance from a point to an arbitrary polygonal mesh. Our algorithm uses a multi-resolution hierarchy of bounding volumes generated by geometric simplification. Our algorithm is dynamic, exploiting coherence between subsequent queries using a priority process (without caching the closest point), and achieving constant time queries in some cases. The method also applies to a mesh that deforms non-rigidly. Achieving from about 500 to several thousand queries per second for a complex polygonal shape on a PC, our method has applications both for interactive and photo-realistic graphics. In particular, we study in this paper the application to computing the Hausdorff distance between two polygonal shapes, with an arbitrary precision.

1. Introduction

Computing the Euclidean distance from a point to a complex polygonal shape is a fundamental problem in computer graphics. There are numerous applications both in interactive techniques (for collision prevention or tolerance verification) and in photo-realistic graphics (for accurate motion dynamics, 3-D path planning or self-intersection detection). Distance carries more information than occurrence or non-occurrence of collision, because it permits prediction, use of coherence, and dynamic path modification.

The existing options for computing the distance to a polygonal shape are as follows: A brute force computation visits all the polygons of the shape and reports the smallest distance. Since the closest point to a convex shape may be efficiently determined [1, 2], another option is to decompose an arbitrary shape into a collection of convex shapes. A Voronoi diagram of the shape may be computed. A Voronoi diagram reports for each point of space, the closest element on the shape [3]. While in general computing a Voronoi diagram of a scene containing tens of thousands of polygons is a very complex endeavor, Hoff *et al.* [4] recently introduced an elegant method, exploiting rasterization hardware, which generates a discretized version of the Voronoi diagram. Another option is to use a spatial data structure, such as an octree [5] to index the shape, and permit efficient querying for the closest point.

While the first two methods are generally impractical, the latter two become more complex when polygonal shapes move with respect to one another, e.g., requiring a separate octree for each shape. Further, the methods simply fail when the shapes move non-rigidly.

Another approach is to build a hierarchy on the polygonal mesh itself. While such hierarchies have been used to decompose planar polygonal curves for a long time [6, 7], the generalization to arbitrary polygonal meshes that do not have subdivision connectivity (that is, are not obtained by regular subdivision of a base mesh) was made possible only recently, with the introduction of Progressive Meshes [8] and similar hierarchies generated using geometric simplification. Using these hierarchies, multi-resolution modeling techniques can be applied to arbitrary meshes.

1.1. Main contributions

In this paper we present a new algorithm for computing the closest point on an arbitrary polygonal mesh.

1. Our algorithm uses a multi-resolution hierarchy of bounding volumes generated by geometric simplification. This hierarchy can be reused for display. As the mesh refines, improving closest point estimates appear to sweep the mesh until the true closest point is found (Section 4), hence the name of the algorithm.
2. The refinement process can be interrupted at any time, providing an approximation to the distance. Otherwise, the final result is the exact closest point and distance.

† gueziec@computer.org

3. The hierarchy is represented using a few arrays and tables, limiting memory usage and fragmentation.
4. Our algorithm is dynamic, exploiting coherence between subsequent queries (without caching the closest point; see Section 5). This dynamic process is responsible for much of the algorithm's speed.
5. Our dynamic algorithm also applies to a mesh that deforms. This is possible because our bounding-volume hierarchy, departing from commonly used ones [9, 10], undergoes little changes when a shape deforms smoothly.
6. The dynamic algorithm can be applied to measuring distances between meshes, by finely sampling one mesh and visiting the samples along continuous paths, while querying the other mesh for the closest point. In this way, we can approximate the expensive Hausdorff distance metric with an arbitrary precision at a reasonable cost (Section 6).

Other possible applications include robot motion planning and shape registration using the Iterative Closest Point method [11].

2. Previous Work

Our method straddles three closely-related areas where there is significant previous work.

2.1. Closest Point

Aside from Voronoi diagrams and methods applying to convex shapes that we mentioned earlier, k-D trees [5] have been used to compute the closest point on an arbitrary triangle mesh [12].

2.2. Separation Distance Between Objects and Collision Detection

To compute the separation distance between arbitrary meshes, a solution is to employ a hierarchy of bounding volumes, such as bounding spheres [9], Oriented Bounding Boxes (OBBs) [13, 14] or segments and rectangles swept by a sphere [15]. Starting with two bounding volume hierarchies, each corresponding to one object, the hierarchies are recursively traversed. Pairs of primitives are tested for determining whether the current estimate of minimum distance can be improved. Pruning strategies are employed to reduce the number of pairs tested. A similar framework has been commonly used for collision detection [16].

2.3. Selective Refinement of Progressive Meshes

Hoppe introduced Progressive Meshes [8], a continuous representation using a base mesh and a sequence of *vertex splits*, which are mesh refinement operations inverting the contraction of an edge. Algorithms and data structures for applying the vertex splits out of sequence were introduced in [17, 18, 19] and used for applying view-dependent levels of detail to meshes.

In [17, 18] and [20] the contraction of an edge of the mesh is represented with two children vertices (the endpoints of the edge) connected to a single parent vertex (the vertex resulting from the contraction) in a dependency graph, while in [21], a single parent-child relationship between vertices represents an edge contraction. Another structure is necessary to record the dependency between edge contractions, which is different from the above-described dependency. This dependency can be expressed using a Directed Acyclic Graph or DAG [19].

Aside from visualization, simplification-based hierarchies have been used for solving GIS queries such as horizon determination [22], point location, iso-contour extraction and visibility queries [20].

3. Distance to Polygonal Curves

The main contribution of this paper is the computation of the closest point on a 3-D polygonal mesh, developed in Section 4 and subsequent sections. To lay the foundation, we start by addressing the problem for a 2-D or 3-D polygonal curve in the present section. The author did not find a similar description in the literature, and therefore this section is most certainly original.

We consider a query point $\mathbf{p}$ and a polygonal curve S (two- or three-dimensional). We build a hierarchy of bounding elements on the polygonal shape with each level of the hierarchy covering the shape in its entirety. At the lowest resolution in the hierarchy, very few bounding regions cover the polygonal shape. A simple observation is that the closest point from $\mathbf{p}$ to the shape S must occur inside one of the bounding regions.

3.1. Bounding Elements

While various types of bounding elements could be accommodated (axis- or shape-aligned rectilinear boxes are a common choice), we have chosen as a bounding element the set of points in space whose distance to the segment is less than ϵ. As represented in Fig. 1, in the plane the elements are formed with segment-aligned rectangles capped with two half-circles. In this way a well-known algorithm for polygonal approximation described in [23] may be directly used to build the hierarchy: starting with a segment, insert the most distant point, associating the distance (or approximation error ϵ) to the segment, and splitting the segment in two; repeat until the error is sufficiently small.

Referring to Fig. 1, the distance d_0 from $\mathbf{p}$ to the portion of the shape S contained in a bounding region R is such that $d - \epsilon \leq d_0 \leq d + \epsilon$, where d is the distance to the segment defining R.

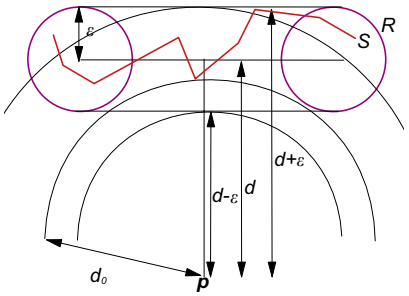


Figure 1: *The distance d_0 between* **p** *and the portion of S contained inside the bounding region R is such that $d-\epsilon \leq d_0 \leq d+\epsilon$.*

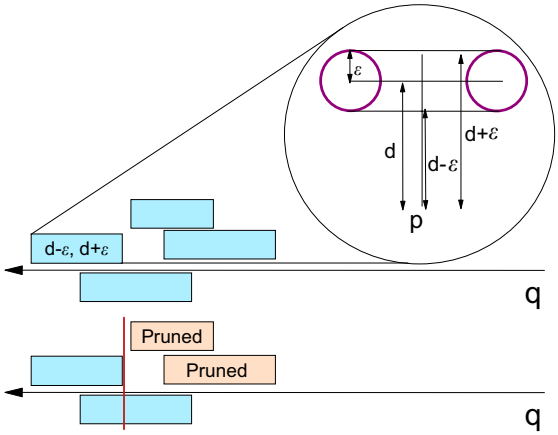


Figure 2: *A priority structure* **q** *represents each region of the hierarchy as an interval. An upper bound to the distance is used to prune the queue.*

3.2. Priority Process

After computing the distances to all regions in the coarsest level of the hierarchy, each region is associated with an interval of the type $[d-\epsilon, d+\epsilon]$ and indexed in a priority queue. When the key used in the queue to assign priorities is $d-\epsilon$, the interval listed in the front of the queue represents a minimum bound to the Euclidean distance between the query point and the shape. Fig. 2 shows this option, where the front of the queue is on the left side, and the back on the right side. The $d+\epsilon$ value corresponding to the leftmost interval provides an upper bound to the distance. As shown in Fig. 2, this upper bound can be used to prune all the elements of the queue whose key exceeds the bound. (When using $d+\epsilon$ as the key in the queue, the leftmost interval provides a better upper bound to the distance, but no lower bound, which is why we prefer $d-\epsilon$ as a key).

After pruning, the process loops on, removing the leftmost interval from the queue, splitting the corresponding region in two regions using the hierarchy described above, and reentering the two new regions in the queue. At each step of the process, the leftmost interval provides both an upper and lower bounds to the distance. If the difference between the upper and lower bounds is acceptably small, the process may be stopped, returning an approximate Euclidean distance to the shape. If the process continues until the upper and lower bounds are the same, then the exact closest point is returned as the closest point to the leftmost region in the queue.

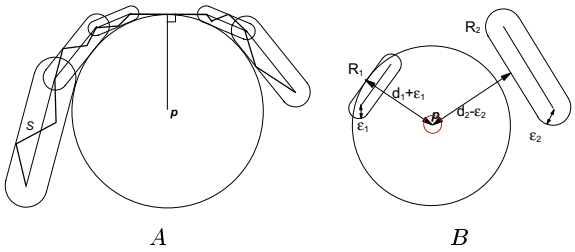


Figure 3: *A: Regions that must be split during a closest point query. B: Configuration permitting constant-time queries.* **p** *denotes the query point.*

Fig. 3-A illustrates what bounding regions must be split during a closest point query. We consider a sphere/circle centered at the query point **p** with a radius equal to the distance from **p** to the shape S. All of the bounding regions intersecting that circle must be split. (A point of S closer to **p** could potentially be located inside such a bounding region.)

3.3. Constant Time Queries

If at the end of the process the first and second intervals $[d_1-\epsilon_1, d_1+\epsilon_1]$ and $[d_2-\epsilon_2, d_2+\epsilon_2]$ have an empty intersection ($d_1+\epsilon_1 < d_2-\epsilon_2$), we can define a sphere centered at the query point, with a radius equal to $\frac{d_2-\epsilon_2-d_1-\epsilon_1}{2}$. Any query point inside that sphere will be closer to the first region than to any other region. We can thus resolve these queries in constant time. This situation is illustrated in Fig. 3-B.

4. In Three Dimensions: Mesh Sweeping Process

4.1. Bounding-Volume Hierarchies Obtained from Mesh Simplification

To generalize the method of Section 3 to polygonal surfaces, we turn ourselves to error-bounding hierarchies obtained from mesh simplification. Existing methods for error-bounding mesh simplification differ in the type of bounding volumes that are used: axis-aligned boxes in [24], oriented prisms in [25] and triangles swept by a sphere in [26]. (Other effective mesh simplification methods may also be used [27, 28], after adding the capability of building bounding volumes.)

We have opted for triangles swept by a sphere as in [26], providing a direct 3-D generalization of the bounding elements

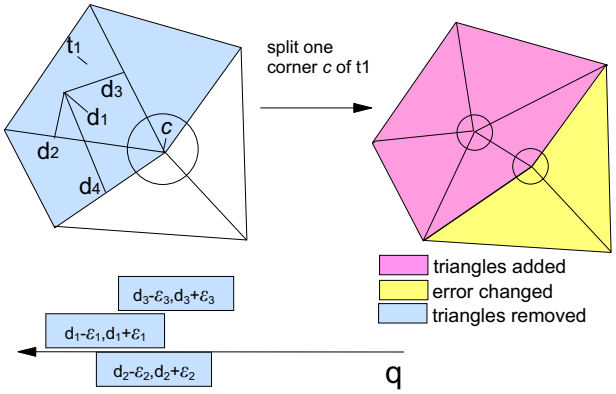


Figure 4: *Vertex split for mesh refinement, in relation to the priority structure. Each operation splits a corner of the highest-priority triangle t_1. Errors attached to vertices and defining the bounding volumes are updated. Some triangles must be removed from the queue, and others inserted as marked.*

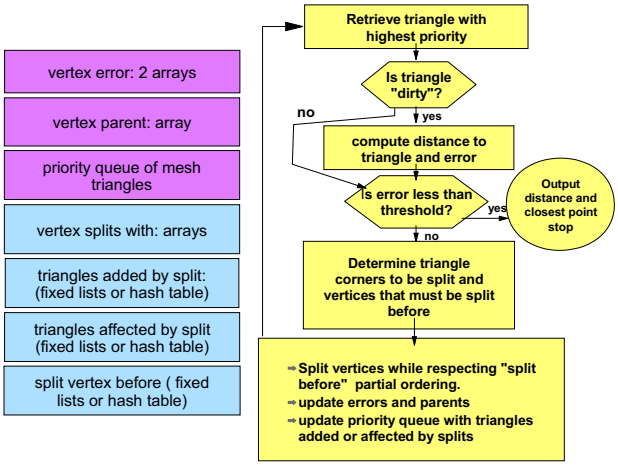


Figure 5: *Data Structures (three changing, four fixed) and algorithm used for mesh refinement. "Dirty" triangles are discussed in the next section.*

that we used for polygonal curves (Section 3): the same priority structures may be re-used with very few changes. Our choice was independently validated in [15], who report better results for distance queries when using volumes swept by a sphere instead of when using oriented and axis-aligned boxes.

4.2. Data Structures for Selective Refinement

Fig. 4 shows a mesh refinement operation (vertex split) in relation to our priority structure. The priority structure is extended from polygons to polyhedra by indexing triangles as opposed to segments. The ϵ value (Section 3) is chosen as the maximum of the sphere radii (or errors) at the triangle vertices. Nothing else is changed in the priority structure.

Referring to Section 2, recent work has shown that a hierarchy of vertices specifying valid edge contractions combined with a graph (specifically, DAG) representing dependencies between the edge contractions, may be used for selective mesh refinement [17, 18, 19, 21]. To specify which contractions must be performed before a given edge contraction is done, we query for vertices incident on the contracting edge in the simplified mesh, and determine whether such vertices were affected by a previous edge contraction.

To simplify the adaptive mesh refinement process and avoid memory fragmentation, we have developed methods for refining the mesh selectively, but *not* for coarsening it selectively. Periodically (after a number of closest-point queries) the mesh is substituted with the base mesh (coarsened all at once).

We found that this solution, departing significantly from the previous work listed above, works well in combination with the dynamic querying process. With this assumption, our data structure can consist exclusively of arrays, and tables or lists with a fixed content. (A priority queue may be implemented as a binary heap using arrays [29].)

Fig. 5 shows our data structures. Four fixed lists or hash tables record the contractions and dependencies, and are not modified by the process: "vertex splits with" lists the splits available for a vertex, "triangles added by split" lists the triangles that should be added (Fig. 4 when splitting, "triangles affected by split" lists the triangles whose bounding volumes change when splitting, and "split before" lists the vertices that must be split before a given vertex.

There is a one-to-one correspondence between an edge contraction and a vertex that is being removed. This observation is used for encoding the current mesh using three changing structures, represented using arrays: "vertex error" stores two copies of the error defining bounding volumes (error associated with the vertex, error associated with the parent before contracting the vertex with the parent); "vertex parent", indicates the status of being contracted or not with the parent; and "priority queue of triangles" was described previously in full detail.

The memory usage of the method should in fact be measured by the three changing structures, since the four fixed structures may be shared by several objects or threads. We note that the size of these structures is proportional to the size of the original mesh; however, the coefficients of proportionality are small: two floating point values (per vertex) for the error, one integer for the parent, two integer and one floating point value (per triangle) for the priority queue.

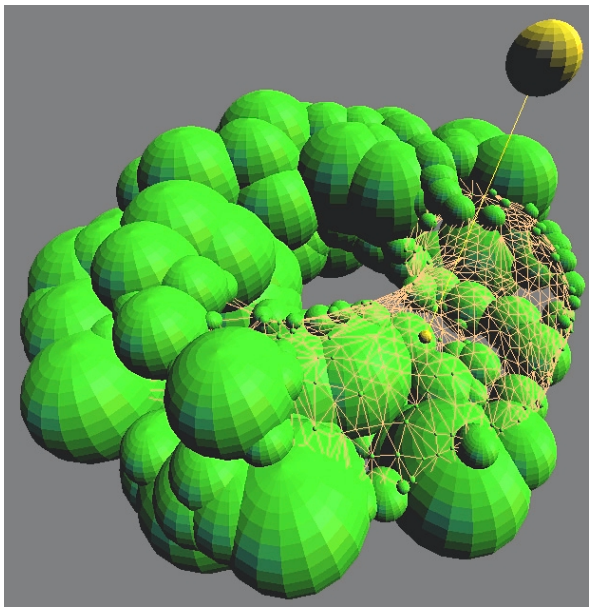


Figure 6: *Mesh Sweeping process: As a spherical particle falls towards a torus, the torus mesh is refined in the vicinity of the closest point to the particle, which is indicated by a small sphere, and is connected to the particle by a line. Note that we do* not *use a bounding sphere hierarchy. Instead, the bounding elements are triangles swept by spheres. We see here a snapshot of an interactive animation where we only represent the spheres at the vertices.*

4.3. "Mesh Sweeping" Process

The flow chart in Fig. 5 shows our algorithm. The highest-priority triangle is removed from the queue. The dynamic process (Section 5) may have marked it as "dirty", in case its distance to the query point is incorrect (but its error bound conservative). If so, the distance must be re-computed. (To compute the distance from a point to a triangle, we use the method of [30].) If not, we decide to split one of the triangle's vertices. Our fixed data structures encode which choices are available for splitting. The "split before" structure is used to determine additional vertices that should be split. In order to restore the "vertex errors" in the proper order, we convert the partial order defined by "split before" to a total order of the vertices split at this iteration. This is achieved using Topological Sort [29]. As vertices are split in order, triangles are removed and inserted in the queue as specified in the "triangles added" and triangles "affected" lists. (The removed triangles are a subset of the added triangles, when vertices are substituted using "vertex parent".)

As the mesh refines, improving closest point estimates "sweep" the mesh until the true closest point is determined. When this process finishes, the refined mesh may look as shown in Fig. 6.

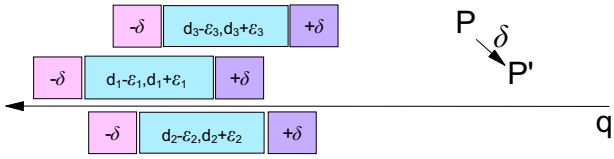


Figure 7: *Update of the priority structure when the query point is translated: the priorities are unchanged, but the intervals are expanded. The triangles are marked as "dirty".*

4.4. "n Closest Triangles" and "All Triangles Within a Distance d" Queries

Once the closest triangle to a query point is determined, the priority process may be used to keep querying for the next closest triangle, and the next, etc. As the mesh has already been refined by the query process in the vicinity of the closest point, the additional cost for obtaining the next closest triangle is very small.

5. Dynamic Algorithm

The coherence between successive queries in not expressed in the location of the closest point, but in the priority structure defined above. A small motion of the query point or mesh is accommodated by changing the sizes of the intervals stored in the queue, preferably without changing the priorities, and by marking mesh triangles as "dirty" as the distance to the query point becomes incorrect.

For a translation of the query point or rigid motion of the mesh, the sizes of the intervals are affected, but the bounding-volume hierarchy is not. For a non-rigid motion of the mesh, the bounding-volume hierarchy must be updated as well.

5.1. Small Motion of the Query Point

If the query point is translated, each interval is expanded on both sides by an amount equal to the magnitude δ of the translation (Fig. 7). In this way we have a correct bound on the distance, however conservative. The priorities are not affected, because each interval is expanded by the same amount. The intervals are expanded all at once, by-passing the priority queue mechanism, and the logarithmic costs involved.

5.2. Small Rigid Motion of the Mesh

Our preferred solution is to determine the maximum displacement among the mesh vertices, and to apply the same method as we did for a moving query point, substituting δ with the magnitude of the maximum displacement. In this way, the priorities are not affected.

5.3. Non-Rigid Motion of the Mesh

When comparing previous bounding volume hierarchies [9, 13, 14, 15] with an error-bounding mesh simplification [25, 24], mesh simplification computes bounding volumes locally, in a bottom-up fashion, instead of a top-down fashion.

For a smooth deformation, the motion of the mesh can be well approximated locally by a rigid motion. A rigid motion does not affect the width of our bounding volumes and they can thus be represented by the same epsilon values.

We need to maintain conservative bounds all the time; otherwise, the closest point could be missed. We introduce two methods for changing the bounds accordingly. (It may be that the changes are so small that we could safely ignore them, at least for some applications. We leave this investigation for future work.)

Both methods use the edge contraction specifications available as part of the data structures of Section 4, and apply them in the same sequence that was used to compute the bounding volumes originally.

The first and simpler method assumes that each vertex is linearly interpolated between key-frames. Before the deformation starts, the sphere radii defining the bounding volumes corresponding to each key frame are recorded, each in a separate array. (This is done by applying the edge contraction sequence for all frames and computing error bounds, as done in the mesh-simplification process, except that the sequence of edge contractions is known.) During the deformation, radii are linearly interpolated between key-frames. The bounding volumes are thus correct.

The second method makes no assumption about the motion. It requires a list of displacement magnitudes for each vertex. The magnitudes should be exaggerated, so as to produce conservative volumes for a few steps of the deformation. Edge contractions are applied in sequence. For each contraction, the maximum displacement magnitude is measured for incident vertices. The bounding volumes are inflated accordingly. Note that this operation does not require the expensive geometric computations involved in building the volumes in the first place [25, 24]. One alternative for deforming meshes is to specify time-varying bounding volumes, as in [31].

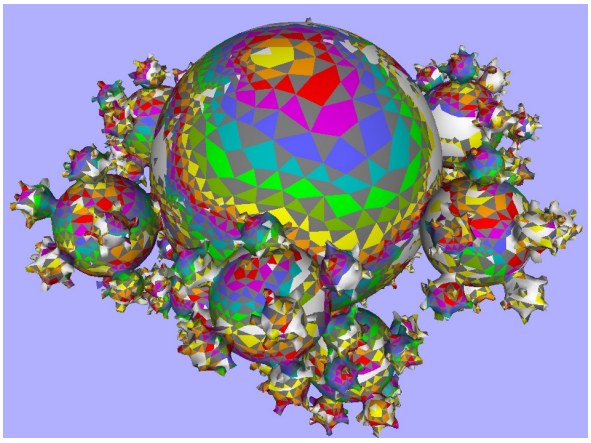


Figure 8: *Spiral traversal of a mesh to minimize the distance between successive samples and use the dynamic distance algorithm.*

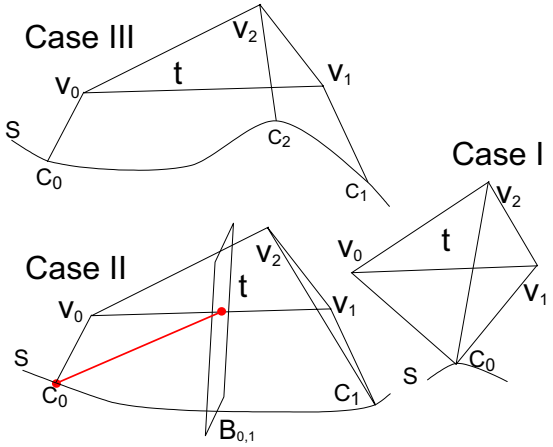


Figure 9: *Case by case analysis on how to bound the distance between a triangle t and a mesh S, knowing the closest point for each triangle vertex.*

6. Hausdorff Distance Between Meshes

With the recent progress in automated mesh simplification algorithms, multi-resolution mesh editing, or filtering, it is becoming increasingly useful to measure the deviation between complex meshes, and to determine the locations of maximum discrepancy.

The Hausdorff distance is a very useful measure of similarity between two shapes. However, it can be extremely expensive to compute for arbitrarily-complex shapes. For instance, referring to Fig. 10, we would like to compute as accurately as possible how a mesh of 29,000 triangles deviates from a simplified mesh containing 4,450 triangles. Sophisticated polygon reduction tools (e.g., [25, 24]) can bound the Hausdorff distance when producing the simplified mesh, but cannot compute it accurately.

To compute the Hausdorff distance, one must determine for each point (not only for each vertex) of the first shape, the distance to the closest point on the other shape, and report the largest distance measure. One must then do the same operation for the second shape. The Hausdorff distance is the larger of the two reported distances. The formal defini-

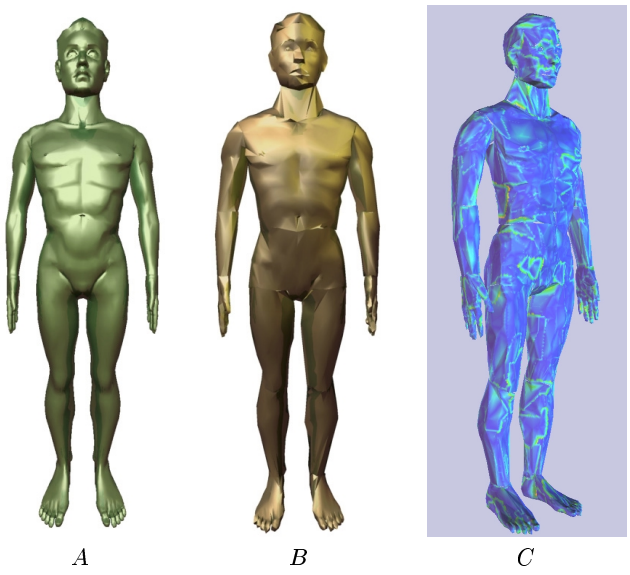
tion is as follows, where S, T designate two meshes, v, w points of the meshes, $H(S,T)$ the Hausdorff distance, and $d(v,S)$ the distance from a point to a mesh: $H(S,T) = \max(\max_{v \in T} d(v,S), \max_{w \in S} d(w,T))$.

Using our technique, this computation can be done with an arbitrary precision by sampling the surfaces and creating continuous paths visiting the samples. Such continuous paths are related to generalized triangle strips that have been used for geometric compression. We use essentially the same process as [32, 33] to create "peels" of a mesh by spiraling about a vertex or a boundary: triangles sharing an edge, and a vertex chosen as a *pivot* are traversed in order, creating a path of visited vertices. The pivot vertex is regularly changed as previously-visited vertices and traversed triangles are encountered. For details, see [32]. The topological distance separating vertices from the first pivot may be used to color faces and illustrate the peeling process, as shown in Fig. 8.

A first operation considers existing mesh vertices as samples for distance computations, after which the closest point to each vertex is known, but nothing is known for samples inside triangles. However, as illustrated in the three cases of Fig. 9 the up-to-three closest points c_0, c_1, c_2 at the vertices v_0, v_1, v_2 can be used to determine an upper bound for the distance at any sample inside the triangle t. Depending on how many vertices among c_0, c_1 and c_2 are different, zero, one, or two bisector planes of the c_i should be computed. An upper bound to the distance between a point of t and the closest vertex among c_0, c_1 and c_2 is obtained for one of the following: intersections of the bisector planes with edges of the triangle t (see Bisector $B_{0,1}$ in Fig. 9), or the intersection of the two bisector planes and the triangle t. This upper bound is most accurate as the distance increases, which is exactly what we need for the Hausdorff distance. We only need to refine the triangles where the distance is the highest, and the bound most accurate.

According to our experience, in most cases we can determine early in the refinement process which of the maximum distances $\max_{v \in T} d(v,S)$ and $\max_{w \in S} d(w,T)$ dominates. Thereafter, the method refines only the appropriate mesh. For instance, in the case of Fig. 10, the distance from the simplified mesh to the original mesh dominates, meaning that the Hausdorff distance is reached for one particular sample of the simplified mesh.

Using this method, after collecting about 211,000 samples of the simplified mesh of Fig. 10-B (and after 4 minutes, 24 seconds of computation) we can be as accurate as 0.04% of the diameter of the original shape in the Hausdorff distance computation. In Fig. 10-C we represent the distances at mesh samples using colors: blue for low values, red for high values (grayscale for the black-and-white paper copy of this article). Our best estimate for the Hausdorff distance is obtained for one of these samples. In comparison, a naive implementation takes 2 hours, 24 minutes for the same computation.



A B C

Figure 10: *Hausdorff Distance Between Meshes. A: Original mesh of 29,200 triangles. B: Simplified mesh of 4,450 triangles. C: Distance plot in pseudo colors (grayscale for the black-and-white copy of this article) between samples of B and A. Our best estimate of $H(A,B)$ is obtained for one of these samples.*

7. Summary and Future Work

We have introduced the new Meshsweeper method for computing the closest point to a query point on an arbitrary polygonal mesh undergoing non-rigid motion. We illustrated this method with an application to computing the Hausdorff distance between meshes with an arbitrary precision.

A detailed computational complexity analysis is left for future work. In the best case, the closest point is computed in constant time (Section 3). In the worst case the closest point is computed in linear time: when observing Fig. 3-A, a pathological case may be easily built; if the shape S is a polygonal approximation of a circle and if **p** is located at the center of S then all the bounding regions will be split. This case is very contrived: our experimental data (which will be reported in a forthcoming archival article) supports the constant time hypothesis better.

8. Acknowledgments

Thanks to Elizabeth Fox for proofreading the manuscript. My discussions with Stephan Gumhold were also helpful.

References

1. E. Gilbert, D. Johnson, and S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4:193–203, April 1988.

2. Ming Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, University of California, Berkeley, 1993.

3. F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Computing Survey*, 23:345–405, 1991.

4. K.E. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. In *SIGGRAPH'99 Proceedings*, pages 277–285, Los Angeles, August 1999.

5. H. Samet. *Applications of Spatial Data Structures*. Addison Wesley, 1989.

6. O. Guenther. *Efficient Structures for Geometric Data Management*. PhD thesis, University of California at Berkeley, 1987.

7. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, Reading, MA, 1990.

8. H. Hoppe. Progressive meshes. In *SIGGRAPH'96 Proceedings*, pages 99–108, New Orleans, August 1996. ACM.

9. S. Quinlan. Efficient distance computation between non-convex objects. In *Conference on Robotics and Automation*, pages 3324–3329. IEEE, 1994.

10. D. Johnson and E. Cohen. Bound coherence for minimum distance computations. In *Conference on Robotics and Automation*, pages 1843–1848. IEEE, 1999.

11. Paul Besl and Neil McKay. A method for registration of $3-D$ shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.

12. D.A. Simon. *Fast and Accurate Shape-Based Registration*. PhD thesis, Carnegie Mellon University, December 1996. CMU-RI-TR-96-45.

13. S. Gottschalk, M.C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *SIGGRAPH'96 Proceedings*, pages 171–180, New Orleans, August 1996. ACM.

14. D. Johnson and E. Cohen. A framework for efficient minimum distance computation. In *Conference on Robotics and Automation*, pages 3678–3683. IEEE, 1998.

15. E. Larsen, S. Gottschalk, M.C. Lin, and D. Manocha. Fast proximity queries with swept sphere volumes. Technical Report TR99-018, UNC Chapel Hill, 1999.

16. J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization*, 4(1):21–36, January-March 1998.

17. J.C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In Yagel and Nielson, editors, *Visualization 96*, pages 327–334. IEEE, October 1996.

18. H. Hoppe. View dependent refinement of progressive meshes. In *SIGGRAPH*, pages 189–198, Los Angeles, August 1997.

19. L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *Visualization 97*, pages 103–110. IEEE, 1997.

20. A. Maheshwahri, P. Morin, and J-R. Sack. Progressive TINs: Algorithms and applications. In *5th international workshop on advances in geographic information systems*, pages 24–29, 1997.

21. A. Guéziec, G. Taubin, F. Lazarus, and W. Horn. A framework for streaming geometry in vrml. *IEEE Computer Graphics and Applications*, 19(2):68–78, March-April 1999.

22. L. De Floriani and P. Magillo. Horizon computation on a hierarchical triangulated terrain model. *The Visual Computer*, 11:134–149, 1995.

23. U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1:244–256, 1972.

24. J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *SIGGRAPH*, pages 115–122, Orlando, July 1998. ACM.

25. C. Bajaj and D. Schikore. Error-bounded reduction of triangle meshes with multivariate data. In *Visual Data Exploration and Analysis III*, volume 2656, pages 34–45. SPIE, March 1996.

26. A. Guéziec. Locally toleranced polygonal surface simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):178–199, April-June 1999.

27. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH'97 Proceedings*, pages 209–216, Los Angeles, August 1997. ACM.

28. H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *Visualization'99*, pages 59–66, San Francisco, CA, oct 1999. IEEE.

29. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Mac Graw Hill, 1989.

30. D. Eberly. Magic software. http://www.magic-software.com/.

31. P.M. Hubbard. Interactive collision detection. In *Proceedings of the IEEE Symposium on Research Frontiers in Virtual Reality*, pages 24–31, October 1993.

32. M. Chow. Optimized geometry compression for real-time rendering. In *Visualization 97*, pages 415–421, Phoenix, AZ., October 1997. IEEE.

33. G. Taubin, W.P. Horn, F. Lazarus, and J. Rossignac. Geometry coding and VRML. *Proceedings of the IEEE*, 86(6):1228–1243, Jun 1998.