“Meshsweeper”: Dynamic Point-to-Polygonal-Mesh Distance and Applications

André Guéziec, Senior Member, IEEE

Abstract—We introduce a new algorithm for computing the distance from a point to an arbitrary polygonal mesh. Our algorithm uses a multiresolution hierarchy of bounding volumes generated by geometric simplification. Our algorithm is dynamic, exploiting coherence between subsequent queries using a priority process and achieving constant time queries in some cases. It can be applied to meshes that transform rigidly or deform nonrigidly. We illustrate our algorithm with a simulation of particle dynamics and collisions with a deformable mesh, the computation of distance maps and offset surfaces, the computation of an approximation to the expensive Hausdorff distance between two shapes, and the detection of self-intersections. We also report comparison results between our algorithm and an alternative algorithm using an octree, upon which our method permits an order-of-magnitude speed-up.

Index Terms—Triangular mesh, closest point, multiresolution hierarchy, priority process, dynamic queries.

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, 3D path planning, or self-intersection detection [1]). Distance carries more information than occurrence or nonoccurrence of collision because it permits prediction, use of coherence [2], and dynamic path modification.

The existing options for computing the distance to a polygonal shape are: brute force computation, decomposition into convex shapes, Voronoi diagram, and spatial data structure. 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 [3], [4], 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 [5]. While, in general, computing a Voronoi diagram of a scene containing tens of thousands of polygons is a very complex endeavor, Hoff et al. [6] 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 [7], [8], 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 nonrigidly.

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 [9], [7], the generalization to arbitrary polygonal meshes that do not have subdivision connectivity was made possible only recently with the introduction of Progressive Meshes [10] and similar hierarchies generated using geometric simplification. Using these hierarchies, multiresolution modeling techniques can be applied to arbitrary meshes (see, e.g., [11]).

1.1 Main Contributions

In this paper, we present a new algorithm for computing the distance and closest point from a query point to an arbitrary polygonal mesh.

1. Our algorithm uses a multiresolution hierarchy of bounding volumes generated by geometric simplification of the polygonal mesh. This hierarchy can be reused for view-dependent resolution display, unlike the bounding-volumes used in traditional collision detection.
2. As the mesh refines, increasingly accurate closest point estimates appear to sweep the mesh until the true closest point is found (Section 4.3), hence the name of the algorithm. The refinement process can be interrupted at any time, providing an approximation to the distance with both an upper and lower bound.
3. The hierarchy is represented using a few arrays and tables, limiting memory usage and fragmentation.
4. As developed in Section 5, our algorithm is dynamic, exploiting coherence between subsequent queries without explicit caching of the closest point or points. (In fact, closest-point caching is performed implicitly as part of the priority process described in

1. Are not obtained by regular subdivision of a base mesh.

This work was done at Multigen-Paradigm, Inc., a Computer Associates Company, 550 S. Winchester Blvd., Suite 500, San Jose, CA 95128.
E-mail: gueziec@computer.org.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 112228.
Section 3.) This dynamic behavior is responsible for much of the algorithm’s speed. Experimental results reported in Section 10 indicate that our dynamic algorithm is one order of magnitude faster than a method using an octree.

5. Our dynamic method also applies to a mesh that deforms, as illustrated with a simulation in Section 7. This is possible because our bounding-volume hierarchy, departing from previously used ones [12], [13], [14], is simply represented by a width around the mesh. It moves with the mesh as the mesh deforms. Instead, a more traditional top-down partition of the mesh in sets of triangles belonging to various bounding elements must be readjusted as the mesh deforms.

In this paper, we also study in detail a few applications of our new algorithm. A direct application of our algorithm for tracking the distance to a moving and deforming shape is implementation of particle systems collisions and dynamics (see Section 7). Our algorithm also accelerates the computation of distance maps and offset surfaces (see Section 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 8). As discussed in Section 4, our method can return the \( n \) closest triangles from a point or all the triangles within a distance \( d \). This can be applied to the detection of self-intersections (Section 9). Other possible applications include robot motion planning and shape registration using the Iterative Closest Point method [15]. While the originality of this paper resides primarily in Section 4 and in the applications that follow, the author is unaware of a previous description similar to Section 3, which may be original.

2 PREVIOUS WORK

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

2.1 Closest Point in a Set of Points

Aside from Voronoi diagrams and methods applying to convex shapes that we mentioned earlier, \( k \)-D trees (see [7, pp. 66-80]) have been used to compute the closest vertex on an arbitrary triangle mesh (see, for instance, [16]). A \( k \)-D tree (\( k = 3 \) in our case) is a binary search tree; at each level of the tree, a different coordinate is tested when determining which hyperplane separates the left and right subtree. A typical closest-vertex query will first obtain an estimate of the closest vertex by traversing the tree from the root down to a leaf. Then, a sphere is extended from the query point to that leaf vertex and the \( k \)-D tree is used for selecting all vertices inside that sphere. Finally, the closest vertex among those selected is retained.

The main issue with the \( k \)-D tree method is that the closest vertex and the closest point on a polygonal mesh may be very distant from one another, as illustrated in Fig. 2; the closest point cannot be easily obtained using this method.

2.2 Separation Distance between Objects and Collision Detection

To compute the separation distance between arbitrary meshes, a hierarchy of bounding volumes, such as
bounding spheres [12], Oriented Bounding Boxes (OBBs) [18], or segments and rectangles swept by a sphere (as in the PQP method [13]) can be employed. Starting with two bounding volume hierarchies, each corresponding to one object, the hierarchies are recursively traversed. Pairs of bounding volumes are tested for determining whether the current estimate of minimum distance can be improved; otherwise, they are pruned from the search. If so, one efficient heuristic is to split the largest bounding-volume of the pair [13]. A closely related framework has been commonly used for collision detection (see, for instance, [17], [19]).

Among these methods for computing distances between objects, the PQP method [13] is the most closely related to the present method. PQP uses a queue as well to prioritize bounding-volume pairs. One major difference with Meshsweeper is that our bounding volumes are attached to the polygonal mesh and move with it as the mesh moves. One of our bounding volumes is simply represented by a width around the mesh. Instead, in PQP and the above methods, the hierarchy is kept separately from the mesh, operating a partition of the mesh polygons/triangles in various bounding elements. In the case of a nonrigid motion, all the bounding elements should be readjusted and, perhaps more importantly, the polygon partition may become obsolete. Instead, Meshsweeper performs well with a nonrigid motion of the mesh, as discussed in Section 5.3. Another difference with PQP and other methods [12], [18] is that Meshsweeper maintains both a lower and an upper bound to the distance at all times, as opposed to a lower bound only. The upper bound is just as important as the lower bound, allowing additional pruning and, also, returning a conservative distance estimate given any time constraint (the more available the time, the more accurate the result).

Although it may seem that the problem of computing the separation distance between objects is a generalization of the point-to-object distance problem, we argue that it is not the case for efficiency reasons. The above-cited algorithms, when adapted for point-to-object distances, would not provide the required efficiency, especially in the presence of deformations. Also, the point-to-object distance problem has by itself an impressive number of applications. We attempt to cover some of these in Sections 6 through 9.

2. First used [17] for collision detection.
3.2 Simplifying $S$ for Defining the Bounding Elements

With this definition of bounding elements, a well-known algorithm for polygonal approximation described in [26], [27] may be directly used to build the hierarchy: Starting with a segment, link the first and last vertices of $S$, insert the most distant vertex of $S$, associating the distance (or approximation error $\epsilon$) to the segment, and splitting the segment in two; repeat until the error is sufficiently small. Note that the portion of the polygonal curve comprised between the two endpoints of a segment of the hierarchy is connected.

Referring to Fig. 4, the distance $d_0$ from $p$ 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 $R$. (Examining Fig. 4 reveals that both extreme cases $d_0 = d - \epsilon$ and $d_0 = d + \epsilon$ are possible since $S$ is connected inside $R$ between the two segment endpoints. When $p$ is closer to a point inside the segment, $d - \epsilon \leq d_0 \leq d$ holds, completing the proof.)

3.3 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. 5 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. 5, 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 into 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 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.

Since both an upper and lower bound to the distance are available at each step, the distance can be computed to any precision. However, the location of the closest point is undefined unless the highest available precision is used for the distance: This is a general fact regarding the distance to a shape, that is not specific to our Meshsweeper algorithm.

3.4 Bounding Regions Split for an Exact Query

Fig. 6 illustrates what bounding regions must be split during an exact 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.)

3.5 Constant Time Queries

If, at the end of the priority 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 $\frac{d_2 - d_1 - \epsilon_1}{2}$. 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 (assuming the same level of accuracy $\epsilon_1$, which may be zero, is requested). This situation is illustrated in Fig. 7. This is an exact solution for the intuitive idea of “caching the closest point.”
4 PART II: 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 [28], oriented prisms in [29], and triangles swept by a sphere in [30]. (Other effective mesh simplification methods may also be used [31], [32], after adding the capability of building bounding volumes.)

4.1.1 Our Choice of Bounding Volume

We have opted for triangles swept by a sphere, as in [30], providing a direct 3D generalization of the bounding elements that we used for polygonal curves (Section 3): The same priority structures may be reused with very few changes. Our choice was independently validated in [13], who report better results for distance queries when using volumes swept by a sphere instead of when using oriented and axis-aligned boxes. Our bounding volume is shown in Fig. 8.

To compute the bounding-volume hierarchy, we use the method of [30], which we briefly summarize here: Valid edge contractions (that do not change the topology) are performed in order of increasing error; the error induced by edge contraction defines the width of the bounding volumes shown in Fig. 8 and is stored on a per-vertex basis. To compute the error, a mapping (common subdivided tesselation) is computed between before and after-contraction configurations; deviations at the vertices of the subdivided tesselations are then measured. These deviations are then compounded with the accumulated error from earlier contractions. Other mesh simplification methods could be used instead [29], [33], [28], [31], [32] when adapted to output our bounding volumes.

4.1.2 Priority Structure for Mesh Refinement

Fig. 9 shows a mesh refinement operation (vertex split) in relation to the priority structure. The priority structure is extended from polygons (Section 3) to polyhedra by indexing triangles as opposed to segments. The interval’s radius \( \epsilon \) is chosen as the maximum of the sphere radii (or errors) at the triangle vertices. Nothing else is changed in the priority structure.

4.1.3 Strategy for Refinement

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 the query point has covered a certain distance), 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. (A solution involving selective coarsening may be preferred for sustained-query-rate applications such as haptics, which we have not investigated.) With this assumption, our data structures can consist exclusively of arrays and tables or lists with a fixed content as discussed below. (A priority queue may be implemented as a binary heap using arrays [34].)

4.2 Encoding the Mesh and Bounding Volume Refinements

Referring to Fig. 10a, we give the edge contraction a direction: \( v_1 \) merges with \( v_2 \). We encode this by specifying that the “vertex parent” of \( v_1 \) is \( v_2 \). Before contraction, the parent of \( v_1 \) is simply \( v_1 \). After contraction (Fig. 10b), for
each triangle corner that used to be $v_1$, parent($v_1$) is used instead, thus closing the gap created by removing the triangles incident to both $v_1$ and $v_2$. The mechanisms involved are explained in detail in [24].

Fig. 10c and Fig. 10d illustrate what happens with vertex errors: $v_2$ has two associated errors: one before and one after contraction; $v_2$’s error before contraction with $v_1$ is stored as parent error($v_1$).

We can thus encode an edge contraction and a vertex split efficiently. The next issue is to specify in which order the splits should occur. This information can be recorded when the mesh is being simplified. When contracting an edge, we query for vertices incident on the contracting edge in the simplified mesh. We then determine whether such vertices were affected by a previous edge contraction, in which case, the corresponding splits must occur last. This information is best recorded in a DAG. However, owing to our refinement strategy, we only need to record the precedences for splitting: this can be done using one linked list or array per vertex, which brings up the issue of data structures.

4.2.1 Data Structures
Fig. 11 shows our data structures. Four fixed lists or hash tables record the contractions and dependencies and are not modified during the refinement process: vertex splits with lists all the vertices that have merged with a given vertex (possible splits), triangles added by split lists the triangles that should be inserted (Fig. 9) 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.

4.2.2 Memory Usage
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. (See Section 7 for some detail of a C++ implementation.) We implement vertex references inside polygons as integers. 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.

4.3 Algorithm
The flow chart in Fig. 11 shows our algorithm. The highest-priority triangle is removed from the queue. The dynamic process (Section 5) may have marked it as dirty, which means that its distance to the query point is inaccurate (but its error bound is conservative). If so, the distance must be recomputed. (To compute the distance from a point to a triangle, we use the method of [35].) If not, we decide to split one of the triangle’s vertices. Our fixed data structures encode the choices that 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 [34]. As vertices are split in order, triangles are removed and inserted in the queue as specified in the “triangles added” and “triangles affected” lists.
4.3.1 Mesh “Sweeping”
As the mesh refines, increasingly accurate 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. 12.

4.3.2 “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 continue 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 (this is confirmed in Table 1 of Section 10).

5 Dynamic Algorithm
We want to exploit the spatial coherence between successive queries such that if the query point or mesh move slightly, the amount of additional computation to determine the new location of the closest point will be minimal.

However, even a very slight perturbation of the query point or mesh can cause the closest point to travel large distances (topological as well as Euclidean) on the mesh. A careful inspection of Fig. 1 can confirm this. The search space for the new location of closest point is thus global to the mesh.

The coherence between successive queries was previously exploited by keeping track of the closest feature-pair (see, for instance, [36]). Since the closest point is subjected to abrupt changes, the cached information may be useless; or worse, it may trap the algorithm in a local state and prevent the discovery of the correct location of the closest point.

In the present approach, we exploit the coherence between successive queries using the priority structure defined in Section 3. The priority structure holds a small percentage of the mesh representation while operating on a global scale on the mesh. 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.

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 $\delta$ of the translation (Fig. 13). 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, bypassing 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 $\delta$ with the magnitude of the maximum displacement. In this way, the priorities are not affected.

5.3 Nonrigid Motion of the Mesh
When comparing previous bounding volume hierarchies [12], [17], [18], [13] with an error-bounding mesh simplification [29], [28], mesh simplification computes bounding volumes locally, in a bottom-up fashion, instead of a top-down fashion (see Fig. 14).

---

Fig. 12. Mesh sweeping process: As a spherical particle falls towards a torus (a), 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 (b). The partially refined mesh is shown together with the bounding elements in (b) and without in (c). Note that we do not use a bounding sphere hierarchy. Instead, the bounding elements are triangles swept by spheres. Here, we show individual frames of an interactive animation where we only represent the spheres at the vertices.

Fig. 13. 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.”
used to compute the bounding volumes originally. Section 4.2 and apply them in the same sequence that was specifications available as part of the data structures of guarantee an accurate distance and closest point computa-

If the goal is to maintain conservative bounds and thus requires unnecessary refinements), underestimation may result in an error in the final distance estimate, bounded by the amount of the underestimation.

In this paper, we argue that, for many applications, the adverse effect of the potential bound underestimation will be minimal with our method. We refer to Fig. 12 for an illustration of the refinement process: The exact distance and closest point are determined on refined portions of the surfaces, where the bounding volumes are very thin. These thin volumes will change very little under deformation. The large volumes, bounding large portions of the surface, could potentially vary significantly during deformation, but get either refined or pruned by the priority process. We report an experiment in Section 7.2 that supports our conjecture. We leave a comprehensive analysis and additional experiments for future work.

5.3.2 Changing the Bounding Volumes
If the goal is to maintain conservative bounds and thus guarantee an accurate distance and closest point computation in all cases, we introduce two methods for changing the bounds accordingly. Both methods use the edge contraction specifications available as part of the data structures of Section 4.2 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 keyframes. Before the deformation starts, the sphere radii defining the bounding volumes corresponding to each frame are recorded, each in a separate array. During the deformation, radii are linearly interpolated between keyframes. 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 relative 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 [29], [28].

One alternative for deforming meshes is to specify time-varying bounding volumes, as in [37]. This section completes the description of the algorithm. We now turn to applications.

6 Distance Maps and Offset Surfaces
One application of our method is to efficiently compute 3D distance maps surrounding arbitrary models. Distance maps may be used for motion planning and obstacle avoidance. With our method, models can be composed of several meshes and do not need to be solid. This is exemplified by a model of the Bart character, shown in Fig. 15a and Fig. 15b. The Bart character is composed of 64 separate meshes totaling 9,000 triangles; most of these meshes have a boundary and thus cannot define a solid object.

The method works by embedding the shape inside a regular volume of space and, for each voxel composing the volume, determining the distance to the shape using the Meshsweeper algorithm. As explained in Section 5, the method is most effective when query points are visited along continuous paths. Creating a continuous path visiting all voxels of a 3D volume is easily performed as follows: The volume is processed plane by plane; in each plane, consecutive scan-lines are traversed in opposite directions; the first scan-line in the following plane is chosen to be adjacent to the last scan-line of the current plane. (Perhaps a more efficient alternative to scan-lines would be to follow space-filling curves with a better spatial coherence, such as a Hilbert space-filling curve [38]. See also [39] for a recent reference discussing space-filling curves.)

For this application, we can compute the exact closest point and distance or we can perform approximate distance queries as well. We can compute a distance estimate as close as desired to the true value.

Using our method, it takes 2 minutes and 47 seconds on a PC (550 MHz Dell) to compute a 64³ distance map surrounding the Bart model. When using a brute force method, this computation takes 2 hours, 26 minutes.

6.1 Comparison with Existing Scan-Conversion Methods
We believe that this algorithm is superior to existing scan-conversion methods, provided that they can handle non-solid and intersecting objects, having no inside or outside (see, for instance, [40]). A discrete Euclidean transform [41] would have to follow scan conversion, approximating all distances with integer values and, thus, introducing aliasing. Finally, a filtering step would be required, introducing further errors. Our exact (or approximate) distance computation avoids this type of aliasing and

3. Which are discrete volumes, each voxel reporting the distance to the closest point on a scene.
permits the extraction of high quality offset surfaces without requiring very high resolution distance maps, trading spatial accuracy with accuracy in the sample values.

See Figs. 15 and 16. Our proposed scan conversion occurs in a natural voxel ordering, avoiding random access to the volume and allowing for a particularly simple implementation (for instance, using the Meshsweeper object described in Section 7).

6.2 Offset Surfaces

We may also use a distance map to compute offset surfaces, which are connected solids surrounding the original model, each point of the surface being at an equal distance from the model. We approximate the offset surfaces with isosurfaces corresponding to a given distance value.

Offset surfaces may be used for interesting effects: An isosurface of the $64^3$ distance map was used in Fig. 15c to simulate the effect of the Bart character being dipped into honey. In Fig. 16, three offset-surfaces rendered as transparent surfaces were combined with the original model to give the effect of the Bart character being transformed into an icicle.

If the object is solid, we can determine whether the query point is inside or outside the surface by comparing the orientation of the line segment joining the query point to the closest point (an exact closest point is required) with the orientation of the normal to the solid’s surface at that point (the normal of the closest triangle). This information can be used to define negative distance values inside the object. The resulting distance map also produces a rasterization of the object. An isosurface at a distance near zero forms an approximation of the original model.

7 Particle-System Dynamics and Collisions

We want to exploit our method to simulate the dynamics of a collection of particles falling on a deforming mesh according to the law of gravity. We implemented a system of spherical particles as described in [42]: Each particle has a six-vector state (position, velocity), a radius, and mass. Dynamics are implemented by a succession of Euler integrations over a $t$ time step.

The behaviors described in Sections 3, 4, and 5 may be implemented inside a Java or C++ “Meshsweeper” class containing a pointer to the original mesh, the various arrays and priority queue of Section 4.2, and the position of the previous query point as private data. Public methods include various instances of the closest point query (including “all triangles closer than $d$” queries, etc.) as well as methods to update the bounding volumes and priority queue in case the mesh moves. A Meshsweeper object may also return the mesh normal at the closest point or build a copy of the current mesh showing partly refined bounding volumes. (Fig. 12b was produced in this way.) To best exploit coherence between successive time steps, each particle is associated with its own Meshsweeper object. However, there is only one copy of the mesh for the entire simulation. It is also possible for all particles to share a Meshsweeper object, in which case, the coherence between the positions of the different particles may be used.
7.1 Using Distance Queries to Detect Collision

Meshsweeper::collDetect(vector<float>&state, float r) is a public method that uses the particle’s state (velocity v, position p), and radius r to determine if the particle might collide with the mesh during the next time step. Position, velocity, and radius determine a cylinder of length \( l = |v| \cdot \delta t \).

The simple idea of checking whether the closest mesh point is inside that cylinder (in the direction of the velocity) does not work because the particle may collide with a more distant part of the surface, still inside the cylinder.

Referring to Fig. 18, our approach uses the idea of recursive approximations of the cylinder using spheres, similar in spirit to Quinlan’s work [12]. We implement up to three levels of recursion. In the first level (L1 in Fig. 18a), a distance query to the mesh is performed from the cylinder midpoint. If the distance is less than \( \sqrt{l^2 + r^2} \) (a “positive” distance query), then a collision is possible and we recurse to the second level (L2). Fig. 18a shows this situation, where the closest point is marked with an x. In the second level, the cylinder is divided into two equal parts and distance queries are repeated on both parts. When the last level is reached, a positive distance query is interpreted as a collision. The last level is L3 unless \( r \) is smaller than the residual length. If no collision is detected, \texttt{collDetect()} still returns the distance to the closest point for further use.

In case of collision, a quadratic equation is solved to determine the time of impact: If \( p, v \) is the particle’s state, \( r \) its radius, and \( q \) the closest point, we solve for the smaller \( t \) such that \( |p - q + tv| = r \). If there is no real solution, (x in Fig. 18c), we minimize \( |p - q + tv| \).

The velocity after collision is computed by flipping \( v \) about the mesh normal at the closest point \( q \) and applying a (mass-dependent) restitution factor to the resulting velocity. The particle then travels with its old velocity for a duration of \( \delta t \) and its new velocity for \( \delta t - t \). The gravity is thus ignored for the \( \delta t - t \) time span.

We detect up to two collisions per time step: The above process is repeated for the cylinder corresponding to a \( \delta t - t \) time span. Meanwhile, a classical Euler step is used for the other noncolliding particles. In this way, each frame corresponds to a time step (\( \delta t \)), whether there are collisions or not. The maximum number of distance queries per particle for a given time step is thus 14.

7.1.1 Reducing the Number of Distance Queries

For each particle, in between calls of \texttt{collDetect()}, we maintain a conservative estimate of the distance to the mesh by subtracting each successive displacement \( |v| \cdot \delta t \) to the distance returned by the previous query. (The mesh may have moved as well. We may use an estimate of the mesh displacement or, in the likely case where particles move faster than the mesh, simply double the subtracted amount.) Only when the distance estimate is less than a given (user-specified) multiple of \( r \) do we call \texttt{collDetect()} again.

Snapshots of our simulation are shown in Fig. 17. The mesh undergoes the following deformation: While the shape rotates about the z axis, the z coordinate of each vertex is scaled by the factor \( 1 + 0.03 \sin(\sqrt{x^2 + y^2 + \tau}) \), where \( \tau \) is the simulation clock time.

7.2 Effects of Ignoring the Bounding Volume Changes when Deforming

For 4,040 closest-point queries, we compared an exact (brute force) computation with the Meshsweeper computation while the mesh deformed and the bounding volumes widths were not changed. For each mesh vertex, the

---

Fig. 17. Snapshots of an interactive particle-dynamics animation. Spherical particles fall on the Sphere Flake mesh that is subjected to a deformation. Each sphere is connected to its closest point on the mesh (represented by a smaller sphere) with a line segment. The spheres are colored red for a few frames following a collision. Distance-tracking still works, although the mesh moves nonrigidly.

Fig. 18. Using distance queries to detect collision. We use the idea of approximating the particle’s cylinder-trajectory with spheres with up to three levels of recursion.
amplitude of the deformation was measured after subtracting the rotation. With a unit shape diameter, the average amplitude was 0.19. We then measured the average distance between the exact closest point and the Meshsweeper closest point: 0.00036, and the average error in the distance: 0.2 percent. These numbers are so small that they would hardly affect interactive collision detection and various visual applications. The magnitude of the deformation may be judged by examining the images of Fig. 17b and Fig. 17c, taken with the same camera perspective.

8 Hausdorff Distance between Meshes

With the recent progress in automated mesh simplification algorithms, multisolution mesh editing, and mesh filtering, it is becoming increasingly useful to measure the deviation between meshes and to determine the locations of maximum discrepancy in order to quantify the accuracy of a particular representation.

The Hausdorff distance is a very useful measurement of similarity between two shapes. However, it can be extremely expensive to compute for arbitrarily-complex shapes. For instance, referring to Fig. 21, 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., [29], [28]) 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 definition 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:

\[
H(S, T) = \max \left( \max_{v \in T} d(v, S), \max_{w \in S} d(w, T) \right).
\]

Using our technique, this computation can be done with an arbitrary precision (but not exactly) 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 [43], [44] 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 [43]. In Fig. 19, the topological distance separating vertices from the first pivot is used to color faces and illustrate the peeling process.

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. 20, 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 to the distance for any sample inside the triangle \( t \). Depending on how many vertices among \( c_0, c_1, c_2 \) are different, zero (Case I in Fig. 20), one (Case II), or two bisector planes of the \( c_i \) (Case III) should be determined. An upper bound to the distance between any point of \( t \) and the closest vertex among \( c_0, c_1, c_2 \) is obtained for one of the following: intersections of the bisector planes with edges of the triangle \( t \) (see Bisector \( B_{01} \) in Fig. 20) or the intersection of two bisector planes and the triangle \( t \).

This upper bound is more accurate for large distances, which is exactly what we need for the Hausdorff distance: Since we are looking for the largest distance, we only refine the triangles where the current distance estimate is already large and the bound is 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. 21, 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. This is why the distances are only shown on the simplified mesh in Fig. 21c.
Using this method, after collecting about 211,000 samples of the simplified mesh of Fig. 21b (and after 4 minutes, 24 seconds of computation), we can be as accurate as 0.04 percent of the diameter of the original shape in the Hausdorff distance computation. In Fig. 21c, we represent the distances from each mesh sample to the other mesh using colors (blue for low values, red for high values). Our best estimate for the Hausdorff distance is obtained for one of these samples. In comparison, a brute force implementation takes 2 hours, 24 minutes for the same computation.

9 DETECTING SELF-INTERSECTIONS

Our distance algorithm may also be used to speed up the detection of all intersecting pairs of triangles in a mesh. Two triangles are considered to intersect if the Euclidean distance between triangles is zero, except for adjacent triangles. A mesh with at least one pair of intersecting triangles has a self-intersection.

Considering a vertex \( v \) of the mesh, we want to determine whether there can be an intersecting pair of triangles closer to \( v \) than to any other vertex of the mesh. One triangle of the pair will be incident on \( v \). Once all vertices of the mesh have been tested in this fashion, all possible intersecting pairs of triangles will be reported.

We consider the list of edges incident on \( v \) and compute the length \( a \) of the longest incident edge. We then define a sphere centered on \( v \) with a radius \( r = a/\sqrt{3} \). (The \( \sqrt{3} \) factor corresponds to the worst case of an equilateral triangle incident on \( v \).) Our closest point algorithm is then used to list all triangles that are closer than \( r \) from \( v \). This list is split into a first list of triangles incident on \( v \) and a second list of triangles not belonging to the first list. Triangles of the first list are tested against triangles of the second list for possible intersections. During this operation, we keep a hashtable that lists all pairs of triangles tested so far. To determine if two particular triangles intersect, we use the method described in [45].

Fig. 21. Hausdorff distance between meshes. (a) Original mesh of 29,200 triangles. (b) Simplified mesh of 4,450 triangles. (c) Distance plot in pseudo colors between samples of (b) and (a). Our best estimate of \( H(a, b) \) is obtained for one of these samples.

10 PERFORMANCE

In this section, we gather experimental results on the performance of our method and compare it with two other methods. The first method we use for the comparison is a brute force method that determines the closest point on a triangular mesh by computing the distance to all the triangles of the mesh. This method sets the “gold-standard” for the accuracy in distance computation: Other methods are expected to provide matching results.

The second method uses an octree. A volume of space enclosing all possible query points is decomposed in a hierarchical fashion using the octree. For each query point, the method determines the smallest nonempty octree cell that contains this point. Up to 26 neighboring cells are also retrieved and the distance from the query point to the closest cell corner is recorded for each neighboring cell. Then, the method visits all triangles that are referenced in the octree as intersecting the cells, subjected to culling as follows: While triangles are being processed, the closest triangle found so far sets a bound on the minimum distance, allowing us to cull triangles (and entire cells) that are outside a sphere centered on the query point and extending to the closest triangle. For faster culling, spheres enclosing the triangles are used. While the closest point moves, a bound on the minimum distance is interactively maintained.

Fig. 23a shows a model of 8,000 triangles representing a Klein bottle, which is a nonorientable mesh with a self-intersection. (No orientation of individual triangles can be consistent across all edges of the mesh.) It takes 10 seconds to detect all intersections on a Dell 550 MHz using our method (405 multiple queries per second), while a brute-force method requires 48 seconds. The algorithm discovered two connected sets of intersecting triangles, which are highlighted in Fig. 23b and Fig. 23c. (In validation tests, our algorithm always discovered the exact same intersecting pairs as the brute force method.)

To use this method on large meshes in practice, it may be useful to combine it with methods for pruning out large areas where self-intersections cannot occur (for instance, using the method of [46]).
by adding the displacement magnitude to the previous bound.

Table 1 compares the performance of the three methods on six different datasets, two of which are introduced in this section and illustrated in Fig. 22: Camp Pendleton and Double Helix. Various types and numbers of queries were specified for each dataset. In general, there was a significant spatial coherence between the query points, as illustrated in Fig. 1. Unless otherwise specified, all timings have been measured on a Dell 550 Mhz PC.

When excluding the terrain dataset (Pendleton) Meshsweeper achieves anywhere between about 500 to a few thousand exact closest point queries per second. We note that the initialization time is significant for Meshsweeper, which uses mesh simplification, as opposed to the other two methods.

Its initialization time notwithstanding, Meshsweeper performs considerably better than the other two methods. This is clear when examining Fig. 24. Fig. 24 is a graph reporting, for each method, the total time necessary to perform 500 queries, including all necessary preprocessing. (The brute force method requires no preprocessing). Meshsweeper is anywhere between 5 and 20 times faster than the Octree method and about 100 times faster than the brute force method (for the maximum mesh complexity that we tested), whose complexity grows linearly in the mesh size. In rare cases, despite our efforts in achieving the best possible implementations, the Octree method provided answers that did not match the gold-standard (brute-force), while Meshsweeper matched the gold-standard in all cases within floating-point precision. This emphasizes the sensitivity of the closest-point problem and soundness of the Meshsweeper approach.

![Fig. 23. Speeding up the detection of self-intersections. (a) Polygonal model of a Klein bottle. (b) First connected set of intersecting triangles. (c) The two connected sets of intersecting triangles are highlighted, as detected using Meshsweeper.](image)

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>Performance Comparison between Meshsweeper, an Octree-Based Method, and a Brute Force Method Using Six Datasets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>Sphere Flake</td>
</tr>
<tr>
<td>Faces</td>
<td>53,422</td>
</tr>
<tr>
<td>Connected Components</td>
<td>1</td>
</tr>
<tr>
<td>Queries</td>
<td>6,500</td>
</tr>
<tr>
<td>Meshsweeper:Init.</td>
<td>35&quot;</td>
</tr>
<tr>
<td>Meshsweeper:Queries</td>
<td>21&quot;</td>
</tr>
<tr>
<td>Octree:Init + Queries</td>
<td>14'7&quot;1'3&quot;</td>
</tr>
<tr>
<td>Brute Force</td>
<td>8'45&quot;1'3&quot;</td>
</tr>
<tr>
<td>Meshsweeper queries/second</td>
<td>310</td>
</tr>
</tbody>
</table>

1 Timing measured on an SGI RealityEngine II. 2 Initialization time factored out. 3 Five hundred queries only. 4 Queries involve reporting all triangles closer than a specified distance d.

![Fig. 24. Performance graph. Comparison between Meshsweeper, Octree, and Brute Force methods. The total query time for 500 queries is reported, including any necessary preprocessing. The size of a mesh is measured by the number of triangles.](image)
11 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 nonrigid motion. The method takes full advantage of coherence between successive queries. We illustrated this method with various applications for interactive and noninteractive 3D graphics.

Another potential application of our method is motion planning in a complex polygonal environment: Each moving mesh could be associated with a Meshsweeper object (or perhaps all meshes could be combined in a single Meshsweeper object since nonrigid motion is allowed). A robot could query the Meshsweeper objects interactively, and adapt its path accordingly.

Mesh registration methods related to the Iterative Closest Point method [15] could also benefit from an efficient closest-point-on-mesh method. Commonly, a point-k-D tree is used for queries, reporting the closest vertex of the mesh instead of the closest point.

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. 6, 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 supports the constant time hypothesis better. Also, to the best of our knowledge, this worst-case example would apply to all methods.

Another possible avenue of future work is to extend our method to compute the separation distance between two polygonal meshes.

Acknowledgments

The author wishes to thank Tao Song He for providing the Sphere Flake dataset which served as an excellent inspiration for this work. Thanks to Viewpoint for the “main” model. Thanks to Elizabeth Fox for proofreading the manuscript. Dan Hunt was kind enough to lend his voice for a videotape that illustrates the paper. Thanks to Fausto Bernardini for providing an implementation of a hierarchical closest point method using an octree that was used for comparison purposes. Claudio Silva and Stephan Gumhold had helpful suggestions for the algorithm (topological sort and pruning). Thanks to Alan Kalvin and Joao Comba for their feedback. Many thanks to the TVCG reviewers. And, finally, the author wants to acknowledge the support of his colleagues and management at Multigen-Paradigm.

References


André Guéziec holds a PhD degree in computer science from the University Paris 11 Orsay that was obtained with honors in 1993. He graduated from the Ecole Centrale Paris in 1989. He is currently working for Sportvision, Inc. In 1992-1994, he was a postdoctoral fellow and adjunct faculty member at the Courant Institute of Mathematical Sciences of New York University, with a joint position at the NYU Medical Center. From 1994 to 1999, he was a research staff member at the IBM T.J Watson Research Center, where he worked on medical imaging and robotics, on geometric modeling, and on the mpeg-4 standard for 3D model coding. He holds seven patents and has published about 30 papers. He was the Keynote Speaker at the 1996 Leeds Statistics Symposium. Before joining Sportvision, Inc., he worked for two years at Multigen-Paradigm, a Computer Associates company, on system design for information visualization and visual simulation. He is a senior member of the IEEE and a member of the IEEE Computer Society.