Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Issue on SMI 2016

PinMesh—Fast and exact 3D point location queries using a uniform grid

Salles V.G. Magalhães [a,b,*], Marcus V.A. Andrade [a], W. Randolph Franklin [b], Wenli Li [b]

[a] Department of Informatics, Federal University of Viçosa, Viçosa, MG 36570-000, Brazil
[b] ECSE Department, Rensselaer Polytechnic Institute, Troy, NY 12180, USA

ARTICLE INFO

Article history:
Received 14 March 2016
Received in revised form
11 May 2016
Accepted 11 May 2016
Available online 17 May 2016

Keywords:
Point location queries
Exact computation
Parallel algorithm
Big data

ABSTRACT

This paper presents PinMesh, a very fast algorithm with implementation to preprocess a polyhedral mesh, also known as a multi-material mesh, in order to perform 3D point location queries. PinMesh combines several innovative components to efficiently handle the largest available meshes. Because of a 2-level uniform grid, the expected preprocessing time is linear in the input size, and the code parallelizes well on a shared memory machine. Querying time is almost independent of the dataset size. PinMesh uses exact arithmetic with rational numbers to prevent roundoff errors, and symbolic perturbation with Simulation of Simplicity (SoS) to handle geometric degeneracies or special cases. PinMesh is intended to be a subroutine in more complex algorithms. It can preprocess a dataset and perform 1 million queries up to 27 times faster than RCT (*Relative Closest Triangle*), the current fastest algorithm. Preprocessing a sample dataset with 50 million triangles took only 14 elapsed seconds on a 16-core Xeon processor. The mean query time was 0.6 µs.

1. Introduction

The 3D point location problem and its 2D analog have many applications in Computer Graphics, Computer Aided Design, Additive Manufacturing, Computer Games, and Geographic Information Science [20,23]. Point location is required for problems such as computing the intersection of two meshes and detecting collisions. Thus a correct and efficient algorithm is important.

Several existing solutions are presented in Ogayar et al. [20]. The most important are Jordan Curve and Feito-Torres. The algorithm based on the Jordan Curve Theorem [19] is an extension of PNPOLY, the well known ray casting 2D point-in-polygon algorithm [10]. The idea is to cast a semi-infinite ray up from the query point, and count the number of intersections between this ray and the polygon's edges. The point is considered to be inside the polygon if and only if the number of intersections is odd. Although this method is simple and efficient for performing single queries, it is a challenge to efficiently implement it to correctly handle all the singularities, which get much worse in 3D.

Feito-Torres [7] builds on the ease of determining whether a point is in a tetrahedron by evaluating the sign of a determinant. It partitions the polyhedron into tetrahedra, one between each triangular polyhedron face and the coordinate origin, and then counts how many of them contain the query point q. q is in the polyhedron if and only if that is odd.

These methods work well for single queries against the polyhedra; without preprocessing, a query takes linear time in the number of faces. However, in the usual cases of many queries on the same dataset this is not optimal. Therefore, Ogayar et al. [20] also extend these two algorithms with common pre-processing techniques (such as the use of an octree to index the mesh) to accelerate the multiple query case.

Recently, Liu et al. presented RCT (*Relative Closest Triangle*) [16], an efficient method for locating points in 3D triangular meshes. RCT can perform point location tests even in models composed of multi-materials, where internal boundaries divide the polyhedron into smaller polyhedra and the objective is to determine in which smaller polyhedron the query point is.

For each query point q, RCT uses an octree to efficiently locate a mesh triangle t that is visible from q. t is visible from q if a line segment that does not cross the mesh can be traced between q and t. Once a visible triangle t is found, an orientation operation that evaluates t's normal is used to determine q's position with respect to the polyhedron. To the best of our knowledge, RCT is the fastest algorithm available for performing multiple queries in polyhedra. Indeed, according to the experiments in Liu et al. [16], RCT was much faster than an efficient ray-crossing algorithm based on Axis-Aligned Bounding Boxes, and also than the CGAL [4] point location algorithm.

* Corresponding author.
E-mail addresses: salles@ufv.br (S.V.G. Magalhães), marcus@dpu.ufv.br (M.V.A. Andrade), mail@wrfranklin.org (W.R. Franklin), liw9@rpi.edu (W. Li).

http://dx.doi.org/10.1016/j.cag.2016.05.017
0097-8493/© 2016 Elsevier Ltd. All rights reserved.

An important challenge in computational geometry, and in particular, in point location algorithms, is avoiding incorrect results caused by floating-point arithmetic's roundoff errors. Heuristic solutions include using a tolerance, or snap rounding. These work for awhile. However, as datasets become larger, the chance of these heuristics failing increases. Any chance of a failure prevents the algorithm from being used as a guaranteed subroutine in a larger system.

This paper presents PinMesh, a very fast algorithm to perform point location queries in 3D polyhedral meshes. The mesh is quickly indexed using a 2-level uniform grid. Also, PinMesh uses exact arithmetic with rational numbers to prevent roundoff errors and symbolic perturbation with Simulation of Simplicity (SoS) to handle geometric degeneracies or special cases. It can preprocess a dataset and perform 1 million queries up to 27 times faster than RCT. Preprocessing a sample dataset with 50 million triangles took only 14 elapsed seconds on a 16-core Xeon processor. The mean query time was 0.6 µs.

2. The algorithm

2.1. Mesh representation

Our 3D universe is partitioned into polyhedra bounded by triangular faces, t. Each t is defined by a list of pointers to its 3 vertices, defined by their Cartesian coordinates. The order of its vertices in the list gives t a well-defined positive and negative side. t also stores the labels of the adjacent polyhedron on its positive and on its negative side. Two triangles can intersect only on shared vertices or edges.

We save space by not explicitly storing the polyhedra, since that information is unnecessary to PinMesh. Thus, the global topology is represented only implicitly. However we do require that the polyhedra be watertight. Not having to worry about global topological properties like face shells and nesting makes many things easier.

Fig. 1 shows an example of a mesh containing 11 triangles and 2 polyhedra (polyhedron 1 in red, polyhedron 2 in blue). Triangle ABC bounds polyhedra 1 (on the negative side) and 2 (on the positive side) while the other triangles bound the outside of the mesh and either polyhedron 1 or 2. The query point q is in polyhedron 2.

2.2. Performing queries

Given a mesh M and a set of query points Q, the objective is to determine which polyhedron contains each point $q \in Q$.

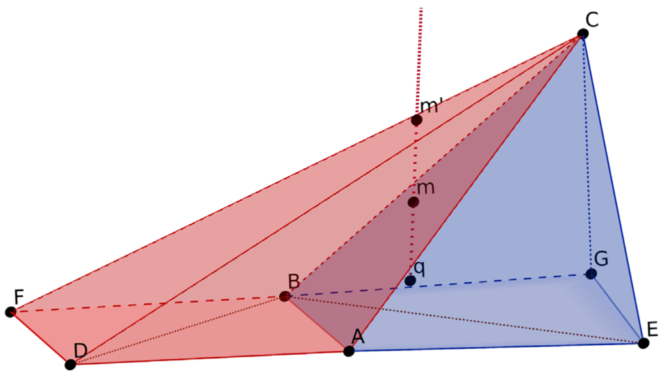

Fig. 1. Example of an input mesh and a query point. (For interpretation of the references to color in this figure, the reader is referred to the web version of this paper.)

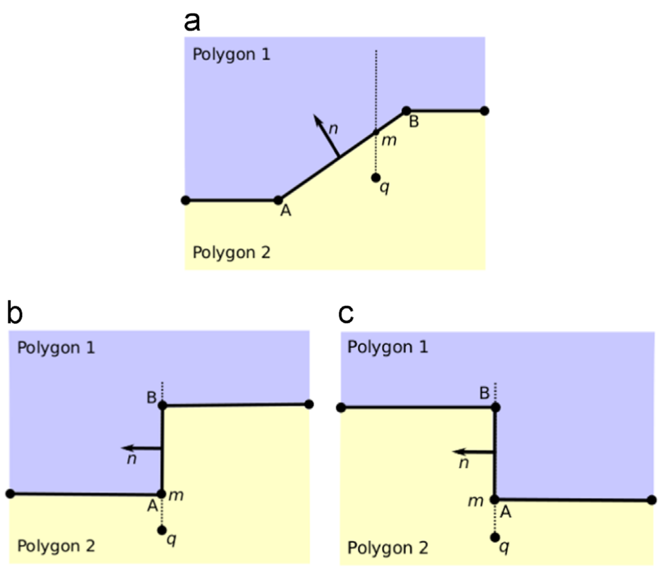

Fig. 2. The messy special case of vertical edges in 2D point location.

We determine q's location by casting a vertical ray l from q going in the positive z direction (the particular orientation is not important). We then determine the first mesh triangle t that intersects l, measured along l from q. Then, ignoring special cases, q is in one of t's two adjacent polyhedra. Which one is determined by which side of t that q is on. Special cases (e.g., when l is parallel to t) will be discussed later.

Fig. 2 presents an example of the 2D version of the point containment problem (the 3D version is equivalent) that illustrates the challenge of determining the position of a query point q basing on the orientation of a vertical edge (vertical triangle in 3D): in Fig. 2(a) the edge AB is the first edge crossed by the ray traced from q, and, since the sign of the dot product between $(0,-1)$ and n is negative, this means that q is on the polygon that is on the negative side of AB. If the normal was parallel to the x-axis (Fig. 2 (b) and (c)), on the other hand, the first point hit by the ray would be on a vertex of AB, and thus the process of determining in what polygon q is would need to treat an ambiguity: even though the segment AB is the same in Fig. 2(b) and (c) and q is on the same polygon in these two figures, in (b) q is on the polygon on the negative side of AB while in (c) it is on the positive side.

PinMesh needs three basic operations in order to determine in what polyhedron a query point q is:

- *isOnProj*(t,q): given a triangle t and a query point q, returns true if and only if the projection of q onto the plane passing through t is on the interior of t.
- *isAbove*(t,q): given a query point q and a triangle t such that *isOnProj*(t,q) is true, returns true if and only if the projection of q onto t is above q, i.e., the triangle is above the point.
- *isBelow*(t,t',q): given two triangles t and t' directly above a query point q, returns true if and only if the z component of the projection of q onto t is smaller than the z component of the projection of q onto t'.

Thus, to locate q, we first construct a subset T of the triangles in the mesh M that are directly above q, that is, $T = \{t \in M \mid \textit{isOnProj}(q,t) \text{ and } \textit{isAbove}(q,t)\}$. Then the lowest triangle u in T is selected using *isBelow*(q,t,t') to compare pairs of triangles. Finally, the location of q is determined by verifying the sign of the dot product between the vector $(0,0,-1)$ and u's normal. The operation *isOnProj*(t,q) was implemented using the barycentric coordinates of the projection of t onto the plane $z=0$.

Similarly, *isBelow*(t, t', q) was implemented by computing the plane equations for the triangles t and t', and then, using these equations to project q onto the triangles and compute the z-coordinates of the two projections; see Section 3.4.

2.3. Using a 2-level uniform grid to accelerate the computation

The uniform grid is useful in computational geometry to efficiently cull a combinatorial set of pairs or triples of objects, to find a much smaller subset whose members are likely to coincide. For data that is uniformly independently and identically distributed, or even that satisfies a Lipschitz condition so that the maximum local density is bounded, the expected number of pairs or triples of objects processed is linear in the size of the input plus the output [2,8,12]. The basic idea is to superimpose a grid over the data, with the grid cell size set so that the expected number of edges per cell remains constant as the total number of edges grows. Then, insert into each cell c the elements (e.g., edges in 2D polygonal maps or triangles in 3D triangulations) of the dataset intersecting c.

To handle very uneven data, 2-level grids have been used successfully in applications like ray-tracing [13], exact parallel 2D polygonal map intersection [17], etc. Although 3-level grids and general trees seem to be attractive extensions, Magalhães et al. [17] found them to be much slower. They are less parallelizable, slower to traverse and to construct. In more detail, the "teapot in a stadium" problem, where there is a small object in a large universe, is cited as a bad case for a uniform grid. However it is not as bad as it would seem because empty uniform grid cells are almost free, and an alternative to try to mitigate this problem could be to set the grid resolution for the densest region, i.e., at the teapot. Also, an adversary can use a few teapots in the stadium to force an octree to refine itself in many places, and each octree cell is more expensive than a uniform grid cell. For the data that we have tried, the sweet point is a 2-level grid.

Therefore this paper uses a 2-level 3D uniform grid, quickly created in parallel, to accelerate the point queries. We performed tests with several datasets with different sizes from 3 distinct sources and in all experiments the uniform grid presented a good performance. Thus, we believe that, except for very pathological datasets, the uniform grid is a good design choice.

Given the input mesh M, a 3D regular $g_1 \times g_1 \times g_1$ grid G overlapping M is created. g_1, the number of grid cells in each dimension, is called its *resolution* or *size*. Then a pointer to each triangle is inserted into each grid cell that it intersects. Here we perform a time-space tradeoff, by inserting the triangle t into all the cells overlapping its bounding box, instead of into only the cells that overlap it. Because of this strategy, the ragged array may have a few more elements than necessary making the query a little slower and the optimum grid resolution a little coarser. In future we may change this decision since, while this strategy simplifies the pre-processing step, in some specific datasets (for example, meshes containing some long triangles whose bounding-boxes intersect a significant percentage of the grid cells) this approximation could degrade PinMesh's performance.

If the grid is sized such that the expected number of triangles per cell is a constant (i.e., if the grid resolution is chosen as a function of the number of triangles in the input mesh such that the average number of triangles per cell is a constant), then the expected query time to locate a point q will be constant, for the following reasons. Determining what grid cell c contains q takes constant time. Testing the ray l against one triangle takes constant time, and so testing against all the triangles in c takes constant expected time. If l hits at least one triangle, the query result is found in constant time. However, there is a probability p that l intersects no triangle in c, and so we must test l against the triangles in the next higher cell, and so on up the column of cells until we find a cell with a triangle that intersects l, or reach the each of the grid. Under the assumption that the data is distributed randomly independent of the cells, the probability of l not hitting any triangle in any particular cell that it is tested against is always p. Therefore the expected number of cells processed until the first cell with a triangle that l hits follows an exponential probability distribution with mean $1/(1-p)$, i.e., constant. Therefore the expected time to locate q is constant.

The grid cells' contents (the pointers to the triangles) are stored in a ragged array. This is more compact than an array of linked lists, and stores each cell's contents contiguously, which might play better with the computer's cache. This common technique is also used in parallel bucket sorting.

Let the total number of grid cells be $n = g_1^3$. The grid cells are numbered from 0 to $n-1$. The total number of triangle pointers in all the cells, equivalently the number of (cell,triangle) incidences, is m. The ragged array data structure contains two arrays: $\vec{A}$ with m elements containing the pointers to the triangles and $\vec{B}$, a dope vector with $n+1$ elements. The number of triangle pointers in the i-th cell is $B_{i+1} - B_i$. The j-th pointer contained in the i-th cell is A_{B_i+j}. Assuming that there are fewer than 2^{32} (cell,triangle) incidences, 4 bytes suffice for one pointer. Therefore the total amount of space required for the ragged array is $4n+4m+4$ bytes.

In contrast, using a linked list for each cell might require $4n+8m$ bytes depending on implementation details. STL vectors might require $4n+4m+24p$ bytes, where $p \le n$ is the number of non-empty cells, i.e., cells with vectors, and assuming that each vector has a 24 byte overhead. This ignores the unused empty space allocated when a vector is resized. The ragged array is much more compact.

While the creation of the ragged array requires two scans through the data, it replaces the allocation of many small memory blocks with one big allocation of the array A. This strategy has two advantages.

First, by avoiding the use of dynamic data structures such as lists and dynamic arrays, the ragged array reduces the memory overhead: since the uniform grid is 3D, the number of cells grows cubicly with g_1, and so the total memory requirement of the grid would be big if each cell stored a dynamic array (with a header and, because of its allocation strategy, usually more slots than the number of elements stored), or a linked list (with its pointers). By using a ragged array, on the other hand, we have exactly one slot for each triangle stored and each grid cell needs to store only one pointer.

Also, performing one big allocation instead of g_1^3 smaller allocations does not fragment the memory, has more data locality, and is faster.

The uniform grid works well even for uneven data for various reasons [2,5,8,9]. First, the total time is the sum of one component (inserting the triangles) that runs more slowly for larger g_1, plus another component (performing the queries) that runs more quickly. The sum varies only slowly with changing g_1. Second, with the ragged array, an empty grid cell is very inexpensive, so that sizing the grid for the dense part of the data works. Nevertheless, to deal with very uneven data, we incorporated a second level grid into those few cells that contain more triangles than some given threshold. For each cell in the first level grid we store a pointer to a second level uniform grid. Fig. 3 shows a 2D example, where cells with more than 2 edges were refined.

This nesting process could be recursively repeated in order to ensure that all grid cells contain fewer triangles than the threshold, creating a structure similar to an octree, but with more branching at each level. Our solution could be considered a special case of that. However, as mentioned earlier, the general solution uses more space for pointers (or, for compact representations, is

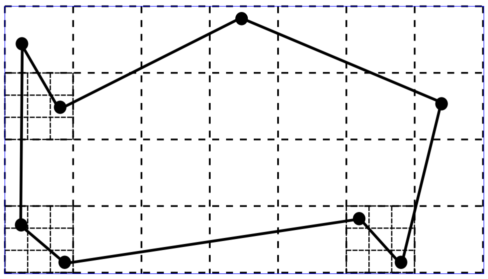

Fig. 3. Example of a 2D uniform grid, where the first level has 4×7 cells and the second level has 3^2 cells. Cells with over 2 edges were refined. Adapted from [17].

expensive to modify) and is irregular enough that parallelization is difficult. Our tests found the best performance with just one or two levels. This can be explained because the first level grid, in general, has many cells with more elements than the threshold justifying the second level refinement. But, past the second level, only a few number of cells exceed the threshold and the overhead (processing time and memory use) to refine those cells is never recaptured.

If g_1 is high, there will be many empty cells. Each empty cell c is completely inside one polyhedron, and we label it with that polyhedron's id. Now, a query against an empty cell will be very fast and constant time. A cell label is determined computing the containing polyhedron with the usual semi-infinite ray, and then using a scan-line flood fill algorithm to propagate that label to adjacent empty cells. We repeat this so long as one empty cell has not yet been labeled.

3. Implementation details

PinMesh contains a preprocessing and a querying step. Algorithm 1 presents the preprocessing pseudo-code. That creates the first level uniform grid, refines some of its cells to create second level grids inside them, inserts the triangles into the ragged array, and labels each empty grid cell with the polyhedron containing it.

Algorithm 1. PinMesh's preprocessing step.

```
1: $M$: mesh represented as a set of triangles
2: $g_1$: resolution of the first level of the uniform grid
3: $g_2$: resolution of the second level of the uniform grid
4: $maxTrianglePerCell$: threshold for refining a cell
5: //Create and refine the uniform grid $G$
6: $\mathbf{G} \leftarrow g_1^3$ 3D uniform grid
7: for each triangle $t$ in $M$ do
8:     Insert $t$ into the $G$'s cells intersecting its bounding-rbox
9: end for
10: for each grid cell $c$ in $G$ do
11:     if $|triangles(c)| > maxTrianglePerCell$ then
12:         Create a $g_2^3$ uniform grid in $c$
13:         for each triangle $t$ in $triangles(c)$ do
14:             Insert $t$ into the $c$'s cells intersecting its b.box
15:         end for
16:     end if
17: end for
18: //Label $G$'s cells
19: Initialize $G$'s cells labels with $\varnothing$
20: for each empty grid cell $c$ in $G$ do
21:     if $c.label = \varnothing$ then
22:         $q \leftarrow$ point in the center of $c$
23:         label $\leftarrow$ locatePointInMesh(q, M, G)
24:         //Use a scanline algorithm to label $c$'s connected
25:         //component of empty cells
26:         scanlineFloodFill(c,label)
27:     end if
28: end for
```

Algorithm 2 presents the high level pseudo-code for the querying, or point location function. For simplicity, we assume that the grid has only one level (the algorithm for a grid with two levels is similar).

Since the objective is to find the lowest intersection point m between a triangle (the *lowest triangle*) and a vertical ray l starting at q, the algorithm iterates up through the grid cells intersecting l. The loop stops if a labeled cell is hit. Otherwise, each triangle t in c is tested against l and the t with the lowest intersection is returned. If none of the t in c intersect l, then we move up to the next higher c.

For performance, every time a lower triangle is found the highest cell that needs to be processed is updated with the cell c containing the projection of q on *lowestTri* (line 15) since no cell above c can have a lower triangle. This strategy reduces the number of grid cells that need to be processed.

Fig. 4 illustrates this process in 2D. The algorithm starts the search on c_0 (q's cell), and among the 4 edges in c_0, only edge xy intersects the ray l (on point m_2). Notice that, even though xy is in c_0, m_2 is not in c_0 since xy intersects several grid cells. In the next iteration, cell c_1 is processed and the edge uv containing the intersection point m_1 (that is lower than m_2) is found. Since the lowest intersection seen so far (m_1) is on an edge in c_1, no grid cell above c_1 needs to be processed.

After the loop on line 7 ends, the algorithm tests if the pointer to the lowest triangle was initialized. If it was not, this means that all cells above q_c (the cell containing q) were processed and no triangle directly above q was found. Thus, q must be outside the mesh. If the pointer to the lowest triangle was initialized, we know that the face of this triangle is directly above q (and the projection of q on this face represents the lowest point on the mesh that is directly above q), and thus the signal of the dot product between $(0, 0, -1)$ and the normal of the lowest triangle is used to determine in what polyhedron q is.

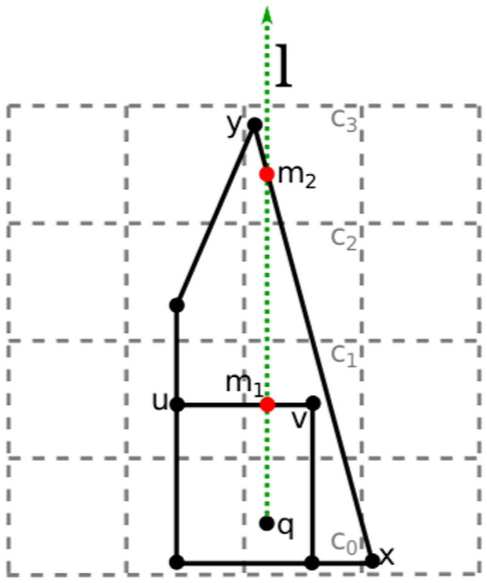

Fig. 4. Computing point locations with a uniform grid.

Algorithm 2. Function locatePointInMesh(q,M,G).

```
1: $q$ (argument): query point
2: $M$ (argument): mesh represented as a set of triangles
3: $G$ (argument): 3D uniform grid created on $M$
4: $(q_{cx}, q_{cy}, q_{cz}) \leftarrow$ coordinates of the grid cell $q_c$ containing $q$
5: $highestCzToProcess \leftarrow G.gridResolution$
6: $lowestTri \leftarrow \varnothing$
7: for $c_z$ in $q_{cz} \ldots highestCzToProcess$ do
8:    $c \leftarrow$ grid cell of G in coordinate $(q_{cx}, q_{cy}, c_z)$
9:    if lowestTri $= \varnothing$ AND $c.label \neq \varnothing$ then
10:       return $c.label$
11:    end if
12:    for each $t$ in $triangles(c)$ do
13:       if $isOnProj(t, q)$ AND $isAbove(t, q)$ then
14:          if lowestTri $= \varnothing$ OR $isBelow(t, lowestTri, q)$ then
15:             lowestTri $\leftarrow t$
16:             highestCzToProcess $\leftarrow getGridZProjection(q, t)$
17:          end if
18:       end if
19:    end for
20: end for
21: if lowestTri $\neq \varnothing$ then
22:    if $lowestTri.normal_z < 0$ then
23:       return lowestTri.positiveSide
24:    else
25:       return lowestTri.negativeSide
26:    end If
27: else
28:    return OUTSIDE //q is outside the mesh
29: end If
```

3.1. Rational numbers

Besides the special cases (that will be treated later), PinMesh could also fail because of floating point round-off errors. We avoid that by using the C++ package *GMPXX* [11] to store and process all coordinates using rational numbers.

Instead of representing each coordinate as a floating point number, a pair of arrays of integers is used, one each for the coordinate's numerator and denominator. *GMPXX* implements routines for the standard arithmetic operations. Their results are exact, e.g., $1/3 + 3/7 = 16/21$ exactly. That example also illustrates that the result of combining two rationals often has as many digits as the total number of digits in the inputs. This is tolerable if the depth of the computation tree is small, which is true for PinMesh.

Since the size of a rational number is unpredictable and can change, *GMPXX* allocates them on the heap. The problem is that heap operations are superlinear in time; the more objects that the heap contains, the more time that each object construction or destruction takes. This gets even worse when several parallel threads are attempting to modify the common heap, since the modifications must be protected with semaphores and serialized.

Adapting parallel code to use rational numbers is not the straightforward task that it would seem to be. Simply replacing the floating-point variables with rational ones does not lead to good performance, especially with large datasets. PinMesh was carefully engineered to be efficient. For example, we avoided creating temporary rational numbers by using pre-allocated variables and carefully forming expressions. We also performed a space–time tradeoff by storing values that would be needed later, such as the grid cell containing a vertex v, rather than repeatedly recomputing them.

Another advantage of using rationals is that the Simulation of Simplicity [6] technique we use to treat the special cases requires exact arithmetic.

Computing with rationals is much slower; while a floating addition takes one cycle, a rational addition is a subroutine involving three multiplications and an addition of vectors of integers followed by a greatest common divisor operation to cast out common factors in the numerator and denominator. We feel that the advantage of no more roundoff errors outweighs the disadvantages.

3.2. Parallel implementation

PinMesh was engineered to be efficient when parallelized. Our current implementation uses OpenMP, which extends C++ with pragmas to compile for a multi-threaded shared-memory model. For example, there is a pragma to say that the iterations of the following `for` loop do not depend on each other and can execute in parallel. During PinMesh's preprocessing, the uniform grid, the axis-aligned bounding boxes of the triangles and the location of the vertices in the grid cells are all computed in parallel.

We allocate the uniform grid and insert the triangles into the corresponding grid cells sequentially because it is such a small fraction of the total algorithm time.

Once the first level of the grid is created, the next step is to refine some grid cells. Since the separate cells being refined do not affect each other and smaller (independent) ragged arrays will be created in each of these refined cells, we do this in parallel. Because in the creation of the nested uniform grids no synchronization is performed and no memory is allocated for the rational numbers (since the triangles are inserted based on the pre-computed grid coordinates of their bounding boxes, no operation with rational numbers is performed during this step), the grid refinement step presents a good parallel scalability; see Section 4.

Next we labeled the empty grid cells with the ids of their containing polyhedra, using a scan-line flood-fill algorithm to label the connected components of empty grid cells. To avoid the implementation of a parallel scan-line algorithm (which would require synchronizations), we divided the first level uniform grid into cubic blocks (for example, a 64^3 grid could be divided into 512 8^3 blocks and used the flood-fill algorithm to label the connected components in each block independently. Since the blocks are processed independently, this processing was performed in parallel.

Finally, since the queries are read-only, they can all be performed in parallel.

3.3. Special cases

Special cases, also called geometric degeneracies, are a nasty part of the life of a geometric algorithm designer. This section presents our solutions to the special cases in point location.

The location of a query point q is computed with ray casting. Some special cases happen when the ray intersects two or more triangles at a same point m. I.e., there is a tie for the lowest triangle, and m is on an edge or vertex. If the tied triangles have different polyhedra below them, then which one is correct?

Assume q is not on a triangle. If m is on a vertex or edge of a triangle, we cannot arbitrarily choose one of the triangles to determine in what polyhedron q is since some of these triangles may not even be on the surface of the polyhedron containing q. For example, see Fig. 5: we have two pyramids and the ray casted from the query point q intersects the mesh firstly in the edge AB. This edge is shared by 3 triangles: ABC, ABD and ABE, and therefore the lowest point directly above q (point m) is shared by the three triangles. However, only the triangles ABD and ABE bound the outside of the mesh (where q is located).

Another possibility is that q is on t. We could handle this case directly. However after Simulation of Simplicity's perturbation, q

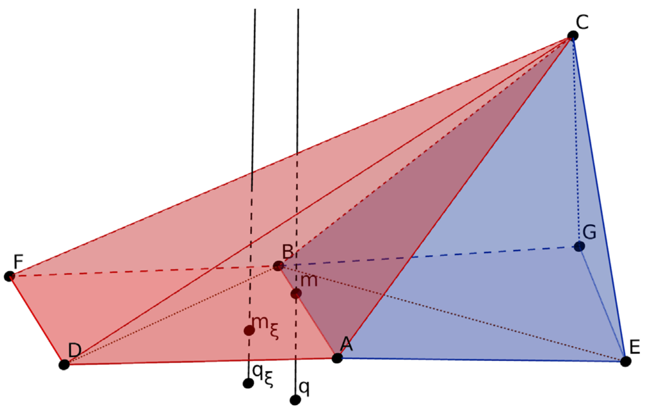

Fig. 5. The ray cast from the query point q intersects the mesh at the point m in the edge AB and the ray from q_ϵ intersects the mesh at the point m_ϵ on the face ABD.

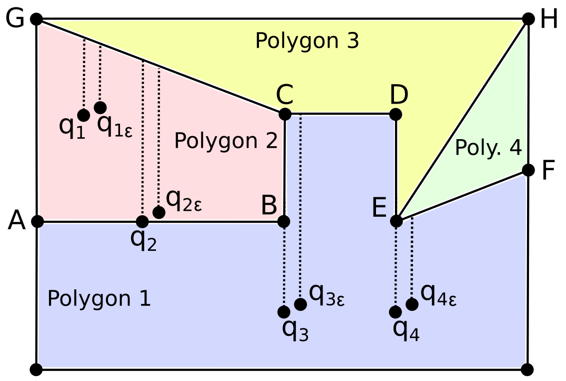

Fig. 6. Special cases in the 2D version of the point location problem removed using SoS.

will no longer be on t, and so this case goes away. While a point will never be considered to be on the mesh surface, if the ability to report points on the surface is a desired feature it is trivial to adapt PinMesh (without changing its running time) to perform this kind of computation.

To correctly handle these special cases, we used Simulation of Simplicity (SoS) [6]. This is a general purpose symbolic perturbation technique designed to treat special (degenerate) cases, or geometric coincidences in the data. The inspiration for SoS is that you might solve the problem of q being exactly on t by moving q, or by moving t's vertices, slightly. However, too big a perturbation may create new problems, while a too small one may be ineffective because of the limited precision of floating point numbers.

Fig. 6 illustrates some special cases of the 2D version of the problem and the effect caused by SoS, when each point q_i is slightly translated to $q_{i\epsilon}$.

SoS is a brilliant solution that uses a symbolic perturbation by a formal indeterminate infinitesimal value, ϵ, or by ϵ^i, for some natural number i. The mathematical formalization of SoS extends some exactly computable field, such as exact reals or rationals, by adding orders of infinitesimals, ϵ^i. Floating point numbers with roundoff error cannot be the base. Indeed, floats are not even a field because roundoff errors cause most of the field axioms to be violated. E.g., because $(10^{-20}+1)$ rounds to 1, so $(10^{-20}+1)-1 \neq 10^{-20}+(1-1)$.

The infinitesimal ϵ is an *indeterminate*. It has no meaning apart from the rules for how it combines. E.g., if a is a positive finite number, then two of those rules are that $a+\epsilon$ is equivalent to a and $\epsilon < a$. For a charming take on this, see [14].

All positive first-order infinitesimals are smaller than the smallest positive number. All positive second-order infinitesimals are smaller than the smallest positive first-order infinitesimal, and so on. All this is logically consistent and satisfies the axioms of an abstract algrebra field. It is attractive to think of ϵ as an actual very small finite number, perhaps 10^{-10} or 10^{-100}. Although it may be useful to develop in initial intuitive understanding, the use of a concrete value for the infinitesimals could lead to wrong conclusions.

The result of SoS is that degeneracies are resolved in a way that is globally consistent. If q were previously on t, then after SoS is applied, q is no longer on t, but is either on its positive or negative side. Which one it is is chosen to be globally consistent even if there are many q and many t. Without a logical framework such as SoS, this would be very difficult to do correctly.

To see how hard this would be, consider the problem of intersecting two edges or line segments, e_1 and e_2. The relevant degeneracies occur when the end vertex of one lies on the other, or when two vertices coincide. With SoS, the lines are symbolically perturbed so that this does not happen. Considering the degeneracies explicitly is difficult because there are many more cases to get correct. Consider the problem of intersecting two polylines p_1 and p_2. Without degeneracies, p_1 and p_2 cross if and only if their constituent edges intersect an odd number of times. With degeneracies, every incidence type of a vertex or edge with another vertex or edge must be considered. SoS simplifies the problem considerably.

For another example, consider the 2D point-in-polygon test PNPOLY [10]. The simple case of the ray intersecting a vertex is easy; the whole PNPOLY function has only 8 lines of executable C code. But now consider running a vertical ray through a triangulation as done by PinMesh. There are some special cases to consider: for example, as it was previously described the ray may hit a triangle whose normal is parallel to the plane $z=0$ or it may hit a vertex or edge of a triangle that do not even bound the polyhedron containing the point (as shown in Fig. 5). As it will be shown later, with SoS these special cases are automatically handled.

Implementing SoS and computing with orders of infinitesimals would seem to be very slow. However, we are using them to determine signs of expressions. The only time that the infinitesimals change the result is when the exact computation with rationals, would have caused a tie in a predicate, e.g., make a determinant to be zero. Then, the infinitesimals break the tie. The effect is to make the code harder to write and longer. However, unless a degeneracy occurs, the execution speed is the same. When a degeneracy does occur, the code is slightly slower.

We implement SoS in PinMesh by conceptually translating the query point $q=(q_x,q_y,q_z)$ to $q_\epsilon=(q_x+\epsilon, q_y+\epsilon^2, q_z+\epsilon^3)$. As will be shown later, this specific choice of perturbation correctly handles all these special cases.

The problem that this solves occurs when the sign of some function f, often a determinant, is used to determine a control path: for + go right, for − go left, but what about 0? When the inputs to f are infinitesimals, then sgn(f) is never 0. However, we do not actually compute with infinitesimals. Rather, we analyze their effect, which is to change sgn(f) when it would have been zero. So, the new code has extra clauses that execute only in that case. Otherwise, the code is the same speed as before.

Edelsbrunner and Mücke [6] presented three requirements to guide the choice of the polynomials used to represent the perturbations in an algorithm using SoS:

1. The perturbed geometric objects must be simple (non-degenerate) if $\epsilon > 0$ is sufficiently small.
2. If an object is non-degenerate, then its perturbed version must retain the properties of its original version.
3. The computational overhead of processing the perturbed objects should be negligible.

In [6] a perturbation scheme is presented to avoid degeneracies in the point orientation problem in E^d. In this problem, the geometric objects are points and the scheme added a different infinitesimal perturbation (more specifically, the j-th coordinate of the i-th point is translated by $\epsilon^{2^{id-j}}$) to each coordinate of each point. By doing that, all the perturbed points are in general position (in 3D, this means that no set of 4 perturbed points can be coplanar), and therefore, the input is non-degenerate. As shown by the authors, this perturbation choice satisfies all the 3 conditions presented above.

We claim that the perturbation scheme used in our work is suitable for point location, and also satisfies the conditions. Condition 2 is automatically satisfied for a sufficiently small ϵ, that is, if the lowest mesh point above a query point q is on the interior of a triangle, the lowest point on the mesh directly above q_ϵ will also be on the interior of the same triangle. Similarly, if q is not inside the mesh, q_ϵ will also not be in the mesh. As will be shown later, the computational overhead of processing q_ϵ instead of q is negligible, and thus condition 3 is also met.

To show that condition 1 is also satisfied we need to show that q_ϵ will be a non-degenerate input, that is, q_ϵ will never be exactly below a vertex or edge of the triangles, and also q_ϵ will never be on the mesh surface.

First, let us show that q_ϵ will never be on the mesh surface. If q is on a mesh triangle t, q_ϵ cannot be on the plane Π passing through t for the following reason: suppose Π has a plane equation: $ax+by+cz+d=0$ and both q and q_ϵ are on Π. Since q is on Π, we have $aq_x+bq_y+cq_z+d=0$, and since q_ϵ is also on Π we have $a(q_x+\epsilon)+b(q_y+\epsilon^2)+c(q_z+\epsilon^3)+d=0$. Thus, $a\epsilon+b\epsilon^2+c\epsilon^3=0$, and since $\epsilon>0$, $a+b\epsilon+c\epsilon^2=0 \Rightarrow a=-b\epsilon-c\epsilon^2$. Because a, b and c cannot simultaneously be 0 , b and c are not simultaneously 0, and thus a is infinitesimal (which is impossible since, by definition, an infinitesimal is smaller than any measurable value).

Also, q_ϵ cannot be on the surface of another triangle t' not intersecting q because a ball with infinitesimal radius centered on q cannot intersect t'.

Next, q_ϵ will never be exactly below a vertex or edge of the triangles. Indeed, consider the projections q' and q'_ϵ of, respectively, q and q_ϵ onto the plane $z=0$. To show that q_ϵ will never be directly below a vertex or edge we need to show that q'_ϵ will never be on the projection of any vertex or edge onto $z=0$.

Lemma 3.1. *If q' is exactly on the projection v' of a mesh vertex v onto the plane $z=0$, q'_ϵ cannot be on the projection of any vertex or edge.*

Proof. Since v' is a point and q' is translated by a positive distance, q'_ϵ cannot be on v'. Also, q'_ϵ cannot be on the projection u' of any other vertex u onto $z=0$ since, otherwise, the distance between v' and u' would be proportional to $\sqrt{\epsilon^2+\epsilon^4}=\epsilon\sqrt{1+\epsilon^2}$, which is infinitesimal.

Furthermore, if q' is on v' then q'_ϵ cannot be on the projection e' of an edge e onto $z=0$ for two reasons. First, if v' does not intersect e', the shortest distance between e' and v' should be a non-infinitesimal positive value, but the distance between q' and q'_ϵ is infinitesimal. Second, if v' intersects e' and q'_ϵ is on e', this means that q' and q'_ϵ should be on the same edge e'. However, q' and q'_ϵ could not be on the same edge because $q'_\epsilon=(q'_x+\epsilon, q'_y+\epsilon^2)$, and thus the slope of this edge would be infinitesimal.□

It is also straightforward from this previous lemma that if q' is on the projection of an edge then q'_ϵ will not be on the projection of a vertex.

It is worth mentioning that the property that q' and q'_ϵ could not be simultaneously on the same segment would not be true for any edge if q_ϵ was equal to $(q_x+\epsilon, q_y+\epsilon, q_z+\epsilon)$ since, in this case, q' and q'_ϵ could be simultaneously on a segment with slope 1. This shows that a careful choice of the infinitesimals is an important task for correctly developing a SoS strategy.

Lemma 3.2. *If q' is on the projection e' of a mesh edge e onto the plane $z=0$, q'_ϵ will not be on the projection of any edge.*

Proof. As mentioned above, q' and q'_ϵ cannot be simultaneously on the same edge (see points q_2 and $q_{2\epsilon}$ in Fig. 6). Thus, we only need to show that if q' is on e', q'_ϵ will not be on the projection f' of another edge f onto $z=0$.

If f' does not intersect e', the shortest distance between f' and e' should be a non-infinitesimal positive value, and thus q'_ϵ could not be on e' (otherwise the shortest distance between e' and f' would be infinitesimal).

If f' intersects e' on a vertex v', q' could be either on v' or not. As mentioned in the previous lemma, if q' is on v', then q'_ϵ cannot be on an edge. If the smallest angle between e' and f' is not zero, it should be a positive non-infinitesimal and since the distance between q' and v' is also a positive non-infinitesimal value, then an infinitesimal disc centered on q' could not intersect the projection of any edge other than e'. Because the distance between q' and q'_ϵ is infinitesimal, it follows that q'_ϵ cannot be on f'.

If the smallest angle between e' and f' is zero and the two edges intersect, it is clear that if q' is on e', then q'_ϵ could not be on f' (otherwise the slope of these edges would be infinitesimal).□

To conclude, all query points q will be translated to a position q_ϵ that is not on the mesh and either below the interior of a triangle or not below any triangle. Also, since a perturbed point q_ϵ will never be directly below a vertex or edge, q_ϵ will also never be directly below a vertical triangle (that is, a triangle whose normal is parallel to the plane $z=0$).

3.4. Implementing the symbolic perturbations

The only parts of PinMesh that directly deal with the point coordinates and need to be adapted to use SoS are the three main operations described in Section 2.2: *isOnProj*(q,t), *isAbove*(q,t) and *isBelow*(q,t,t'). This section will describe how they were implemented and adapted for SoS.

Given a triangle t and a point q, *isOnProj*(q,t) uses q's barycentric coordinates to determine whether or not the projection of q onto the plane passing through t is on t. More specifically, we project both q and t onto the plane $z=0$, creating the point q' and the triangle t', compute the barycentric coordinates of q' with respect to t', and then use these coordinates to check whether or not q' is on the interior of t'.

Consider the three vertices t'_0, t'_1 and t'_2 of t' and the point q'. The barycentric coordinates λ_0, λ_1, λ_2 of q' with respect to t' can be computed using the following equations:

$$\lambda_0=\frac{(t'_{1y}-t'_{2y})\times(q'_x-t'_{2x})+(t'_{2x}-t'_{1x})\times(q'_y-t'_{2y})}{det} \tag{1}$$

$$\lambda_1=\frac{(t'_{2y}-t'_{0y})\times(q'_x-t'_{2x})+(t'_{0x}-t'_{2x})\times(q'_y-t'_{2y})}{det} \tag{2}$$

$$\lambda_2=1-\lambda_0-\lambda_1 \tag{3}$$

$$det=(t'_{1y}-t'_{2y})\times(t'_{0x}-t'_{2x})+(t'_{2x}-t'_{1x})\times(t'_{0y}-t'_{2y}) \tag{4}$$

In order for q' be in the interior of t', the following condition must be met: $0<\lambda_i<1$, for $i=0,1,2$. The degenerate cases for this test happen when $det=0$ (this means the normal of the original triangle t is parallel with respect to the plane $z=0$, and therefore t is a vertical triangle), when one of the λ_i is 1 (this means q' is on one of the vertices of t') or when one of the λ_i is 0 and the others are not 1 (q' is on one of the edges of t').

As mentioned in Section 3.3, a perturbed vertex will never be exactly below a vertical triangle, and therefore, if $det = 0$, then *isOnProj*(t, q) should return false. Therefore, the only special cases that need to be considered are the ones that happen when at least one of the λ_i is either 0 or 1.

If q is perturbed creating the new point $q_\epsilon = (q_x + \epsilon, q_y + \epsilon^2, q_z + \epsilon^3)$, the projection q'_ϵ of q_ϵ onto the plane $z = 0$ will be equal to $(q_x + \epsilon, q_y + \epsilon^2)$. Replacing q' with q'_ϵ in Eqs. (1)–(3) we have the following barycentric coordinates of the perturbed points:

$$\lambda_{\epsilon_0} = \lambda_0 + \frac{(t'_{1y} - t'_{2y}) \times \epsilon + (t'_{2x} - t'_{1x}) \times \epsilon^2}{det} \tag{5}$$

$$\lambda_{\epsilon_1} = \lambda_1 + \frac{(t'_{2y} - t'_{0y}) \times \epsilon + (t'_{0x} - t'_{2x}) \times \epsilon^2}{det} \tag{6}$$

$$\lambda_{\epsilon_2} = 1 - \lambda_{\epsilon_0} - \lambda_{\epsilon_1} \tag{7}$$

As expected, because of SoS, λ_{ϵ_0} will never be 0 or 1: λ_{ϵ_0} is equal to λ_0 plus an expression containing an infinitesimal and this expression can never be zero since, otherwise, we would have $t'_{1y} = t'_{2y}$ and $t'_{2x} = t'_{1x}$ (which would imply in $det = 0$). This same observation is also valid for λ_{ϵ_1} and λ_{ϵ_2}.

Therefore, if *isOnProj*(t, q) is implemented to it return true if and only if $0 < \lambda_{\epsilon_i} < 1$, for $i = 1, 2, 3$, no degeneracy will happen. As it can be seen below, this implementation will be as efficient as an implementation that does not consider the special cases (and thus does not deal with the infinitesimals), satisfying requirement 3 of SoS mentioned in Section 3.3.

For example, consider the problem of verifying the following predicate $0 < \lambda_{\epsilon_0} < 1$:

- if $\lambda_0 \neq 0$ or 1 (which is expected to happen most of the time): it is clear that $0 < \lambda_{\epsilon_0} < 1 \Longleftrightarrow 0 < \lambda_0 < 1$.
- if $\lambda_0 = 0 : \lambda_{\epsilon_0} < 1$, and thus we only need to check if $0 < \lambda_{\epsilon_0}$. Considering that $det > 0$ (if $det < 0$ the value of this predicate needs to be negated), $0 < \lambda_{\epsilon_0} \Longleftrightarrow (t'_{1y} - t'_{2y}) \times \epsilon + (t'_{2x} - t'_{1x}) \times \epsilon^2 > 0$, which happens if $t'_{1y} > t'_{2y}$ or if $(t'_{1y} = t'_{2y})$ *and* $(t'_{2x} > t'_{1x})$ (since $\epsilon^2 \lll \epsilon$).
- if $\lambda_0 = 1 : \lambda_{\epsilon_0} > 0$, and thus we only need to check if $\lambda_{\epsilon_0} < 1$. Considering that $det > 0$ (again, if $det < 0$ the value of this predicate needs to be negated), $\lambda_{\epsilon_0} < 1 \Longleftrightarrow (t'_{1y} - t'_{2y}) \times \epsilon + (t'_{2x} - t'_{1x}) \times \epsilon^2 < 0$, which happens if $t'_{1y} < t'_{2y}$ or if $(t'_{1y} = t'_{2y})$ *and* $(t'_{2x} < t'_{1x})$.

This same strategy can be used to implement *isAbove*(t, q) and *isBelow*(t, t', q). Besides the three functions mentioned in this section, the only other step of PinMesh that deals with the coordinates of the query points is the operation of determining in what uniform grid cell c a query point q is. The only possibility of degeneracy in this operation happens when q is exactly on the border of a cell. This case is treated by considering that the point is in the cell with greatest index (e.g., a point on the border between cell 8 and cell 9 is considered to be in cell 9), which is consistent with the perturbation presented in this section (that adds positive infinitesimals to all coordinates).

4. Experimental evaluation

We implemented PinMesh in C++ using GMPXX [11] to provide multiple precision rational numbers and OpenMP to provide shared memory parallel programming constructors. Our implementation was compiled using g++ 4.9.3 (with the -O3 optimization flag) and tested on a workstation with a dual 8-core Xeon 3.1 GHz E5-2687 hyperthreading processors, 128 GiB of RAM and running the Linux Mint 17 operating system. Since PinMesh is parallel, unless otherwise noted it was configured to use 16 threads.

Table 1
Test dataset details.

Dataset	Source	Creator	Vertices	Triangles
Horse	Georgia Tech	–	48,485	96,966
Armadillo	Stanford	–	172,974	345,944
Hand	Georgia Tech	–	327,323	654,666
Pierrot	AIM@SHAPE	Frank_terHaar	443,805	887,606
Chinese dragon	AIM@SHAPE	Laurent_Saboret	655,980	1,311,956
Rolling stage	AIM@SHAPE	INRIA	660,267	1,320,558
Buddha	AIM@SHAPE	VCG-ISTI	719,553	1,439,102
Ramesses	AIM@SHAPE	Marco_Attene	826,266	1,652,528
Elephant	AIM@SHAPE	ISTI	1,512,290	3,024,588
Neptune	AIM@SHAPE	Laurent_Saboret	2,003,932	4,007,872
6 Materials	–	–	6,378,288	12,756,604
12 Materials	–	–	12,756,576	25,513,208
24 Materials	–	–	25,513,152	51,026,416

PinMesh was compared with RCT, the sequential point location algorithm proposed by Liu et al. [16]. Since RCT is based on floating-point arithmetic, it does not always locate points correctly. As far as we know, RCT is the most efficient available point location algorithm, and is much faster than other algorithms such as in CGAL and the AABB-tree-based algorithm proposed by Baerentzen et al. [3]. RCT's C++ source code was kindly made available by the authors [16], and so we were able to compile and run RCT using the same platform as PinMesh.

We used 13 datasets, sized from one hundred thousand triangles to fifty million triangles; see Table 1. Some were downloaded from the Stanford Scanning Repository [22], others from Georgia Tech's Large Geometric Model Archive [15], and others from the AIM@SHAPE-VISIONAIR Shape Repository [1]. Table 1 also includes the creator of each model downloaded from the AIM@SHAPE-VISIONAIR Shape Repository. Fig. 7 illustrates some of these meshes.

The ten smallest meshes are single-material, containing only one very large polyhedron. The three largest ones are multi-material, containing many polyhedra. The 6 Materials dataset was created by joining the six largest single-material meshes used in the experiments. The datasets with 12 and 24 Materials, on the other hand, were generated by joining, respectively, 2 and 4 copies of each object used in the 6 Materials dataset.

The fact that we can process the largest datasets from three different repositories lends confidence that pathologies that might execute very slowly will rarely occur. Even though these examples exhibit quite a non-uniform distribution of triangles, so that most of our cells are empty, PinMesh is still very fast.

4.1. Correctness evaluation

PinMesh is simple enough that its correctness is more obvious than would be the case with a more complicated algorithm, such as a topological sweep line. In addition, two strategies were used to verify the correctness of our implementation.

1. Points randomly positioned in the bounding-box of the objects were queried using PinMesh and RCT. Since it is improbable that point location queries with random points are incorrectly computed (because of floating point errors or special cases not treated), it is expected that the results obtained by PinMesh are equal to the ones obtained by RCT.
2. Experiments with query points positioned to represent special cases were used to test if PinMesh correctly located the points.

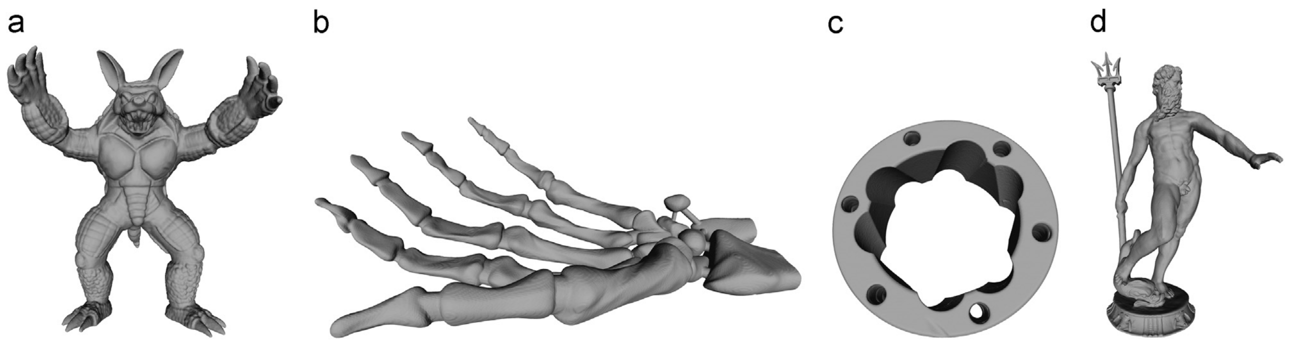

Fig. 7. Some test datasets (rendered with MeshLab): (a) Armadillo, (b) Hand, (c) Rolling Stage, and (d) Neptune.

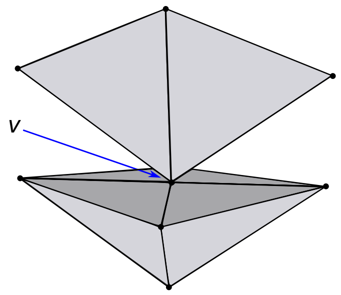

Fig. 8. A messy degenerate special case, handled correctly only by PinMesh.

Our experiments evaluated millions of queries; PinMesh correctly handled all of them. Since we are carefully treating all the singularities using SoS and, because of the exact arithmetic, PinMesh does not have any roundoff error, we believe that it can correctly handle any valid input.

We also generated a mesh with a messy special case to challenge PinMesh; see Fig. 8. It contains two upside-down pyramids of height 10, representing two polyhedra: 1 on the bottom and 2 on the top. The two pyramids intersect at the vertex $v = (0, 0, 10)$. v is on 8 different triangles.

Consider a query point q at $(0, 0, 9)$, directly below v inside polyhedron 1. The vertical ray used by PinMesh will intersect the mesh at v. Of those 8 triangles, the ray must choose one of the 4 triangles bounding polyhedron 1. PinMesh successfully handled this special case. RCT, on the other hand, was not able to correctly compute the polyhedron containing q for the following reason: in this mesh, RCT will try to find the closest triangle to q. Since all the 8 triangles containing v are at the same distance from q (that is, they are the closest ones to q) and since RCT does not perform any treatment to disambiguate this computation, the first of these 8 triangles found by the algorithm is used to compute q's position. If the triangles bounding the polyhedron 2 are stored before the triangles bounding the polyhedron 1, RCT will choose a triangle from polyhedron 2 to locate q, and thus return an incorrect output.

4.2. Performance evaluation

For each test dataset, we created a set of query points containing 500 thousand points randomly and uniformly distributed inside the mesh and 500 thousand points randomly distributed outside the mesh, but inside its bounding-box. These same points were used to evaluate the performance of RCT and PinMesh. Our reported running times represent the average wall-clock (elapsed) time of 10 runs.

The first level of PinMesh's uniform grid always had 64^3 cells. Each cell containing 1 or more triangles was refined. (1-triangle cells were refined to increase the fraction of empty cells, which are faster to process). The resolution of the second level grid was chosen such that the expected number of triangles per cell is 0.0005, making the number of cells on each side of each second level grid $\sqrt[3]{t/(64^3 \times 0.0005)}$, where t is the number of triangles in the mesh.

Table 2
Preprocessing and query times for PinMesh and RCT on 13 test datasets. N_t is the number of triangles in each dataset. G_2 is the resolution of the second level uniform grid. T_p is the preprocessing time, in elapsed seconds, T_q is the query time per point, in elapsed microseconds.

Mesh	$N_t \times 10^3$	G_2	PinMesh		RCT	
			T_p (s)	T_q (μs)	T_p (s)	T_q (μs)
Horse	97	9^3	0.30	0.45	0.42	0.84
Armadillo	346	14^3	0.86	0.35	1.88	1.07
Hand	655	17^3	0.78	0.47	3.64	1.78
Pierrot	888	19^3	2.83	0.27	5.24	2.07
R.Stage	1312	22^3	3.73	0.30	8.27	2.05
C.Dragon	1321	22^3	3.34	0.29	7.31	1.43
Buddha	1439	22^3	2.89	0.28	8.20	1.37
Ramesses	1653	23^3	1.84	0.30	10.72	1.04
Elephant	3025	28^3	3.38	0.25	20.50	1.93
Neptune	4008	31^3	3.07	0.36	28.88	1.85
6 Mat.	12,757	46^3	5.86	0.39	91.34	1.76
12 Mat.	25,513	58^3	7.31	0.56	187.18	1.97
24 Mat.	51,026	73^3	14.00	0.59	388.38	2.55

Table 2 compares the times spent by PinMesh and RCT. The total running-time (preprocessing plus time to perform one million queries) of PinMesh was up to 27 times faster than RCT. The speedup of PinMesh improves as the size of the mesh increases, which indicates that it scales better than RCT.

PinMesh's main advantage is in the preprocessing, which is the bottleneck of both algorithms even for 1 million queries. Because of a careful implementation of the uniform grid and of the use of parallel programming (possible because our algorithm is parallelizable), PinMesh constructs the index up to 28 times faster than RCT. The difference in the query times between RCT and PinMesh is smaller; PinMesh is up to 8 times faster.

Fig. 9 shows the processing time of PinMesh and RCT, as a function of the number of triangles in the mesh. The preprocessing step of both algorithms scales linearly, but PinMesh is much faster than RCT. PinMesh's query time changes very slowly as the number of triangles increases (being almost constant).

To stress-test the algorithms, the multi-material meshes were generated to be very unevenly distributed in the different dimensions. In the largest mesh, only 0.01% of the grid cells contain triangles, 0.08% are empty and completely inside the mesh, while the remaining 99.92% are empty and outside the mesh (but inside its bounding-box). Even this dataset was efficiently processed. This shows that sizing the grid so that most cells are empty

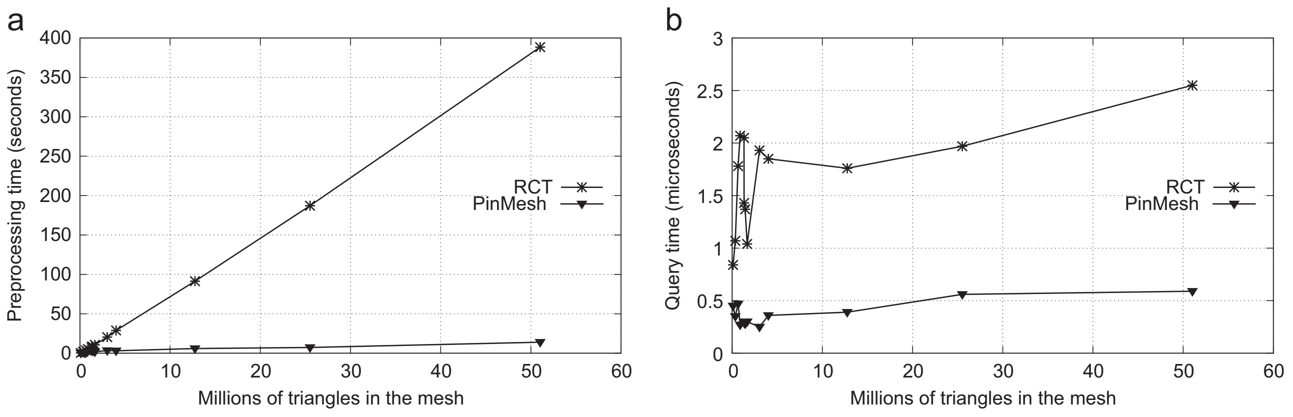

Fig. 9. Comparing (a) the preprocessing and (b) the average query times of PinMesh and RCT.

Table 3
Times of the main steps of PinMesh on the 24 Materials dataset using a varying number of threads. The query time is the average time per query (in μs).

Threads	Preprocessing (s)				Queries (μs)	
	Locate grid cells	Create 1 st level grid	Refine grid	Label empty grid cells	Compute query points' grids	Locate points
1	31.92	4.04	15.08	12.57	1.30	5.31
2	16.43	4.34	8.52	8.44	0.74	2.85
4	8.84	4.41	4.85	5.88	0.39	1.54
8	4.60	4.13	2.86	5.54	0.21	0.80
16	2.49	4.04	2.00	5.46	0.13	0.46

reduces the time. Also, since we can quickly process all these datasets of varying characteristics, datasets that are bad should be very unusual.

To evaluate PinMesh's parallel performance, we ran it with varying numbers of threads. Table 3 shows the time spent by the main steps of PinMesh during the processing of the 24 Materials dataset, the largest mesh used in the experiments.

Even when only 1 thread is used, PinMesh was able to preprocess the dataset faster than RCT. Indeed, while RCT sequentially spent 388 s to preprocess the 24 Materials mesh and 2.55 μs to perform each query, PinMesh processed it in 64 s and spent 6.61 μs per query when only one thread was used. Since the preprocessing time is usually much larger than the query time, the total time spent by RCT is only smaller than the total time spent by PinMesh running with 1 thread when more than 80 million points are queried in this mesh. When 4 or more threads were used, PinMesh was faster than RCT in both the preprocessing and query steps.

The slowest preprocessing steps of PinMesh with only one thread are the initial computation of the grid cell containing each vertex, which is slow because of the use of rationals, and the refinement of the grid. Luckily, they parallelize quite well.

When PinMesh runs with one thread, creating the first level uniform grid, is the fastest step. This was not parallelized since it consists basically in inserting the ids of the triangles into the relevant positions of the ragged array. This is very memory intensive, and would require synchronizations to run in parallel. As stated in Section 3.2, preliminary experiments showed that the overhead associated with these synchronizations is big compared to the time spent inserting the ids into the array.

PinMesh scales well as the number of threads increases from 1 to 8. For example, when the number of threads is increased from 1 to 2, the total running time is reduced by 41%. This scalability happens because the slowest steps are the ones that scale better. Another consideration is that the Xeon processor used in the tests has the Intel® Turbo Boost technology to increase the CPU frequency from the 3.1 GHz base up to 3.8 GHz. However, to keep from overheating the CPU, when more cores are active, less Turbo Boosting is allowed. Therefore, one cannot expect perfect scalability even in a completely parallelizable function.

When the number of threads increases from 8 to 16, on the other hand, the total running time is reduced by 20%. This smaller reduction happens mainly because of two reasons. First, because of Amdahl's law, the percentage of time spent performing operations that do not scale (such as memory allocations) and running the steps that were not implemented in parallel increases as the number of threads increases. Second, some of the steps are very memory intensive, and we believe that they saturate the processor's memory bus when 16 threads are used.

PinMesh, in general, uses more memory than RCT. For example, when the 24 Materials dataset was processed PinMesh used 23 GB of RAM while RCT used 14 GB. This larger memory footprint is mainly because PinMesh stores all the coordinates of the mesh and of the query points using arbitrary-precision rationals. Indeed, in this dataset, PinMesh uses 9.2 GB to store the coordinates. Furthermore, as mentioned in this paper, we perform several space–time tradeoffs.

5. Conclusion and future work

This paper presented PinMesh, an exact and efficient algorithm for quickly and correctly locating points in 3D meshes. PinMesh was carefully engineered to always handle point location queries correctly, even with inputs having many special cases. PinMesh's correctness results from a combination of four techniques: a 2-level uniform grid, rational number coordinates, Simulation of Simplicity, and parallelization. PinMesh was tested on large datasets from three different geometry repositories.

Although PinMesh is currently the fastest 3D point location algorithm (according to our experiments), it is possible that other exact arithmetic techniques might make PinMesh even faster, such as *Adaptive Precision Floating-Point Arithmetic* [21] and *interval arithmetic* [18]. One research topic would be to determine what is the necessary precision as the dataset size grows, and whether it is even possible to preprocess the dataset with a fixed precision, and be able to correctly handle a query point arbitrarily close to a dataset element. However, this might speed up determining which cell contains each vertex. Also, interval arithmetic could be used to identify the majority of vertices that are far enough from any cell boundary that roundoff error is not a problem there.

The efficient support to rational coordinates by PinMesh is also important in some applications where the data can only be represented using rationals. For example, vertices resulting from the intersection of faces or edges may not be exactly represented using floating-point numbers.

The next task, now underway, is extending these techniques to compute the intersection of two very large 3D big meshes, such as those that occur in Computational Fluid Dynamics.

PinMesh's C++ source code is freely available under GNU General Public License (GPL) for other researchers to use and extend at http://wrfranklin.org/Main/Software.

Acknowledgments

This research was partially supported by FAPEMIG, CAPES (Ciencia sem Fronteiras – grant 9085/13-0), CNPq, and NSF under grant IIS-1117277.

References

[1] AIM@SHAPE-VISIONAIR Shape Repository. AIM@SHAPE-VISIONAIR shape repository. ⟨http://visionair.ge.imati.cnr.it/⟩; 2016 [accessed on February 2016].
[2] Akman V, Franklin WR, Kankanhalli M, Narayanaswami C. Geometric computing and the uniform grid data technique. Comput Aid Des 1989;21(7):410–20.
[3] Baerentzen JA, Aanaes H. Signed distance computation using the angle weighted pseudonormal. IEEE Trans Vis Comput Graph 2005;11(3):243–53. http://dx.doi.org/10.1109/TVCG.2005.49.
[4] The CGAL Project. CGAL User and Reference Manual. CGAL Editorial Board; 4.8 ed.; 2016.
[5] Cucu L, Dragan M, Negru V, Mangu D. Three dimensional Delaunay triangulation using an uniform grid. In: The 11th European workshop on computational geometry. Linz, Austria: Universität Linz; 1995. p. 21–3.
[6] Edelsbrunner H, Mücke EP. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. ACM Trans Graph 1990;9(1):66–104.
[7] Feito FR, Torres JC. Inclusion test for general polyhedra. Comput Graph 1997;21(1):23–30. http://dx.doi.org/10.1016/S0097-8493(96)00067-2 ⟨http://www.sciencedirect.com/science/article/pii/S0097849396000672⟩.
[8] Franklin WR, Chandrasekhar N, Kankanhalli M, Seshan M, Akman V. Efficiency of uniform grids for intersection detection on serial and parallel machines. In: Magnenat-Thalmann N, Thalmann D, editors. New trends in computer graphics (Proceedings of computer graphics international'88). Geneva, Switzerland: Springer-Verlag; 1988. p. 288–97.
[9] Franklin WR, Sivaswami V, Sun D, Kankanhalli M, Narayanaswami C. Calculating the area of overlaid polygons without constructing the overlay. Cartogr Geogr Inf Syst 1994;21(2):81–9.
[10] Franklin WR. Pnpoly-point inclusion in polygon test. ⟨http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html⟩; 2006 [accessed on February 2016].
[11] Granlund T. the GMP development team. GNU MP: the GNU multiple precision arithmetic library; 6.0.0 ed. ⟨http://gmplib.org/⟩; 2014 [accessed on June 2015].
[12] Hopkins S, Healey RG. A parallel implementation of Franklin's uniform grid technique for line intersection detection on a large transputer array. In: Brassel K, Kishimoto H, editors. The 4th international symposium on spatial data handling, Zürich; 1990. p. 95–104.
[13] Kalojanov J, Billeter M, Slusallek P. Two-level grids for ray tracing on GPUs. Computer graphics forum; 2011. http://dx.doi.org/10.1111/j.1467-8659.2011.01862.x.
[14] Knuth DE. Surreal Numbers: how two ex-students turned on to pure mathematics and found total happiness: a mathematical novelette. Addison-Wesley; 1974.
[15] Large Geometric Model Archive. GIT large geometric model archive. ⟨http://www.cc.gatech.edu/projects/large_models/⟩; 2016 [accessed on February 2016].
[16] Liu J, Chen YQ, Maisog JM, Luta G. A new point containment test algorithm based on preprocessing and determining triangles. Comput Aid Des 2010;42(12):1143–50. http://dx.doi.org/10.1016/j.cad.2010.08.002.
[17] Magalhaes SVG, Andrade MVAA, Franklin WR, Li W. Fast exact parallel map overlay using a two-level uniform grid. In: Proceedings of the 4th ACM SIGSPATIAL international workshop on analytics for big geospatial data. BigSpatial '15. New York, NY, USA: ACM; 2015.
[18] Moore RE. Interval arithmetic and automatic error analysis in digital computing. Technical Report. DTIC Document; 1962.
[19] O'Rourke J. Computational geometry in C, 2nd ed. New York, NY, USA: Cambridge University Press; 1998. ISBN 0521640105.
[20] Ogayar CJ, Segura RJ, Feito FR. Point in solid strategies. Comput Graph 2005;29(4):616–24. http://dx.doi.org/10.1016/j.cag.2005.05.012.
[21] Shewchuk JR. Adaptive precision floatingpoint arithmetic and fast robust geometric predicates. Discrete Comput Geom 1997;18(3):305–63.
[22] Stanford Scanning Repository. The Stanford 3D scanning repository. ⟨http://graphics.stanford.edu/data/3Dscanrep/⟩; 2016 [accessed on February 2016].
[23] Wang W, Li J, Sun H, Wu E. Layer-based representation of polyhedrons for point containment tests. IEEE Trans Vis Comput Graph 2008;14(1):73–83. http://dx.doi.org/10.1109/TVCG.2007.70407.