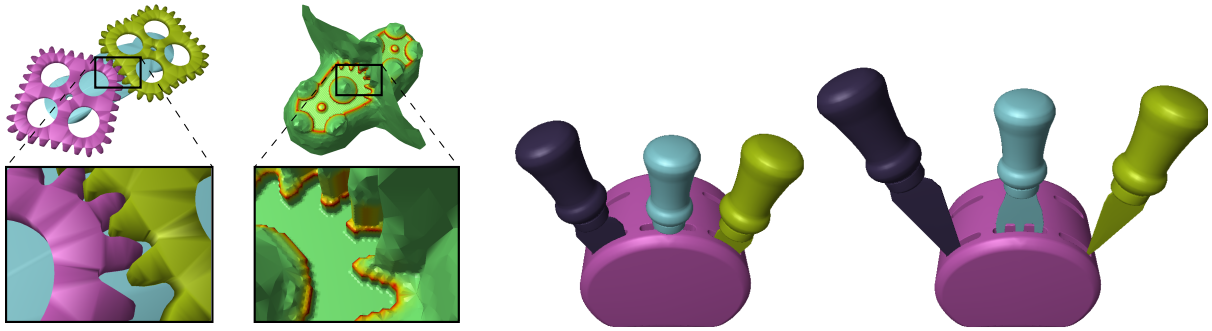
EUROGRAPHICS 2015 / O. Sorkine-Hornung and M. Wimmer
(Guest Editors)

Volume 34 (2015), Number 2

Approximating the Generalized Voronoi Diagram of Closely Spaced Objects

John Edwards[1] and Eric Daniel[2] and Valerio Pascucci[1] and Chandrajit Bajaj[3]

[1]Scientific Computing and Imaging Institute, University of Utah
[2]Google, Inc.
[3]The University of Texas at Austin



(a) (b) (c) (d)

Figure 1: Two example applications of the approximated generalized Voronoi diagram (GVD) computed by our novel, adaptive algorithm. Previous GVD methods require a gridded space of 2^{24} (gears dataset) and 2^{36} (knives dataset) voxels to resolve the closely spaced objects. (a) Two gears with regions of very tight spacing. (b) The GVD of the gears model. The surface is colored red in areas of very close tolerance. (c) Three butter knives in a wood block. To animate removal of the knives without intersecting the block requires extreme care because of close mesh spacing. (d) Intersection-free motion is guaranteed by computing motion vectors based on the GVD and allowing motion only within a Voronoi cell.

Abstract

We present an algorithm to compute an approximation of the generalized Voronoi diagram (GVD) on arbitrary collections of 2D or 3D geometric objects. In particular, we focus on datasets with closely spaced objects; GVD approximation is expensive and sometimes intractable on these datasets using previous algorithms. With our approach, the GVD can be computed using commodity hardware even on datasets with many, extremely tightly packed objects. Our approach is to subdivide the space with an octree that is represented with an adjacency structure. We then use a novel adaptive distance transform to compute the distance function on octree vertices. The computed distance field is sampled more densely in areas of close object spacing, enabling robust and parallelizable GVD surface generation. We demonstrate our method on a variety of data and show example applications of the GVD in 2D and 3D.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.5]: Computational Geometry and Object Modeling—Boundary representations Computer Graphics [I.3.6]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

The generalized Voronoi diagram (GVD) is an important structure that divides space into a complex of generalized Voronoi cells (GVCs) around objects. Similar to the ordinary Voronoi diagram, each GVC contains exactly one object, or site, and every point in the GVC is closer to its contained object than to any other object. The generalized Voronoi diagram is the boundary of the cell complex, and thus every point on the GVD is equidistant from two or more closest objects. Applications of the GVD range from motion path planning to GIS analysis to mosaicking.

Ordinary Voronoi diagrams have been studied extensively and efficient algorithms exist to compute them, but the GVD is difficult to compute analytically in general [BWY06, HIKL*99] and so the majority of approaches compute an approximation. Whereas most algorithms are efficient and robust on certain datasets, all algorithms to our knowledge require inordinate amounts of memory on datasets where objects are very closely spaced relative to the size of the domain. The failures occur because the space is uniformly gridded. In such approaches, voxel size must be small enough to resolve object spacings, and if two objects are very close to each other the number of voxels can become prohibitively large.

We present an algorithm to compute a GVD approximation on arbitrary datasets, including those with closely spaced objects. The approach applies a distance transform over an octree representation of the objects. Our octree, its associated data structure, and our distance transform are novel and optimized to GVD approximation. For the remainder of the paper, "GVD" will refer to the approximated Generalized Voronoi Diagram.

This paper demonstrates GVD computation on data beyond the computational abilities of previous algorithms, unlocking interesting and important applications. Our approach allows GVD-based proximity queries and other applications using a larger class of meaningful datasets.

Main contributions The three primary technical contributions described in the paper are as follows.

1. Most octree decompositions of objects resolve for object feature retention, but ours resolves only for object-object separation, which makes our subdivision computation largely independent of object complexity. Further, our octree data structure optimizes for octree vertex neighbor finding by storing cell vertices in an adjacency list rather than storing cells hierarchically.
2. Our distance transform is computed after the octree is built and uses a scheme that requires $O(N \log N + M)$ distance computations where N is the number of octree leaf cells and M is a measure of object complexity (e.g., number of polyhedron facets). Distances are computed on octree vertices with a conjectured error bound.
3. We trace out the GVD over the octree distance field using an efficient and parallelizable $O(N)$ algorithm. The GVD is guaranteed to separate each object into its own generalized Voronoi cell, i.e., any path from a point p on object S_i to a point q on object S_j must intersect the GVD, a guarantee that is not usually made by uniformly gridded methods.

We demonstrate various applications of the GVD in two and three dimensions, including motion path planning, proximity queries, and exploded diagrams.

Our GVD algorithm has four main steps: 1) build the octree over the set of objects; 2) compute distances on octree vertices using a wavefront expansion; 3) resolve ambiguous cells through further subdivision; and 4) compute the GVD surface by finding octree edges with differing end labels. After a discussion of related work, we discuss each step in detail and present applications.

2. Related work

Related work falls into two categories: algorithms that compute the GVD and algorithms that compute distance fields, many of which are adaptive.

Generalized Voronoi diagrams A theoretical framework for generalized Voronoi diagrams can be found in Boissonnat et al. [BWY06]. Ordinary Voronoi diagrams are well studied and efficient algorithms exist that compute them exactly [DBCVK08], but exact algorithms for the generalized Voronoi diagram are limited to a small set of special cases [Lee82, Kar04]. In an early work, Lavender et al. [LBD*92] define and compute GVDs over a set of solid models using an octree. Etzion and Rappoport [ER02] represent the GVD bisector symbolically for lazy evaluation, but are limited to sites that are polyhedra. Boada et al. [BCS02, BCMAS08] use an adaptive approach to GVD computation, but their algorithm is restricted to GVDs with connected regions and is inefficient for polyhedral objects with many facets. Two other works are adaptive [TT97, VO98] but are computationally expensive and are restricted to convex sites.

In recent years Voronoi diagram algorithms that take advantage of fast graphics hardware have become more common [CTMT10, FG06, HT05, RT07, SGGM06, SGG*06, HIKL*99, WLXZ08]. These algorithms are efficient and generalize well to the GVD, but most are limited to a subset of site types. More importantly, all of them use uniform grids and require an extraordinary number of voxels to resolve closely spaced objects (for example, Figs. 1c and 13 would require 2^{36} and 2^{48} voxels, respectively). To our knowledge, ours is the first fully adaptive algorithm that computes the generalized Voronoi diagram for arbitrary datasets.

Distance fields and octrees The GVD is a subset of the locus of distance field critical points, a property that we take advantage of. In that light, the GVD could be a post-processing step to any method that computes a distance field.

Distance transforms compute a distance field, but most are uniformly gridded [JBS06] and are thus no more suitable than GVD algorithms that use the GPU.

Two seminal works adaptively compute the Adaptive Distance Field (ADF) on octree vertices. Strain [Str99] fully resolves the octree everywhere on the object surface, and Frisken et al. [FPRJ00] resolve the octree fully only in areas of small local feature size. Both approaches are designed to retain features of a single object rather than resolving between multiple objects, as is required for GVD computation. Qu et al. [QZS*04] implement an energy-minimizing distance field algorithm that preserves features at the expense of efficiency. Many recent works on fast octree construction using the GPU are limited to point sites [BGPZ12, Kar12, ZGHG11]. Most octree approaches that support surfaces [BLD13, CNLE09, LK11, LH07] are designed for efficient rendering, and actual construction of the octree is implemented on the CPU.

Two works [BC08, PLKK10] implement the ADF using GPU parallelism to compute the distance value at sample points, but building the octree itself is done sequentially. Yin et al. [YLW11] compute the distance field entirely on the GPU using a bottom-up approach by initially subdividing into a complete octree, resulting in memory usage that is no better than using a uniform grid. A method by Kim and Liu [KL14] computes the octree and a BVH entirely on the GPU. However, octree construction is performed on barycenters of triangles, and so a leaf octree cell can have an arbitrary number of triangle intersections as long as it contains no more than one triangle's barycenter. We have found no GPU octree construction method that can resolve between objects.

3. Build octree

Our algorithm works in both 2D and 3D. Lacking a dimension-independent term, we use "octree" as a general term to refer to both quadtrees and octrees.

We construct the octree over a set of objects $S = \{S_i\}$ as a pre-processing step to computing a distance field. Whereas other methods construct the octree and distance field at the same time [FPRJ00, Str99], our decoupling allows us to optimize our octree construction implementation by temporarily converting polyhedra to integer-based representations, and using entirely integer arithmetic during octree construction.

Our octree data structure stores octree vertex adjacencies rather than an explicit cell hierarchy. In this paper, "vertex" will always refer to an octree vertex. Each vertex structure contains an object label, a closest point on an object, and references to its neighbors. We initialize the system with a single octree cell, represented by 2^D vertices, where D is the number of dimensions. We use our subdivision predicate `SHOULD_SUBDIVIDE(c)`, explained below, to determine whether to subdivide c. To subdivide cell c, we first create

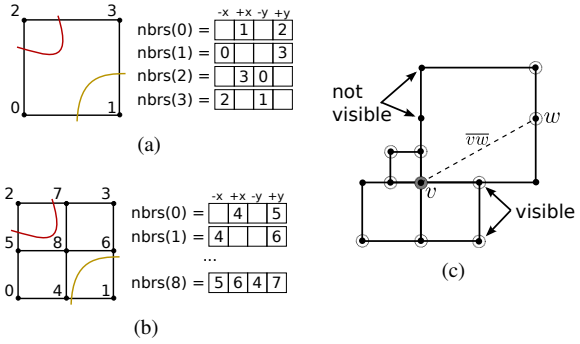


Figure 2: Quadtree (octree in 3D) cell representation. (a) In two dimensions, a quadtree cell is defined by four vertices. Each vertex maintains references to its neighbors with the *nbrs* array. Two intersecting objects are shown in red and yellow. (b) When a cell is subdivided, vertices are added and each vertex's neighbor references are updated. (c) v is shaded and vertices that are visible from v are circled. A vertex w is visible from v if w is a neighbor of v or if the line segment $\overline{vw}$ has no intersections with any octree cell boundaries.

$3^D - 2^D$ new vertices (in 3D, one vertex for each of the 12 edges, one for each of the 6 faces, and one for the center). We then assign vertex neighbors appropriately (see Figures 2a and 2b). After intersecting new subcells with objects that intersect c, we call `SHOULD_SUBDIVIDE` on each subcell and recursively subdivide if necessary.

Our adjacency structure is amenable to our most important operations, which are to find an edge neighbor for the `SHOULD_SUBDIVIDE` predicate, and finding visible vertices for the wavefront propagation. Finding an edge neighbor is $O(1)$ with our octree representation. Gargantini's [Gar82] data structure has logarithmic neighbor finding, and Frisken and Perry's [FP02], which has fast neighbor finding like ours, requires significant extra storage. Computational complexity of finding visible vertices is asymptotically equal to the number of visible vertices and is discussed in Section 4. Our nonhierarchical representation is not suitable for general purposes, however, because the point location operation is $O(N)$. If the need for point location arises, then our data structure can be converted to a traditional hierarchical representation in $O(N \log N)$ time.

Previous algorithms ensure that object features are resolved well by subdividing all nonempty cells up to a maximum level. More sophisticated feature resolution predicates have also been used (e.g., the bilinear interpolation test of Frisken et al. [FPRJ00]). Our subdivision predicate is different in that we are primarily interested in resolving between objects. Define a face-neighbor of cell c to be a cell that shares a face with c. The predicate `SHOULD_SUBDIVIDE(c)` works as follows. Subdivide c if (1) c intersects more than one object, or (2) a face-neighbor

(a) (b) (c)

Figure 3: Quadtree, GVD, and distance field of intersecting, embedded, and nonmanifold 2D objects. (a) The quadtree is subdivided only far enough so that there is a one cell buffer between objects and so that ambiguous cells are resolved. Object self-intersections and nonmanifold points have no effect on quadtreetree depth (solid arrow), but intersections between objects are subdivided to the maximum level (dotted arrow). (b) Computed GVD. (c) The distance field computed from the quadtree vertices.

of c intersects with a different object. Test (1) ensures that every leaf cell after construction will intersect at most one object, and test (2) ensures that objects will be separated from each other by at least one empty buffer cell. One advantage of `SHOULD_SUBDIVIDE(c)` is that octree complexity becomes independent of object complexity. As shown in Figure 3a, even object self-intersections or nonmanifold points have no effect on octree depth, whereas the octree would be fully subdivided in those areas if using a conventional subdivision predicate.

4. Distance transform

Our distance transform is inspired by that of Breen et al. [BMW98], with the fundamental difference being that we compute using an octree rather than a uniform grid. Each octree vertex has two properties – a label and a closest point. An octree vertex is empty if all incident cells are empty. Define the euclidean distance between two points $\text{dist}(a,b)$ to be infinity if either a or b is `null` and let $\{S_i\}$ be the set of objects intersecting octree cells incident to a vertex v. The two main steps of distance computation are initialization of nonempty vertices and wavefront expansion. Algorithm `COMPUTE_DISTANCES` comprises both steps.

Algorithm `COMPUTE_DISTANCES` initializes the point assignments of nonempty vertices (Figures 4b and 4d). All closest points $point[v]$ (ties are broken arbitrarily) are stored as an array and each vertex maintains an index into the array, taking advantage of the fact that roughly half of computed closest points are shared among multiple vertices. The closest points are computed exactly: if $\alpha(v)$ is the distance from v to its closest neighbor, then only surface points in cells within $\sqrt{D}\alpha(v)$ of the vertex need be searched, making the loop an $O(N+M)$ operation where N is the number of octree leaf cells and M is a measure of object complexity (e.g., number of triangles). All octree cells are then added to the priority queue, an $O(N)$ or $O(N \log N)$ operation, depending on the type of heap used for the queue. It then iterates over vertex priority queue `V` in multiple expanding wavefronts, which are similar in behavior to Dijkstra shortest-cost path wavefronts in that the priority queue is sorted on distance to the nearest object. The vertex v at the front of the queue is the closest vertex to an object among all vertices in the queue, modulo an error term discussed below. The assigned closest point p is pushed to all vertices that are visible from v. A vertex w is visible from v if w is a neighbor of v or if the intersection of the line segment $\overline{vw}$ with the edges of the octree is empty (Figure 2c). Visible vertices are found efficiently using our vertex adjacency data structure. In 2D, a simple walk around the cell is sufficient. For example, in Figure 2c, a walk is done beginning from vertex v in the positive x direction. At each vertex, the walk turns left if possible, otherwise it continues forward. Once the walk returns to v, the visible vertices in the upper-right cell are found, and walks are performed for the other three cells. The complexity of the walk is $O(m)$, where m is the number of visible vertices.

The number of vertices that are visible from v is dependent on gradation. Let c_1 and c_2 be adjacent cells, and let $L(c)$ be the octree level of cell c. Without loss of generality,

Algorithm 1: COMPUTE_DISTANCES

```
Input: Octree
// Initialization
foreach vertex v in Octree do
    if v is nonempty then
        point[v] := closest point on {S_i} to v
        label[v] := i
    end
    else
        point[v] := null
    end
end
V := the set of all vertices in Octree
// Wavefront expansion
while V is not empty do
    v := vertex in V with minimum dist(v, point[v])
    remove v from V
    foreach visible vertex a from v do
        if dist(a, point[v]) < dist(a, point[a]) then
            point[a] := point[v]
            label[a] := label[v]
            reorder a in V
        end
    end
end
```

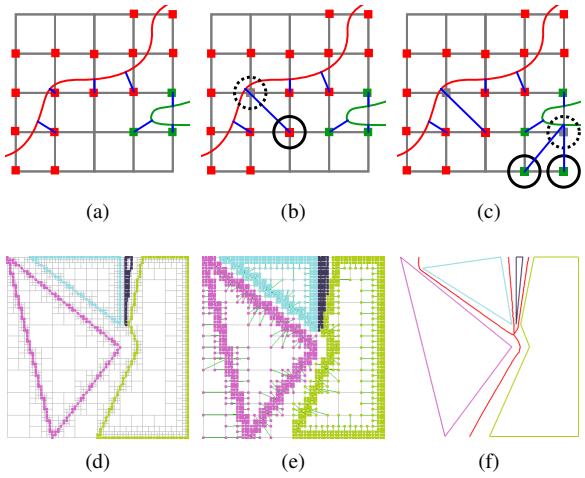


Figure 4: Wavefront expansion. (a) Portions of the surfaces of two objects. Assign every nonempty vertex an exact closest point and add the vertex to the wavefront priority queue. (b) Pop the top priority vertex (dotted circle) and push its closest point to its neighbors (circle). (c) Next iteration. (d) Initialized wavefront (as in (b)). (e) Expanded wavefront. Green lines connect quadtree vertices to computed closest points. (f) The GVD is shown in red.

let $L(c_1) > L(c_2)$. Gradation between adjacent cells c_1 and c_2 is defined to be $L(c_1) - L(c_2)$. Let g be the maximum gradation of any two adjacent cells. Then the number of visible vertices is $O(2^{g(D-1)})$. To see this, consider the 2D vertex in Figure 2c. Let the upper-right cell in the figure be cell c. v has two "opposite" edges through cell c, where an opposite edge is one that is not incident to v. v has four incident cells, for a total of eight opposite edges. Let each incident cell be maximally graded, such that each opposite edge is incident to 2^g cells. Then there are $(8)(2^{g(D-1)}) = O(2^g)$ visible vertices. Derivation in the 3D case is similar. If the number of visible vertices becomes a significant factor in practice, then the octree can be constructed with bounded gradation (similar to the approach of Strain [Str99]), thereby bounding the number of visible vertices with a constant.

Our algorithm approximates distances on empty vertices, and our empirical tests suggest an error bound of $\frac{\varepsilon(v)}{\delta(v)} \leq \frac{1}{2}$, where $e(v)$ is the error at a vertex v and $\delta(v)$ is the distance from v to the nearest point on S, implying that the distance transform is a $\frac{3}{2}$-approximation. The error occurs because only closest points assigned in the wavefront initialization are propagated in the wavefront expansion. Distance transform time complexity is $O(N+M)$ for the initialization step, $\leq O(N \log N)$ for the priority queue initialization, and $O(2^{g(D-1)} N \log N)$ for the wavefront expansion, where the reordering of a in V provides the $\log N$ factor. Thus, total time complexity of the transform is $O(2^{g(D-1)} N \log N + M)$, or $O(N \log N + M)$ if gradation is bounded by a constant.

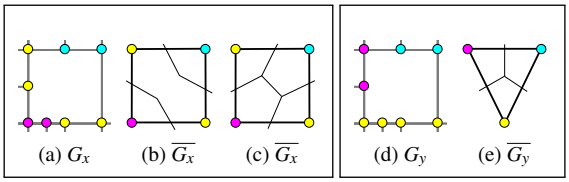


Figure 5: Ambiguous cells are detected by collapsing edges with identically labeled vertices. (a) Cell x is ambiguous because $\overline{G_x}$ is not a simplex. (b)-(c) Two possible GVD topologies in cell x are shown. (d) Cell y is unambigous. (e) Simplex $\overline{G_y}$ induces a unique GVD topology.

Although not designed specifically to do so, our distance-assigned octree vertices can be used to compute distance values of arbitrary points in the domain. Given a point p and its containing leaf octree cell c, choose the closest point among the closest points of vertices on c. That is, $point[p] = \arg\min_{point[v \in c]} \text{dist}(p, point[v])$. The assigned distance is $|point[p] - p|$. See Figure 3c for a distance field example.

5. Resolve ambiguities

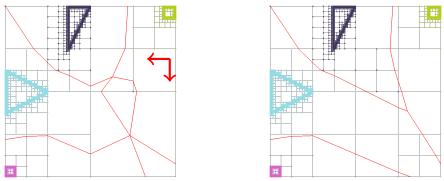
Our distance function is built over a discretized space that can potentially yield ambiguities in the topology of the GVD. We call an octree cell ambiguous if the topology of the GVD in the cell is ambiguous. We test for cell ambiguity as follows. Let G be an embedded graph with vertices and edges corresponding to octree vertices and edges, respectively. Let G_c be the intersection of G with octree cell c. See Figure 5. Collapse all edges in G_c that have endpoints with identical labels. Call the new graph $\overline{G_c}$. Cell c is unambiguous if $\overline{G_c}$ is a $(\leq D)$-dimensional simplex.

Proof Let $M = \{M_i\}$ be the set of generalized Voronoi cells (GVCs) that intersect with c. Because every vertex in a simplex shares an edge with every other vertex, the topology of the GVD in c is given by $M_i \cap M_j \cap c \neq \emptyset$. □

Ambiguous cells are subdivided recursively until the ambiguity is resolved or a threshold subdivision level is reached (Figure 6). If the threshold is reached then the topology is decided as described in Section 6.

6. Compute GVD surface

With the distance function in place we compute the generalized Voronoi diagram, or the set of all points that have at least two closest points with differing labels, using an algorithm inspired by marching cubes [LC87]. We store the GVD as sets of simplices (i.e. line segments in 2D, triangles in 3D), one set per object, representing the boundary of the



(a) (b)

Figure 6: The effect of ambiguous cells and how they are removed. (a) The cells indicated by the arrows are ambiguous, causing the GVD to be topologically incorrect. (b) The ambiguities are resolved by recursive subdivision.

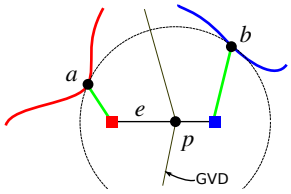


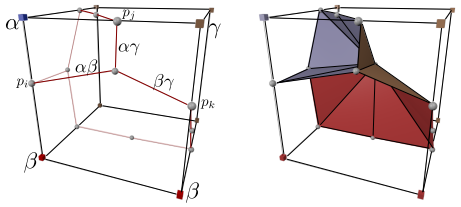
Figure 7: Computing the intersection of the GVD with quadtree edge e. a and b are closest points on objects, and p is the intersection of edge e with the GVD.

generalized Voronoi cell (GVC) for that object. Each simplex is stored twice, once for each GVC it borders, oriented toward the inside of the GVC.

We first compute which edges of the octree intersect the GVD by considering each octree edge e with endpoint vertices (e_0, e_1). Let $\mathscr{L}(v)$ be the label of octree vertex v. If $\mathscr{L}(e_0) \neq \mathscr{L}(e_1)$, then e intersects the GVD. Let $a = point[e_0]$ and $b = point[e_1]$. Assume without loss of generality that e is aligned with the x-axis. Cases of alignment with y and z are similar. We seek point $p = (x, y, z) \in e$ such that $\text{dist}(a, p) = \text{dist}(b, p)$ (see Figure 7). In our euclidean setting this reduces to

$$x = \frac{2y(a_y - b_y) + 2z(a_z - b_z) + b^T b - a^T a}{2(b_x - a_x)} \qquad (1)$$

Once all edge bisectors for a cell are calculated, we fit simplices (Figure 8). Suppose bisector p_i lies on edge e^i. We connect bisector points on a 2D face f by creating line segments between each bisector $p_i \in f$ and the centroid of all face bisectors $r_f = \sum p_i / |\{p_i\}|$. If the 2D cell is unambiguous there will be at most three bisectors. In the case that the cell is subdivided to the threshold level without resolving ambiguity then the centroid r_f becomes an interface to four or more generalized Voronoi cells. Other approaches to ambiguity resolution include [Nat94, NH91, SB07]. We label each line segment by the labels of the vertices adjacent to p_i, that is, $\mathscr{L}(p_i) = \alpha\beta$ where $\mathscr{L}(e_0^i) = \alpha$ and $\mathscr{L}(e_1^i) = \beta$. If we are computing a 2D GVD then we are done.



(a) (b)

Figure 8: GVD surface generation. (a) The 2D algorithm creates GVD edges from bisectors $\{p_i\}$ to the centroid. Each new GVD edge is given the two labels of the incident octree edge. (b) After finding the 2D GVD on its faces, the 3D cell fits triangles from 2D edges to the centroid.

In 3D, given a cell c, we first find the line segment complex of each 2D face c using the 2D algorithm and union all segments into a set C_c. We then form triangles from each segment in C_c to the 3D centroid r_c of the bisectors (Figure 8b). Triangle t_i has vertices (p_i, r_f, r_c). We maintain a set T_α of triangles assigned to the GVC of the object with label α. To determine the orientation we use the normal n_{t_i} and the vertex $v_{p_i}^\alpha$ adjacent to p_i with label α. If $n_{t_i} \cdot (v_{p_i}^\alpha - r_c) < 0$, then we invert t_i before adding it to T_α, and similarly with T_β. The resulting GVD is water tight and each GVC is orientable.

The GVD surfacing algorithm is parallelizable, as the simplices in each octree cell are computed independently of all other octree cells.

7. Results and applications

Our implementation[†] of the algorithm supports polygons and triangulated objects, and our wavefront initialization step is implemented on the GPU using OpenCL. All tests were run on a MacBook Pro laptop with a dual-core 2.9 GHz processor, 8 GB memory, and Intel HD 4000 graphics card. Figure 10 shows our implementation of the GVD computation pipeline, and Figure 11 shows the computed GVD on a more challenging dataset. We compare our method with other work and then show examples in three application settings: path planning, proximity queries, and exploded diagrams.

7.1. Comparison to other methods

Our GVD computations use commodity hardware, even on the most challenging datasets. Uniformly gridded approaches [CTMT10, FG06, HT05, RT07, SGGM06, SGG*06,

[†] Source code is available at http://cedmav.org/research/project/33-gvds.html.

HIKL*99, WLXZ08, YLW11] require 2^{Dn} voxels, where D is the dimension and n is the number of octree levels needed to resolve objects. Table 1 shows our results on datasets that would require prohibitive numbers of voxels using uniformly gridded methods.

In addition to datasets with closely spaced objects, our algorithm works on intersecting and embedded objects, non-manifold objects, and objects with multiple connected components (see Figures 3a, 3b, and 12), which are not supported by Boada et al. [BCMAS08].

We ran timing comparisons of our method with Laine's algorithm [LK11], which is a sparse voxel octree (SVO) method [BLD13, BC08, CNLE09, LH07, Str99]. Laine's algorithm (and other SVO methods) subdivide octree cells in areas of object complexity, whereas our algorithm subdivides in areas of inter-object proximity. In other words, previous methods model the objects, while our approach models the space between objects. Because the two methods compute different octrees, a direct timing comparison is not meaningful, so we compared the number of seconds required for each megabyte of octree. From Figure 9 we see that the execution time of our algorithm is comparable to the state of the art SVO construction.

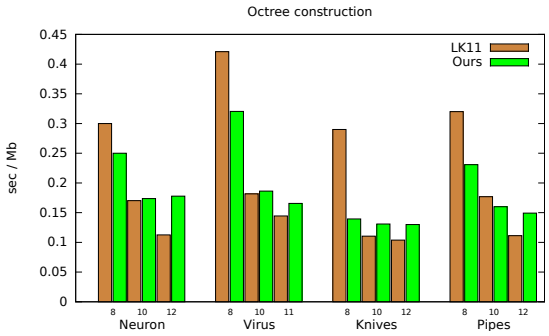


Figure 9: Comparison of our octree building to the sparse voxel octree method of Laine and Karras [LK11]. We used four datasets, each at three octree levels and compared seconds per Mb of octree memory used. Laine's algorithm was run on a Windows 7 desktop with 3.5 GHz processor, 16 GB memory, and NVIDIA GeForce GTX 660 graphics card.

7.2. Path planning

Motion planning is an important application in robotics and other fields [CD88, Thr98, HICK*00, VMS11]. Roadmap methods of path finding [MS07, RP12] retract the set of feasible movements to a lower-dimensional space, often using the Generalized Voronoi Diagram as the retraction. Once the GVD is available, a variety of methods can be used for computation of the final path, from simple graph searches [HICK*00, MS07] to fast marching [GMAM06]. As noted

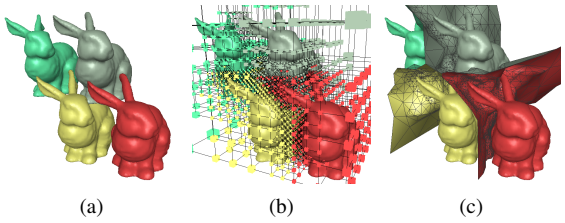


Figure 10: The method pipeline in 3D. (a) Four bunnies placed in proximity. (b) The octree with appropriately-labeled vertices. (c) The constructed GVD.

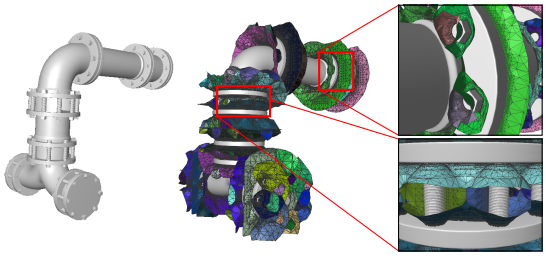


Figure 11: A pipes dataset with 392 objects and 207K object triangles. The octree was subdivided to level 10 for a total of 1.8m octree cells. The GVD separates even the nuts from the bolts (upper right).

by Foskey et al [FGLM01], a shortcoming of this approach is the difficulty of computing the GVD.

We have augmented our GVD algorithm with a simple path finding implementation that uses Dijkstra's graph search algorithm in order to show that path planning is possible on difficult datasets. On simple datasets, our method is not competitive with uniform grid approaches [HICK*00, FGLM01, GMAM06], but our algorithm is robust on datasets with closely spaced objects, which would cause uniform grid approaches to fail. One such dataset is shown in Figure 13, where we computed the GVD for 470 2D objects with object spacings that vary widely. Computation took 2.0 seconds with the quadtree reaching level 24 and 140,680 cells (a uniform grid would require 2^{48} pixels to resolve the closest object spacings). Our GVD algorithm could be coupled with a more sophisticated path search algorithm, such as the method proposed by Garrido et al [GMAM06], for improved paths.

7.3. Proximity queries

Proximity queries are used in collision detection, object overlap, and object separation distance queries [LM03], and a wealth of algorithms are available in the literature. One class of proximity query uses distance fields or the GVD [HIZLM01, GRLM03, TKH*05]. As in path planning, these approaches rely primarily on uniform grids, making dis-

dataset	objects	object Δs ($\times 10^3$)	octree depth	octree cells ($\times 10^3$)	octree memory (Mb)	GVD (sec)	GVD Δs ($\times 10^3$)
Fig. 1b	3	7	8	54	3	0.9	83
Fig. 1c	4	15	12	146	9	3.9	232
Fig. 13	470	5	24	158	8	2.0	151
Fig. 14	448	4015	8	2716	151	195	8100
Fig. 15	35	1500	8	496	70	19	2700

Table 1: Table of octree/GVD computation statistics and timings on datasets that are unmanageable using other methods. Columns are: *objects* - the number of objects in the dataset; *object* Δs - the number of line segments (2D) or triangles (3D) of all objects in the dataset; *octree depth* - required octree depth in order to resolve objects; *octree cells* - total number of leaf octree cells; *octree memory* - amount of memory used by the octree; *GVD (sec)* - seconds to perform all steps of GVD computation; *GVD* Δs - number of line segments (2D) or triangles (3D) in the GVD.

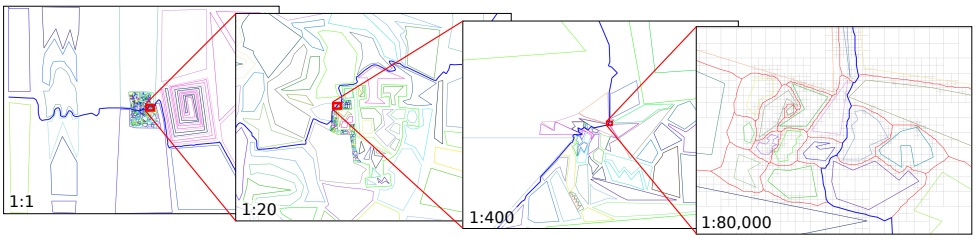
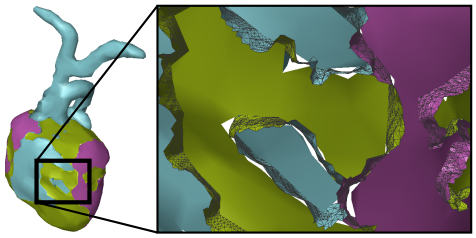


Figure 13: Path planning in 2D. We built a quadtree over hundreds of objects ranging in size and spacing over orders of magnitude. The quadtree reached level 24 before the closest spacings were resolved. The shortest-cost path between two points is shown in blue. The rightmost figure shows the quadtree in gray and GVD boundary complex in red at 80,000x magnification.



(a) (b)

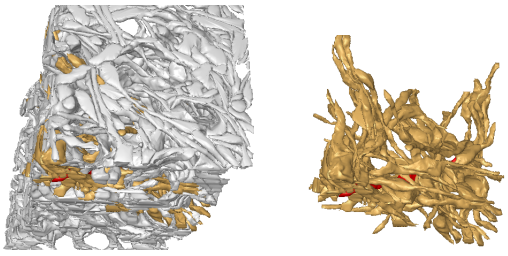
Figure 12: Heart defibrillation dataset showing support for multiple connected components. (a) The blue and purple objects have multiple components. (b) The GVD naturally handles multiple components of an object. Portions of the GVD are clipped away for visualization.

tance field computation expensive or impossible on difficult datasets. Our algorithm may enable these types of distance queries on difficult datasets.

We have implemented a form of topological proximity query suitable for our application of neuronal modeling, where a frequent query is to identify neuron pairs that have high probability of forming a synaptic connection [MHS*10]. In Figure 14 we show a portion of a neuron of interest in red, which is difficult to see because of tight packing. Rather than using a typical proximity query, that of finding neurons within a threshold distance of the neuron of interest, we take a topological approach: find all neurons whose Generalized Voronoi Cells (GVCs) share a boundary with the GVC of the red neuron. By taking the dual of the GVD we can perform this query efficiently, giving us a fast and robust pre-filter to finding synaptic pairs. Beyond our neuronal modeling application, Figure 14 shows that this type of query may also find usefulness in visualization of occluded objects.

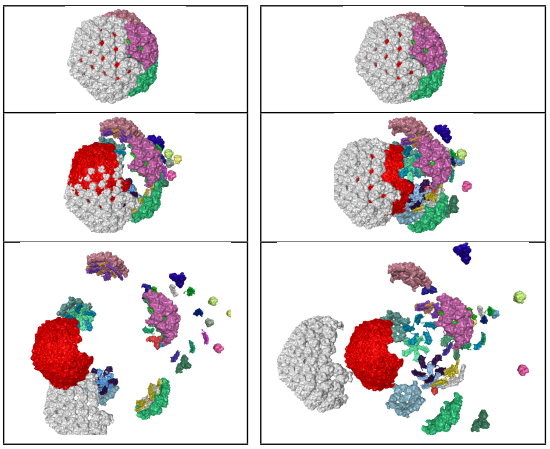
7.4. Exploded diagrams

An effective way of visualizing multiple objects in proximity is to "explode" the objects away from each other in a meaningful way that retains some of the spatial coherence of the collection. Exploded diagrams are typically used in CAD applications, but their usefulness extends to other applications, such as molecular modeling. Virus molecules are composed of hundreds or thousands of atoms, often in symmetric structure that can be meaningfully segmented into constituent collections of atoms. Often many regions are occluded from view. Furthermore, molecules are often layered radially from the center, forming shells. In the case of radial layering, one need only translate objects radially away from the center to create an exploded diagram, but this ap-



(a) (b)

Figure 14: GVD-based proximity queries on neuron data give a way to discover which neurons potentially synapse with another neuron. (a) In a dense collection of 450 neurons, the neuron of interest is colored red. (b) Neurons in the 1-ring of the GVD dual graph are colored brown.



(a) (b)

Figure 15: Explosion diagrams of the rice dwarf virus. (a) The vectors that objects travel along are computed using object centroids. Objects travel in nonintuitive directions. (b) Travel vectors are computed using triangles of the GVD boundary complex. The directions of travel are intuitive and separate the objects in a meaningful way.

proach is not effective if we wish to explode objects away from an anchor object that is not near the center. A naive approach to creating an exploded diagram is to choose a primary object C and move all objects along a vector derived from the object centroids. That is, given a object D, move $D \leftarrow D + \lambda(\mathscr{C}(D) - \mathscr{C}(C))$ where $\mathscr{C}(C)$ is the centroid of C and λ is a speed constant. In many cases, the centroid of D lies in a very different direction relative to $\mathscr{C}(C)$ than where D should intuitively travel (Figure 15a).

Our approach is to utilize the GVD boundary complex to compute directions of travel for each object (Figure 15b). Let C be the anchor object and let K be the dual graph of the GVD as described in Section 7.3. Further, define T_{CD} as the set of triangles adjacent to both GVC_C and GVC_D with normals oriented toward GVC_D. $\mathscr{R}_C(i)$ is the set of objects in the i-ring of C and $A_{CD} = \sum_{t \in T_{CD}} A_t$ is the summed areas of triangles in T_{CD}. Using graph K, if $D \in \mathscr{R}_C(1)$ then $D \leftarrow D + \mathscr{D}_1(D)$ where

$$\mathscr{D}_1(D) = \frac{\sum_{t \in T_{CD}} n_t A_t}{A_{CD}}$$

Since the $(i > 1)$-ring neighbors of C have no direct interface to C, they are moved in an average direction of the $(i-1)$-ring neighbors that are adjacent to D. That is, $D \leftarrow D + \mathscr{D}_i(D)$ where

$$\mathscr{D}_i(D) = \frac{\sum_{N \in \mathscr{R}_c(i) \cap \mathscr{R}_D(1)} \mathscr{D}_{i-1}(N) A_{DN}}{\sum_{N \in \mathscr{R}_c(i) \cap \mathscr{R}_D(1)} A_{DN}}$$

8. Conclusions

We have presented and demonstrated the effectiveness of a novel generalized Voronoi diagram algorithm, which includes an octree subdivision algorithm, octree data structure, distance transform, and GVD surfacing algorithm. We have also shown, in addition to the popular motion path planning, important applications of the GVD. Our method opens the door to investigation of other applications that might benefit from a GVD algorithm specifically tailored to collections of objects that are closely spaced.

Our algorithm is general; our implementation supports polygons and triangulated objects, but the algorithm supports curved objects equally as well. Our approach is particularly suited to objects for which a point-object distance computation is expensive since initialization of the wavefront is the only step that requires the point-object distance operation.

With the proven usefulness of the ordinary Voronoi diagram and the growing uses of the generalized Voronoi diagram, the algorithm presented in this paper fills a need for a practical GVD algorithm that supports previously unmanageable datasets – those with tightly packed objects. Interestingly, it is often applications using these very datasets that are in greatest need of the GVD. For example, traditional path-planning is straightforward unless the objects are tightly packed; finding regions of close tolerance is unnecessary unless close tolerances, in fact, exist. These applications and others can now be more fully explored with the availability of our adaptive GVD algorithm.

Acknowledgements

Thanks to Kristen Harris for use of the neuronal data and Jonathan Bronson for the heart data. The work of

JE and VP was supported in part by NSF IIS-1314896, NSF ACI-0904631, DOE/NEUP 120341, DOE/UV-CDAT DESC0006872, DOE/Codesign P01180734, DOE/SciDAC DESC0007446, DOE/PIPER DESC0010498, and DOE/CCMSC DENA0002375. This work initiated at the University of Texas when JE, ED and CB were supported in part by NIH contract R01-EB00487, NSF Grant OCI-1216701 and SNL contract 1439100.

References

[BC08] BASTOS T., CELES W.: Gpu-accelerated adaptively sampled distance fields. In *Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on* (2008), IEEE, pp. 171–178. 3, 7

[BCMAS08] BOADA I., COLL N., MADERN N., ANTONI SELLARES J.: Approximations of 2d and 3d generalized voronoi diagrams. *International Journal of Computer Mathematics 85*, 7 (2008), 1003–1022. 2, 7

[BCS02] BOADA I., COLL N., SELLARES J.: The voronoi-quadtree: construction and visualization. *Eurographics 2002 Short Presentation* (2002), 349–355. 2

[BGPZ12] BÉDORF J., GABUROV E., PORTEGIES ZWART S.: A sparse octree gravitational< i> n</i>-body code that runs entirely on the gpu processor. *Journal of Computational Physics 231*, 7 (2012), 2825–2839. 3

[BLD13] BAERT J., LAGAE A., DUTRÉ P.: Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), ACM, pp. 27–32. 3, 7

[BMW98] BREEN D. E., MAUCH S., WHITAKER R. T.: 3d scan conversion of csg models into distance volumes. In *Volume Visualization, 1998. IEEE Symposium on* (1998), IEEE, pp. 7–14. 4

[BWY06] BOISSONNAT J.-D., WORMSER C., YVINEC M.: Curved voronoi diagrams. In *Effective Computational Geometry for Curves and Surfaces*. Springer, 2006, pp. 67–116. 2

[CD88] CANNY J., DONALD B.: Simplified voronoi diagrams. *Discrete & Computational Geometry 3*, 1 (1988), 219–236. 7

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), ACM, pp. 15–22. 3, 7

[CTMT10] CAO T.-T., TANG K., MOHAMED A., TAN T.-S.: Parallel banding algorithm to compute exact distance transform with the gpu. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (2010), ACM, pp. 83–90. 2, 7

[DBCVK08] DE BERG M., CHEONG O., VAN KREVELD M.: *Computational geometry: algorithms and applications*. Springer, 2008. 2

[ER02] ETZION M., RAPPOPORT A.: Computing voronoi skeletons of a 3-d polyhedron by space subdivision. *Computational Geometry 21*, 3 (2002), 87–120. 2

[FG06] FISCHER I., GOTSMAN C.: Fast approximation of high-order voronoi diagrams and distance transforms on the gpu. *Journal of Graphics, GPU, and Game Tools 11*, 4 (2006), 39–60. 2, 7

[FGLM01] FOSKEY M., GARBER M., LIN M. C., MANOCHA D.: A voronoi-based hybrid motion planner. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on* (2001), vol. 1, IEEE, pp. 55–60. 7

[FP02] FRISKEN S. F., PERRY R. N.: Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools 7*, 3 (2002), 1–11. 3

[FPRJ00] FRISKEN S. F., PERRY R. N., ROCKWOOD A. P., JONES T. R.: Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), ACM Press/Addison-Wesley Publishing Co., pp. 249–254. 3

[Gar82] GARGANTINI I.: An effective way to represent quadtrees. *Communications of the ACM 25*, 12 (1982), 905–910. 3

[GMAM06] GARRIDO S., MORENO L., ABDERRAHIM M., MARTIN F.: Path planning for mobile robot navigation using voronoi diagram and fast marching. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on* (2006), IEEE, pp. 2376–2381. 7

[GRLM03] GOVINDARAJU N. K., REDON S., LIN M. C., MANOCHA D.: Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), Eurographics Association, pp. 25–32. 7

[HICK*00] HOFF III K., CULVER T., KEYSER J., LIN M. C., MANOCHA D.: Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on* (2000), vol. 3, IEEE, pp. 2931–2937. 7

[HIKL*99] HOFF III K. E., KEYSER J., LIN M., MANOCHA D., CULVER T.: Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), ACM Press/Addison-Wesley Publishing Co., pp. 277–286. 2, 7

[HIZLM01] HOFF III K. E., ZAFERAKIS A., LIN M., MANOCHA D.: Fast and simple 2d geometric proximity queries using graphics hardware. In *Proceedings of the 2001 symposium on Interactive 3D graphics* (2001), ACM, pp. 145–148. 7

[HT05] HSIEH H.-H., TAI W.-K.: A simple gpu-based approach for 3d voronoi diagram construction and visualization. *Simulation modelling practice and theory 13*, 8 (2005), 681–692. 2, 7

[JBS06] JONES M. W., BAERENTZEN J. A., SRAMEK M.: 3d distance fields: A survey of techniques and applications. *Visualization and Computer Graphics, IEEE Transactions on 12*, 4 (2006), 581–599. 3

[Kar04] KARAVELAS M. I.: A robust and efficient implementation for the segment voronoi diagram. In *International symposium on Voronoi diagrams in science and engineering* (2004), Citeseer, pp. 51–62. 2

[Kar12] KARRAS T.: Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics* (2012), Eurographics Association, pp. 33–37. 3

[KL14] KIM Y. J., LIU F.: Exact and adaptive signed distance fieldscomputation for rigid and deformablemodels on gpus. *IEEE Transactions on Visualization and Computer Graphics 20*, 5 (2014), 714–725. 3

[LBD*92] LAVENDER D., BOWYER A., DAVENPORT J., WALLIS A., WOODWARK J.: Voronoi diagrams of set-theoretic solid models. *Computer Graphics and Applications, IEEE 12*, 5 (1992), 69–77. 2

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *ACM Siggraph Computer Graphics 21*, 4 (1987), 163–169. 5

[Lee82] LEE D.-T.: Medial axis transformation of a planar shape. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* (1982), 363–369. 2

[LH07] LEFEBVRE S., HOPPE H.: Compressed random-access trees for spatially coherent data. In *Proceedings of the 18th Eurographics conference on Rendering Techniques* (2007), Eurographics Association, pp. 339–349. 3, 7

[LK11] LAINE S., KARRAS T.: Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on 17*, 8 (2011), 1048–1059. 3, 7

[LM03] LIN M. C., MANOCHA D.: Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*, Goodman J., O'Rourke J., (Eds.). London: Chapman and Hall/CRC, 2003. 7

[MHS*10] MISHCHENKO Y., HU T., SPACEK J., MENDENHALL J., HARRIS K., CHKLOVSKII D.: Ultrastructural analysis of hippocampal neuropil from the connectomics perspective. *Neuron 67*, 6 (2010), 1009–1020. 8

[MS07] MASEHIAN E., SEDIGHIZADEH D.: Classic and heuristic approaches in robot motion planning-a chronological review. *World Academy of Science, Engineering and Technology 29*, 1 (2007), 101–106. 7

[Nat94] NATARAJAN B. K.: On generating topologically consistent isosurfaces from uniform samples. *The Visual Computer 11*, 1 (1994), 52–62. 6

[NH91] NIELSON G. M., HAMANN B.: The asymptotic decider: resolving the ambiguity in marching cubes. In *Proceedings of the 2nd conference on Visualization'91* (1991), IEEE Computer Society Press, pp. 83–91. 6

[PLKK10] PARK T., LEE S.-H., KIM J.-H., KIM C.-H.: Cuda-based signed distance field calculation for adaptive grids. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on* (2010), IEEE, pp. 1202–1206. 3

[QZS*04] QU H., ZHANG N., SHAO R., KAUFMAN A., MUELLER K.: Feature preserving distance fields. In *Volume Visualization and Graphics, 2004 IEEE Symposium on* (2004), IEEE, pp. 39–46. 3

[RP12] RAJA P., PUGAZHENTHI S.: Optimal path planning of mobile robots: A review. *International Journal of Physical Sciences 7*, 9 (2012), 1314–1320. 7

[RT07] RONG G., TAN T.-S.: Variants of jump flooding algorithm for computing discrete voronoi diagrams. In *Voronoi Diagrams in Science and Engineering, 2007. ISVD'07. 4th International Symposium on* (2007), IEEE, pp. 176–181. 2, 7

[SB07] SOHN B.-S., BAJAJ C.: Topology preserving tetrahedral decomposition of trilinear cell. In *Computational Science–ICCS 2007*. Springer, 2007, pp. 350–357. 6

[SGG*06] SUD A., GOVINDARAJU N., GAYLE R., KABUL I., MANOCHA D.: Fast proximity computation among deformable models using discrete voronoi diagrams. *ACM Transactions on Graphics (TOG) 25*, 3 (2006), 1144–1153. 2, 7

[SGGM06] SUD A., GOVINDARAJU N., GAYLE R., MANOCHA D.: Interactive 3d distance field computation using linear factorization. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (2006), ACM, pp. 117–124. 2, 7

[Str99] STRAIN J.: Fast tree-based redistancing for level set computations. *Journal of Computational Physics 152*, 2 (1999), 664–686. 3, 5, 7

[Thr98] THRUN S.: Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence 99*, 1 (1998), 21–71. 7

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNENAT-THALMANN N., STRASSER W., ET AL.: Collision detection for deformable objects. In *Computer Graphics Forum* (2005), vol. 24, Wiley Online Library, pp. 61–81. 7

[TT97] TEICHMANN M., TELLER S.: Polygonal approximation of voronoi diagrams of a set of triangles in three dimensions. In *Tech Rep 766, Lab of Comp. Sci., MIT* (1997). 2

[VMS11] VACHHANI L., MAHINDRAKAR A. D., SRIDHARAN K.: Mobile robot navigation through a hardware-efficient implementation for control-law-based construction of generalized voronoi diagram. *Mechatronics, IEEE/ASME Transactions on 16*, 6 (2011), 1083–1095. 7

[VO98] VLEUGELS J., OVERMARS M.: Approximating voronoi diagrams of convex sites in any dimension. *International Journal of Computational Geometry & Applications 8*, 02 (1998), 201–221. 2

[WLXZ08] WU X., LIANG X., XU Q., ZHAO Q.: Gpu-based feature-preserving distance field computation. In *Cyberworlds, 2008 International Conference on* (2008), IEEE, pp. 203–208. 2, 7

[YLW11] YIN K., LIU Y., WU E.: Fast computing adaptively sampled distance field on gpu. In *Pacific Graphics Short Papers* (2011), The Eurographics Association, pp. 25–30. 3, 7

[ZGHG11] ZHOU K., GONG M., HUANG X., GUO B.: Data-parallel octrees for surface reconstruction. *Visualization and Computer Graphics, IEEE Transactions on 17*, 5 (2011), 669–681. 3