

Parallel Octree Construction on Collections of Objects

John Edwards and Nathan Vollmer and Nicholas Harrison

Idaho State University

Abstract

*We present a parallel octree algorithm that resolves between geometric objects. The octree has the property that no octree cell intersects more than one labeled object. Previous parallel algorithms either spawn kernels hierarchically, separate points only, or make no hard guarantees of object separation. Our algorithm runs in **complexity?** in the average case and has excellent results in practice. We demonstrate with results on 2D and 3D datasets.*

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

Previous paper: [\[EDPB15\]](#)

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.

Our algorithm has three steps:

1. Construct an octree on object vertices using Karras’ algorithm [\[Kar12\]](#)
2. Detect octree cells that intersect more than one object, which we call “conflict cells” (contribution)
3. Subdivide conflict cells to resolve objects (contribution)

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. ?? and ?? 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

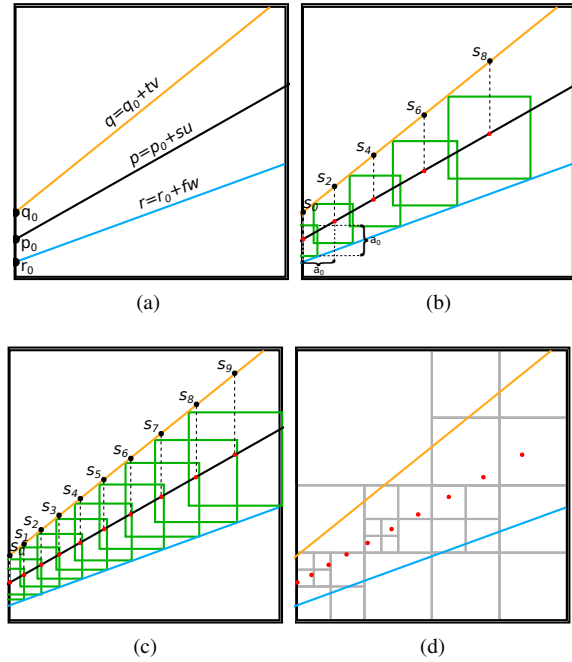


Figure 1: (a) A conflict cell with two lines from different objects. (b) Fitting boxes such that any box intersecting both lines contains at least one sample (red dots). (c) Fitting boxes such that any box intersecting both lines contains at least two samples. This ensures that an octree built from the samples using Karras' algorithm (panel (d)) will have no leaf cells that intersect both lines, ensuring that the new octree is locally free of conflict cells.

no GPU octree construction method that can resolve between objects.

3. Algorithm

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 refer to object facets, which are lines in 2D and triangles in 3D.

3.1. Resolving conflict cells

To resolve conflict cells we consider pairs of lines of different labels that intersect a cell c . Figure 1a shows two lines

$$q(t) = q = q_0 + tv \quad (1)$$

$$r(f) = r = r_0 + fw \quad (2)$$

along with their bisector

$$p(s) = p = p_0 + su \quad (3)$$

Our strategy will be to sample points P on $p(u)$ (figure 1d) such that an octree built on $V \cup P$ will completely “separate” q and r , i.e., no descendent cell of c will intersect both q and r . We do this by ensuring that P is sampled such that every box that intersects both q and r also intersects at least two points in P . Because Karras’ algorithm guarantees that every leaf cell intersects at most one point, we know that no leaf cell will intersect q and r and thus no leaf cell will be a conflict cell.

We consider only bisecting lines with slope $0 \leq m \leq 1$. All other cases can be transformed to this case. We begin by fitting the smallest box centered at ??? that intersects both q and r .

3.1.1. Case $w^y > 0$

We wish to find $a = q^x - p^x$, which results in a box that touches both lines. Given a point p , we find q by substituting in equations (1) and (2) and solving for t ,

$$t = \frac{p^x + p^y - (q_0^x + q_0^y)}{v^x + v^y} \quad (4)$$

Thus,

$$\begin{aligned} a(s) &= q^x(t) - p^x(s) \\ &= q_0^x + v^x \frac{p^x(s) + p^y(s) - (q_0^x + q_0^y)}{v^x + v^y} - p^x(s) \\ &= \frac{q_0^x v^y - q_0^y v^x + p^y(s) v^x - p^x(s) v^y}{v^x + v^y} \\ &= \frac{q_0^x v^y - q_0^y v^x + p_0^y v^x - p_0^x v^y + s(u^y v^x - u^x v^y)}{v^x + v^y} \\ &= s \frac{u^y v^x - u^x v^y}{v^x + v^y} + \frac{q_0^x v^y - q_0^y v^x + p_0^y v^x - p_0^x v^y}{v^x + v^y} \end{aligned} \quad (5)$$

As shown in figure 1, we desire to build a sequence $S = (s_0, s_1, \dots)$ such that $p^x(s_i) = p^x(s_{i-2}) + a(s_{i-2})$. That is,

$$x_i = x_{i-2} + a(s_{i-2}) \quad (6)$$

Substituting in equation (5),

$$x_i = x_{i-2} + s_{i-2} \frac{u^y v^x - u^x v^y}{v^x + v^y} + \frac{q_0^x v^y - q_0^y v^x + p_0^y v^x - p_0^x v^y}{v^x + v^y} \quad (7)$$

Substituting in equation (3),

$$x_i = p_0^x + s_{i-2} u^x + s_{i-2} \frac{u^y v^x - u^x v^y}{v^x + v^y} + \frac{q_0^x v^y - q_0^y v^x + p_0^y v^x - p_0^x v^y}{v^x + v^y} \quad (8)$$

$$= s_{i-2} \frac{v^x(u^x + u^y)}{u^x(v^x + v^y)} + \frac{q_0^x v^y - q_0^y v^x + p_0^y v^x + p_0^x v^y}{v^x + v^y} \quad (9)$$

We’re now ready to build the sequence. From (3),

$$s_i = \frac{x_i - p_0^x}{u^x} \quad (10)$$

Substituting in equation (9),

$$s_i = s_{i-2} \frac{v^x(u^x + u^y)}{u^x(v^x + v^y)} + \frac{q_0^x v^y - q_0^y v^x + p_0^y v^x - p_0^x v^y}{u^x(v^x + v^y)} \quad (11)$$

$$= \alpha s_{i-2} + \beta \quad (12)$$

$$\text{where } \alpha = \frac{v^x(u^x + u^y)}{u^x(v^x + v^y)} \text{ and } \beta = \frac{q_0^x v^y - q_0^y v^x + p_0^y v^x - p_0^x v^y}{u^x(v^x + v^y)}.$$

To solve the recurrence relation (12),

$$s_{i+2} = \alpha s_i + \beta \quad (13)$$

$$s_{i+2} - s_i = \alpha s_i - \alpha s_{i-2} \quad (14)$$

$$s_{i+2} = (\alpha + 1)s_i - \alpha s_{i-2} \quad (15)$$

The characteristic polynomial is

$$p(t) = t^2 - (\alpha + 1)t + \alpha \quad (16)$$

which has roots α and 1. Thus,

$$s_i = k_1 + k_2 \alpha^i \quad (17)$$

The initial condition when i is even is $s_0 = 0$. When i is odd, the initial condition will be where s is halfway along the first box (it could be anywhere in the interior of the first box, but we use halfway for convenience). So from (3),

$$s_1 = \frac{p^x - p_0^x}{2u^x} \quad (18)$$

$$= \frac{a(0)}{2u^x} \quad (19)$$

$$k_1^{\text{even}} = k_1^{\text{odd}} = \beta \left(1 - \frac{\alpha}{\alpha - 1} \right) \quad (20)$$

$$k_2^{\text{even}} = \frac{\beta}{\alpha - 1} \quad (21)$$

$$k_2^{\text{odd}} = \frac{a(0)}{2u^x} + \frac{\beta}{\alpha - 1} \quad (22)$$

It is tempting to formulate the recurrence relation using a single initial condition. This would be done by replacing equation (6) with

$$x_i = x_{i-1} + \frac{a(s_{i-1})}{2} \quad (23)$$

This formulation can lead to conflict cells as shown in figure 2.

3.2. Build octree on vertices

We first construct an octree on the vertices of the objects, which we call the “vertex octree”. We use Karras’ algorithm [Kar12] which sorts the Morton codes of the vertices in parallel, then constructs the binary radix tree in parallel. With the binary radix tree, the octree can be constructed with a single parallel call. The strength of this algorithm lies in the fact that overall performance scales linearly with the number of cores, regardless of the distribution of points. That is, even

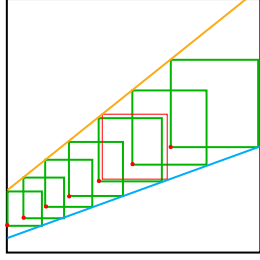


Figure 2: Using a single recurrence solution we can no longer guarantee that every box that intersects both lines contains at least two samples. The box in red contains only one sample, so the octree may contain conflict cells.

if a large number of vertices are clustered in a small area, requiring deep octree subdivision, only a constant number of parallel calls need be made. Given enough parallel units, the Karras algorithm runs in $O(\log N)$ time, where N is the number of vertices.

3.3. Identify conflict cells

Our end goal is to construct an octree such that no octree cell intersects more than one object. Note that a cell is allowed to intersect more than one facet, but all facets must belong to the same object, or, in other words, all facets must share the same label. It is possible, but unlikely, that the vertex octree has this property. If so, then we are done. Otherwise, we must identify and subdivide conflict cells.

One naive algorithm to identify conflict cells is to process each leaf cell c in parallel and store which facets intersect c . This is $O(N)$. Another approach is to process each facet in parallel and add it to every cell that it intersects. This is $O(k \log N)$ where k is maximum number of cells that any facet intersects. As we will show, our algorithm is $O(j + \log N)$ where j is the maximum number of facets that intersect any cell. In practice, $\log N > j$, making our algorithm $O(\log N)$.

We identify conflict cells as shown in algorithm 1. In lines 1-9, for each internal octree cell c , we store all facets for which c is the smallest containing cell. Since we are implementing this in a GPGPU environment, we don't have dynamic memory, so each facet must be processed twice. The first loop discovers how many facets are to be stored in each cell after which we allocate space for the facets. We use parallel prefix sums to determine the amount of space we need to allocate as well as the offsets for each internal cell. The second loop actually stores the facets.

The $\text{container}(f)$ procedure finds the smallest octree cell that fully contains the facet f . A straightforward implementation of $\text{container}(f)$ is to perform a standard octree search on the vertices of f and take the smallest octree cell that contains all of them. (Note that the cell is always an internal

node, since a post-condition of the Karras algorithm is that no leaf cell contains more than one vertex.) In our implementation however, we take advantage of our existing data structures. The octree cell that contains a vertex v is uniquely determined by the D-tuple bits of its morton code. For example, if a 2D vertex has morton code 010010, then the octree is traversed from the root to child 01 to child 00 to child 10. To determine $\text{container}(f)$, we find the longest common prefix (lcp) of the vertices. Truncating the length of lcp to a multiple of D , we find the smallest octree cell that contains all vertices of f . The complexity of $\text{container}(f)$ is $O(\log N)$ for both implementations. Thus, lines 1-9 run in $O(\log N)$ time.

Lines 10-28 of the algorithm identify and store all facets that intersect with a given leaf cell c . Again, it is done in two steps for memory allocation purposes. Each leaf cell c looks at its $O(\log N)$ ancestors and tests all facets stored in those ancestors for intersection with c . Any intersecting facets get stored in c . These lines run in $O(F)$ time, where F is the number of facets in all objects. Even though the loop is doubly-nested, each facet is stored in a unique internal node, so no more than F facets will be visited in the loops. In practice, far fewer than F facets will be checked for each leaf cell, because most datasets have facets that are completely contained in internal cells that are reasonably low in the tree.

The entire conflict cell detection algorithm runs in $O(\log N + L) = O(L)$ because $L > \log N$. However, average case is $O(\log N)$, considering that most lines are contained entirely in a cell at reasonably low depth.

In Step 4, Stack is preallocated to size $M \cdot 2^D$ where M is the maximum octree depth and D is the dimension. A conflict cell is a cell that intersects at least two different objects, or two lines of different labels.

The second procedure we use is $\text{direct_ancestors}(c)$, which finds all ancestors of octree cell c .

In Fig. 3, R (Root) is the smallest containing cell for lines A, B, and C, cell 20 contains line D, and cell 2 contains lines E and F. After Step 3 of the algorithm, line A is stored in leaf cells 202, 203, 21, and 3. Conflict cells, which are the only cells that are subdivided, are 203 and 21.

4. Compute GVD surface

5. Results and applications

Our implementation[†] of the algorithm supports polygons and triangulated objects, and our wavefront initialization

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

Algorithm 1: FIND_CONFLICT_CELLS**Input:** VertexOctree

```

// Store contained facets
1 for facet  $f$  in Objects do in parallel
2    $a := \text{container}(f)$ 
3    $a.\text{numFacets} := a.\text{numFacets} + 1$ 
4 end
5 Allocate space for facets in internal cells
6 for facet  $f$  in Objects do in parallel
7    $a := \text{container}(f)$ 
8    $a.\text{facets} := a.\text{facets} \cup f$ 
9 end
// Store intersecting facets
10 for leaf cell  $c$  in VertexOctree do in parallel
11   foreach cell  $a$  in  $\text{direct\_ancestors}(c)$  do
12     foreach facet  $f$  in  $a.\text{facets}$  do
13       if  $f$  intersects  $c$  then
14          $c.\text{numFacets} := c.\text{numFacets} + 1$ 
15       end
16     end
17   end
18 end
19 Allocate space for facets in leaf cells
20 for leaf cell  $c$  in VertexOctree do in parallel
21   foreach cell  $a$  in  $\text{direct\_ancestors}(c)$  do
22     foreach facet  $f$  in  $a.\text{facets}$  do
23       if  $f$  intersects  $c$  then
24          $c.\text{facets} := c.\text{facets} \cup f$ 
25       end
26     end
27   end
28 end

```

Algorithm 2: REFINE_OCTREE**Input:** Octree, conflict_cells

```

// 4. Octree refinement
1 for leaf cell  $c$  in Octree do in parallel
2    $c' := c$ 
3   while  $c' \in \text{conflict\_cells}$  do
4      $(c'_0, c'_1, \dots, c'_{2^d-1}) := \text{subdivide } c'$ 
5     push  $(c'_0, c'_1, \dots, c'_{2^d-1})$  onto Stack
6      $c' := \text{Stack.pop}$ 
7   end
8 end

```

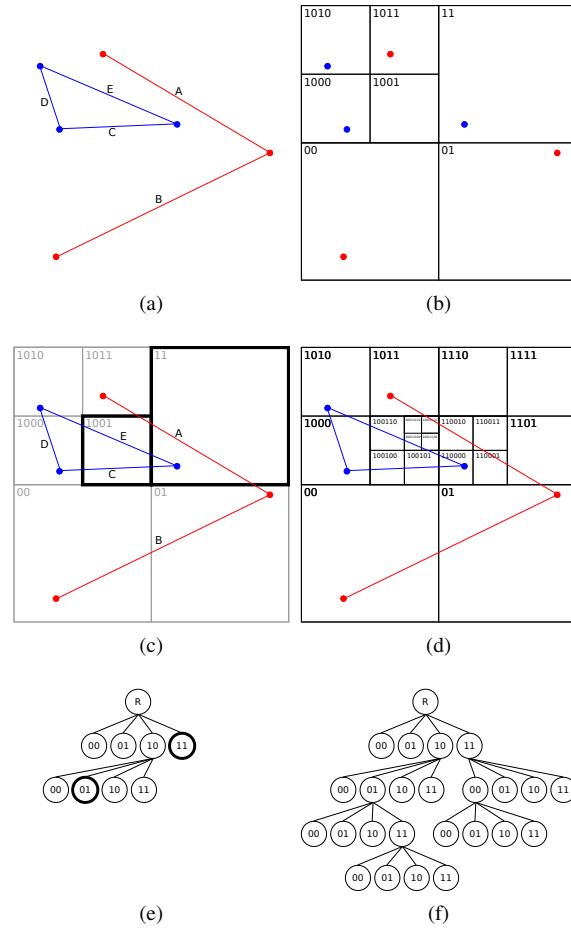


Figure 3: (a) A red object and a blue object. (b) The vertex octree, or octree built on the object vertices using Karras' algorithm. (c), (e) The vertex octree with conflict cells highlighted. Note the label of an octree cell in (c) is the concatenation of labels from root R to the leaf cell in (e). This value also corresponds to the highest order bits of the morton code of any point in the cell. (d), (f) The octree after resolution of conflict cells.

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 ?? shows our implementation of the GVD computation pipeline, and Figure ?? 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.

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. ??	3	7	8	54	3	0.9	83
Fig. ??	4	15	12	146	9	3.9	232
Fig. ??	470	5	24	158	8	2.0	151
Fig. ??	448	4015	8	2716	151	195	8100
Fig. ??	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.

5.1. Comparison to other methods

6. Conclusions

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. [2](#)
- [BCMAS08] BOADA I., COLL N., MADERN N., ANTONI SEL-LARES J.: Approximations of 2d and 3d generalized voronoi diagrams. *International Journal of Computer Mathematics* 85, 7 (2008), 1003–1022. [2](#)
- [BCS02] BOADA I., COLL N., SELLARES J.: The voronoi-quadtrees: construction and visualization. *Eurographics 2002 Short Presentation* (2002), 349–355. [2](#)
- [BGPZ12] BÉDORF J., GABUROV E., PORTEGIES ZWART S.: A sparse octree gravitational n -body code that runs entirely on the gpu processor. *Journal of Computational Physics* 231, 7 (2012), 2825–2839. [2](#)
- [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. [2](#)
- [BWY06] BOISSONNAT J.-D., WORMSER C., YVINEC M.: Curved voronoi diagrams. In *Effective Computational Geometry for Curves and Surfaces*. Springer, 2006, pp. 67–116. [1](#)
- [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. [2](#)
- [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](#)
- [DBCVK08] DE BERG M., CHEONG O., VAN KREVELD M.: *Computational geometry: algorithms and applications*. Springer, 2008. [1](#)
- [EDPB15] EDWARDS J., DANIEL E., PASCUCCI V., BAJAJ C.: Approximating the generalized voronoi diagram of closely spaced objects. *Computer Graphics Forum* 34, 2 (2015), 299–309. [1](#)
- [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](#)
- [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. [2](#)
- [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. [1](#), [2](#)
- [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](#)
- [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. [2](#)
- [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. [1](#)
- [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. [1](#), [2](#), [3](#)
- [KL14] KIM Y. J., LIU F.: Exact and adaptive signed distance fields computation for rigid and deformable models on gpus. *IEEE Transactions on Visualization and Computer Graphics* 20, 5 (2014), 714–725. [2](#)
- [LBD*92] LAVENDER D., BOWYER A., DAVENPORT J., WALLIS A., WOODWARD J.: Voronoi diagrams of set-theoretic solid models. *Computer Graphics and Applications, IEEE* 12, 5 (1992), 69–77. [1](#)
- [Lee82] LEE D.-T.: Medial axis transformation of a planar shape. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* (1982), 363–369. [1](#)

- [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. [2](#)
- [LK11] LAINE S., KARRAS T.: Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on* 17, 8 (2011), 1048–1059. [2](#)
- [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. [2](#)
- [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. [2](#)
- [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](#)
- [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](#)
- [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](#)
- [Str99] STRAIN J.: Fast tree-based redistancing for level set computations. *Journal of Computational Physics* 152, 2 (1999), 664–686. [2](#)
- [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](#)
- [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](#)
- [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. [2](#)
- [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. [2](#)