Submission to GMP 2017

Parallel Quadtree Construction on Collections of Objects

Submission number: 142

Abstract

We present a parallel quadtree algorithm that resolves between geometric objects. The quadtree has the property that no quadtree 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.

1 Introduction

`sec:intro`

Constructing quadtrees on objects in an important task with applications to collision detection, distance fields, robot navigation, object description, and other applications. Quadtrees built on objects most often model the objects themselves, providing a space-efficient representation of arbitrarily complex objects. Our work however, centers on using quadtrees to separate, or resolve, collections of closely spaced objects. Using a quadtree, we can model the space between objects, the first step in constructing distance fields, detecting collisions, and computing the generalized Voronoi diagram. Modeling inter-object spacing is computationally straightforward when inter-object spacing is large compared to the world bounding box. Approaches typically involve a uniform grid of the space, which leads to efficient computation that often uses graphics processors.

Difficulties arise when spacing between objects is small relative to the size of the domain. An approach using a uniform grid would have excessive memory requirements in order to resolve between objects. What does "resolve" mean? In these cases, teh uniformly sized grid cell must be small enough to fit between objects at every location in the domain. To our knowledge, only one algorithm [EDPB15] computes an adaptive data structure that fully resolves between objects without using unreasonable amounts of memory.

We present an algorithm that builds a quadtree on arbitrarily spaced objects in parallel. This work extends the work done by Edwards et al [EDPB15] by computing the quadtree in parallel with an algorithm targeted for the GPU. Our algorithm performs an order of magnitude faster than the previous work and will be an excellent base for later distance transform and generalized Voronoi diagram computation.

Our algorithm has three steps:

1. Construct an quadtree on object vertices using Karras' algorithm [Kar12]
2. Detect quadtree cells that intersect more than one object, which we call "conflict cells" (contribution)
3. Subdivide conflict cells to resolve objects (contribution)

Each step is done in parallel either on vertices, quadtree cells or object facets.

2 Related work

In an early work, Lavender et al. [LBD*92] define and compute quadtrees over a set of solid models. 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.

Two seminal works build octrees on objects in order to compute the Adaptive Distance Field (ADF) on octree vertices. Strain [Str99] fully resolves the quadtree everywhere on the object surface, and Frisken et al. [FPRJ00] resolve the quadtree 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. Many recent works on fast quadtree construction using the GPU are limited to point sites [BGPZ12, Kar12, ZGHG11]. Kim and Liu's work [KL14] is similar, computing the quadtree on the barycenters of triangles, giving an approximation of an object-resolving quadtree. Most quadtree approaches that support surfaces [BLD13, CNLE09, LK11, LH07] are designed for efficient rendering, and actual construction of the quadtree is implemented on the CPU.

Two works [BC08, PLKK10] implement Adaptive Distance Fields in parallel on quadtrees but building the quadtree itself is done sequentially. Yin et al. [YLW11] compute the octree entirely on the GPU using a bottom-up approach by initially subdividing into a complete quadtree, resulting in memory usage that is no better than using a uniform grid. We have found no GPU quadtree construction method that is fully adaptive and can resolve between objects.

3 Algorithm

`sec:algorithm`

We refer to quadtree leaf cells that intersect two or more objects as "conflict cells." A necessary and sufficient condition for a quadtree to resolve objects is to have no conflict cells. Our approach to computing such a quadtree is to first detexct conflict cells in parallel and then resolve them.

Our algorithm proceeds in three main steps:

1. Build initial octree on vertices V [Kar12]

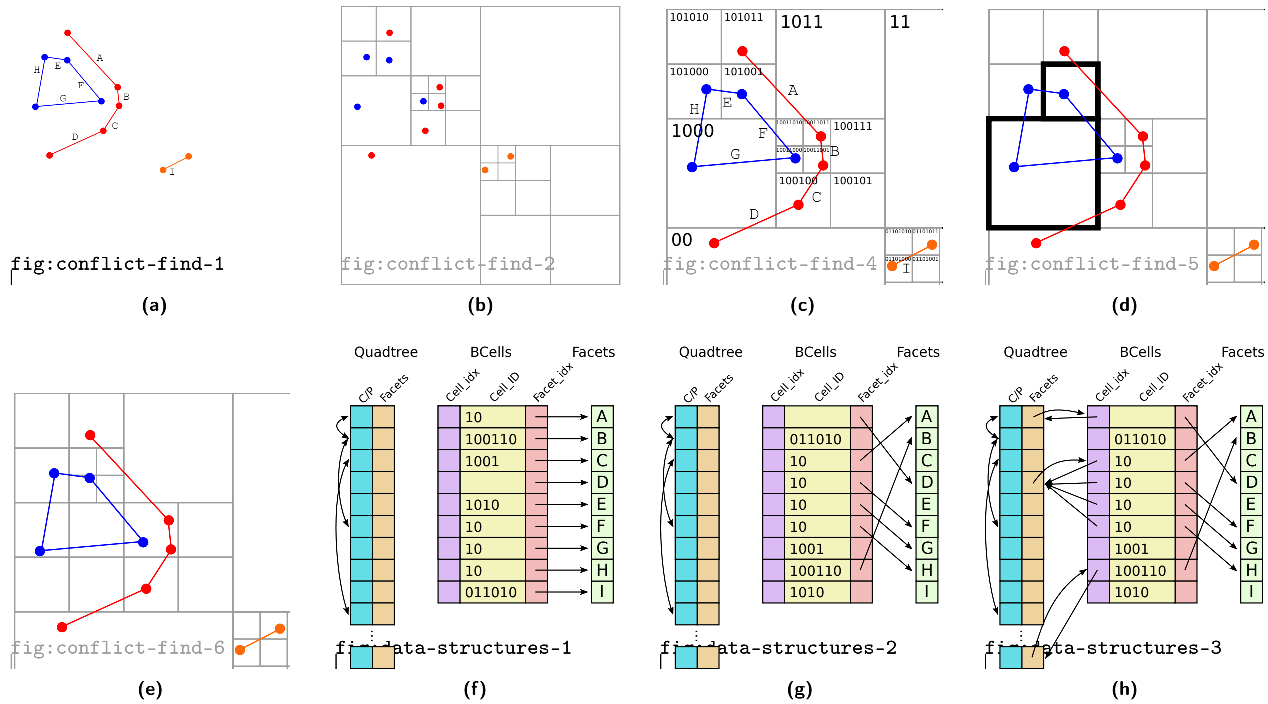

Figure 1: (a) We have three objects, blue, red, and orange with facets labeled A-I. (b) We construct an initial quadtree on the vertices using Karras' algorithm. (c) We then compute the smallest common cell (SCC) for each facet. These pairs are given in figure **??**. (d) Conflict cells, which intersect more than one object, are highlighted. (e) The new quadtree after conflict resolution. (f) The bounding cells (BCells) are stored in an array initially sorted on facet index (letters are used here for clarity). (g) We sort the BCells array using a parallel radix sort on BCell address for fast indexed access. (h) We then, in parallel on each element of BCells, store the index of the quadtree cell corresponding to the Cell_ID and, if the previous element in the BCells array does not have an identical Cell_ID, store the BCell index in the quadtree Facets field. I don't think we need `BCells.Cell_idx`. I don't use it in the algorithm description.

`fig:conflict-find`

`fig:scc-sort`

2. Detect conflict cells
   (a) Build bounding cells (BCells) array (Figure 1f)
   (b) Sort BCells array (Figure 1g)
   (c) Index BCells with octree data structure (Figure 1h)
   (d) Find facets that intersect each leaf cell
3. Resolve conflict cells
   (a) Compute sample points S
   (b) Build new octree on $V \cup S$
4. Iterate

Steps 1 and 2 are independent of dimension, and so our descriptions will use dimension-independent terms. In this paper, step 3 is described in 2D, with a 3D extension left to future work.

3.1 Build initial octree

Our first step is to build an octree on the vertices using Karras' algorithm [Kar12]. This step gives us our first approximation to our final octree.

3.2 Detect conflict cells

Let the "quadtree address" refer to the unique ID of a quadtree cell C found by concatenating the local addresses of its ancestors from Root to C. The address of the root cell is a special case and is defined as R. Figure 1c shows the address of each leaf cell in a quadtree.

We define a bounding cell (BCell) to be the smallest internal quadtree node which entirely contains a given facet. Given a facet defined by n endpoints $P = \{p_1, p_2, \ldots, p_n\}$, the quadtree address of the BCell is the longest common prefix of the Morton codes of the points in P. Figure **??** gives the addresses of the BCells of the facets in figure 1c.

We begin by constructing an array of BCells (see figure 1f), which is done in parallel over all facets. Each facet f computes the longest common prefix of its vertices and stores the result in `BCells[`f`].Cell_ID`.

Next we sort the `BCells` array on the `Cell_ID` field using a parallel radix lexicographical sort (Figure 1g). `BCells` array construction and sorting is done in parallel with the initial Karras octree construction.

Then we use the `BCells` array and octree data structure to find the conflict cells using algorithm 2. We process each leaf cell L in parallel. We set L's color to -1 (uninitialized). We then investigate each ancestor A of L. We find the ancestors using the `Parent` field in the octree data structure. Using the `Facets` field, we find the first of possibly multiple facets bounded by A. The first facet's index is found using `BCells[Quadtree[`A`].Facets].Facet_idx`. Any facet f for which A is the bounding cell could potentially intersect the leaf cell L. We test for intersection between f and L and continue on here....

Algorithm 1: FIND_CONFLICT_CELLS

```
Input: VertexQuadtree
for leaf cell L do in parallel
    L.color = -1
    foreach cell A in direct_ancestors(L) do
        id := compute_cell_index(A)
        foreach cell2facet in identical elements of cells2facets[id] do
            f := cell2facet.facet
            if f intersects L then
                if L.color == -1 then
                    // First facet found
                    // that intersects L
                    L.color = f.color
                    L.facet[0] = f
                end
                else if L.color != f.color then
                    // Cell L is ambiguous
                    L.color = -2
                    L.facet[1] = f
                end
            end
        end
    end
end
```

alg:quadtree_intersection alg:find_conflict_cells

3.3 Resolve conflict cells

A conflict cell is a quadtree cell that intersects at least two different objects. To resolve a conflict cell c, we consider pairs of lines of differing labels that intersect c. Figure 2a shows two lines

$$q(t) = q = q_0 + tv \tag{1}$$ eqn:q

$$r(f) = r = r_0 + fw \tag{2}$$ eqn:r

along with a line

$$p(s) = p = p_0 + su \tag{3}$$ eqn:p

that bisects q and r. Our strategy will be to sample points P on $p(s)$ (figure 2d) such that an quadtree 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 will find a series of boxes such that each box's left-most intersection with $p(s)$ is a sample point meeting the above criterion.

We consider only cases where the slope of p is in the range $0 \le m \le 1$. All other cases can be transformed to this case using rotation and reflection. We begin by fitting the smallest box centered on a point p that intersects both q and r. We break the problem into two cases: the *opposite* case (see Figure 2b) is where $w^y > 0$, so each box intersects q and r at its top-left and bottom-right corners, respectively. The *adjacent* case (see Figure 2e) is where $w^y < 0$, so the line intersections are adjacent at the top-left and bottom-left corners of the box.

3.3.1 Finding $a(s)$ – *opposite* case

Given a point $p(s)$, we wish to find $a = a(s)$, which will give us the starting x coordinate for the next box. Consider the top-left corner of the box $q(t(s)) = q(t)$ and the bottom-right corner $r(f(s)) = r(f)$.

Because $p^x(s) = q^x(t)$,

$$t = \frac{p^x(s) - q_0^x}{v^x} = \frac{p_x^x - q_0^x + su^x}{v^x} \tag{4}$$ eqn:t

Because our boxes are square,

$$r(f) = r_0 + fw = q_0 + tv + a \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{5}$$ eqn:ro

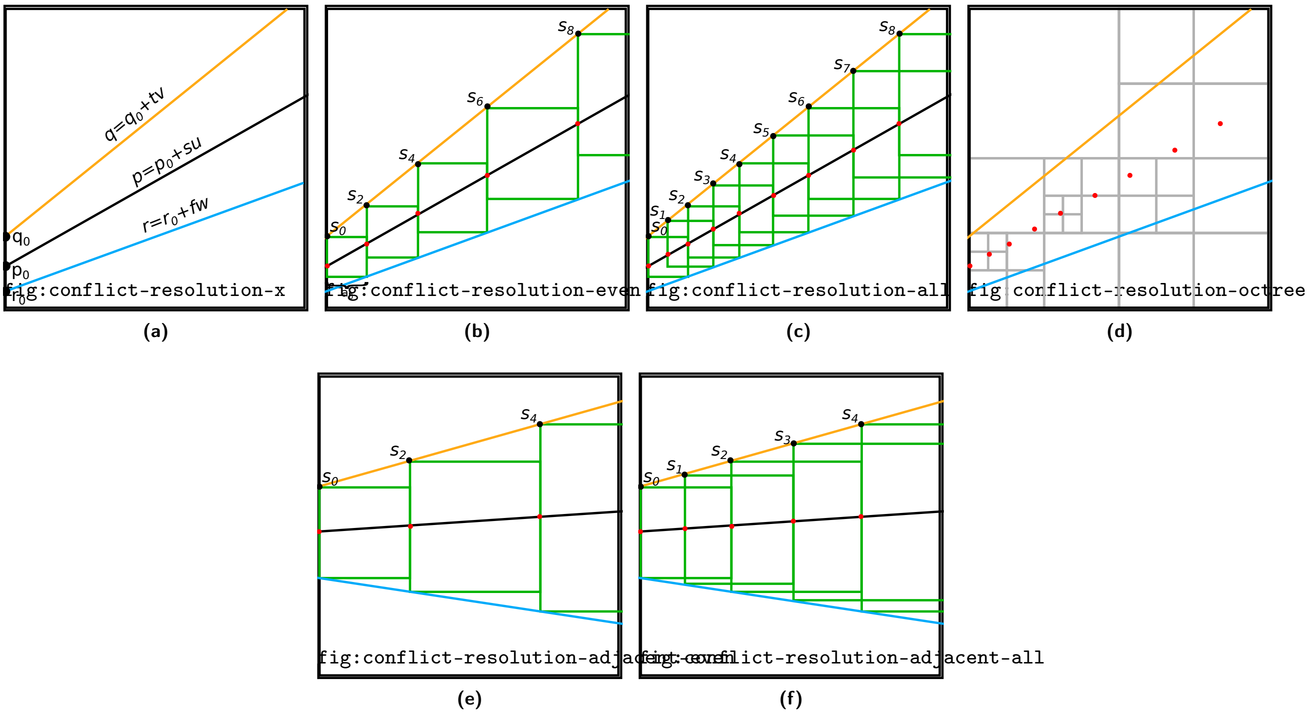

Figure 2: (a) A conflict cell with two lines from different objects. (b) Fitting boxes such that any box intersecting both lines contains at least one sample (red dots). (b) Fitting boxes such that any box intersecting both lines contains at least two samples. This ensures that an quadtree built from the samples using Karras' algorithm (panel (d)) will have no leaf cells that intersect both lines, ensuring that the new quadtree is locally free of conflict cells. `fig:conflict-resolution`

From (5),

eqn:f

$$f = \frac{1}{w^y}(q_0^y + tv^y - a - r_0^y) \tag{6}$$

eqn:a1o

$$a = r_0^x + fw^x - q_0^x - tv^x \tag{7}$$

Substituting equations (4) and (6) into equation (7) and solving for a,

eqn:ao

$$a(s) = \hat{\alpha}_o s + \hat{\beta}_o \tag{8}$$

where

$$\hat{\alpha}_o = \frac{u^x |w \times v|}{v^x(w^x + w^y)} \tag{9}$$

and

$$\hat{\beta}_o = \frac{|w \times v|(p_0^x - q_0^x) + v^x(|r_0 \times w| + |w \times q_0|)}{v^x(w^x + w^y)} \tag{10}$$

3.3.2 Finding $a(s)$ – *adjacent* case

Consider the top-left corner of the box $q(t(s)) = q(t)$ and the bottom-left corner $r(f(s)) = r(f)$. $r(f)$ is now defined as

eqn:ra

$$r(f) = r_0 + fw = q_0 + tv + a \begin{bmatrix} 0 \\ -1 \end{bmatrix} \tag{11}$$

Equations (4) and (6) remain the same while (7) becomes

eqn:a1a

$$0 = r_0^x + fw^x - q_0^x - tv^x \tag{12}$$

Substituting equations (4) and (6) into equation (12) and solving for a,

eqn:aa

$$a(s) = \hat{\alpha}_a s + \hat{\beta}_a \tag{13}$$

where

$$\hat{\alpha}_a = \frac{u^x}{v^x w^x} \tag{14}$$

and

$$\hat{\beta}_a = \frac{w^x(p_0^x - q_0^x) + |w \times q_0| + |r_0 \times w|}{w^x} \tag{15}$$

3.3.3 Sampling

In both the *opposite* and the *adjacent* cases, $a(s)$ is of the form $a(s) = \hat{\alpha}s + \hat{\beta}$. We now use $a(s)$ to construct a sequence of s values $S = \{s_0, s_1, s_2, \ldots, s_n\}$ that meet our sampling criterion. We first construct the even samples (see Figures 2b and 2e). Given a starting point $p(s_0)$,

$$p^x(s_{i+2}) = p^x(s_i) + a(s_i) \tag{16}$$

Substituting in equations (3) and (8)/(13),

$$p_0^x + s_{i+2}u^x = p_0^x + s_i + \hat{\alpha}s_i + \hat{\beta} \tag{17}$$

Solving for s_{i+2} gives the recurrence relation

eqn:recurrence

$$s_{i+2} = \alpha s_i + \beta \tag{18}$$

where

$$\alpha = 1 + \frac{\hat{\alpha}}{u^x} \tag{19}$$

and

$$\beta = \frac{\hat{\beta}}{u^x} \tag{20}$$

Constructing the odd samples is identical, except that we start at

$$s_1 = \left(1 + \frac{\hat{\alpha}}{2u^x}\right) s_0 + \frac{\hat{\beta}}{2} \tag{21}$$

which is the point in the center of the first box in the x-dimension.

We solve the recurrence relation (18) using the characteristic polynomial to yield

$$s_i = k_1 + k_2\alpha^i \tag{22}$$

where

$$k_1^{even} = \frac{\beta}{1-\alpha} \tag{23}$$

$$k_1^{odd} = \frac{\beta}{1-\alpha} \tag{24}$$

$$k_2^{even} = \frac{\alpha s_0 + \beta - s_0}{\alpha - 1} \tag{25}$$

$$k_2^{odd} = \frac{\alpha s_1 + \beta - s_1}{\alpha - 1} \tag{26}$$

The last step to formulating P for parallel computation is to determine how many samples we will need. Let $p(s_{exit})$ be the point at which the line p exits the cell.

$$k_1 + k_2\alpha^i < s_{exit} \tag{27}$$

results in

$$i < \log_\alpha \frac{s_{exit} - k_1}{k_2} \tag{28}$$

3.4 Build quadtree on vertices

We first construct an quadtree on the vertices of the objects, which we call the “vertex quadtree”. 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 quadtree 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 if a large number of vertices are clustered in a small area, requiring deep quadtree 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.

3.5 Identify conflict cells

Our end goal is to construct an quadtree such that no quadtree 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 quadtree has this property. If so, then we are done. Otherwise, we must identify 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 2. In lines 1-9, for each internal quadtree 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 *container*(f) procedure finds the smallest quadtree cell that fully contains the facet f. A straightforward implementation of *container*(f) is to perform a standard quadtree search on the vertices of f and take the smallest quadtree 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 quadtree 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 quadtree is traversed from the root to child 01 to child 00 to child 10. To determine *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 quadtree cell that contains all vertices of f. The complexity of *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 quadtree 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 *direct_ancestors*(c), which finds all ancestors of quadtree cell c.

Algorithm 2: FIND_CONFLICT_CELLS

```
   Input: VertexQuadtree
   // Store contained facets
 1 for facet f in Objects do in parallel                    alg:quadtree_containing_begin
 2     a := container(f)
 3     a.numFacets := a.numFacets + 1
 4 end
 5 Allocate space for facets in internal cells
 6 for facet f in Objects do in parallel
 7     a := container(f)
 8     a.facets := a.facets ∪ f
 9 end                                                      alg:quadtree_containing_end
   // Store intersecting facets
10 for leaf cell c in VertexQuadtree do in parallel         alg:quadtree_intersections_begin
11     foreach cell a in direct_ancestors(c) do
12         foreach facet f in a.facets do
13             if f intersects c then
14                 c.numFacets := c.numFacets + 1
15             end
16         end
17     end
18 end
19 Allocate space for facets in leaf cells
20 for leaf cell c in VertexQuadtree do in parallel
21     foreach cell a in direct_ancestors(c) do
22         foreach facet f in a.facets do
23             if f intersects c then
24                 c.facets := c.facets ∪ f
25             end
26         end
27     end
28 end                                                      alg:quadtree_intersections_end
```

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.

Algorithm 3: REFINE_QUADTREE

Input: Quadtree, conflict_cells

`// 4. Quadtree refinement`

for *leaf cell c in Quadtree* **do in parallel**
  c' := c
  while *c' ∈ conflict_cells* **do**
    $(c'_0, c'_1, \ldots, c'_{2^D-1})$:= subdivide c'
    push $(c'_0, c'_1, \ldots, c'_{2^D-1})$ onto Stack
    c' := Stack.pop
  end
end

alg:refine-quadtree

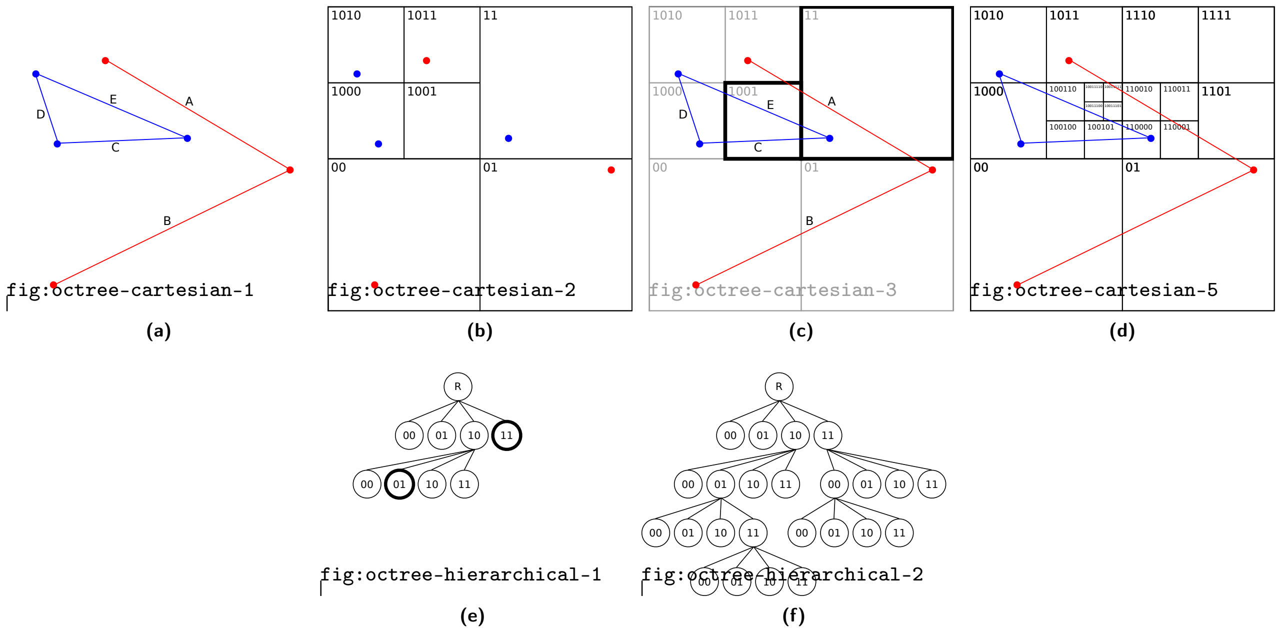

Figure 3: (a) A red object and a blue object. (b) The vertex quadtree, or quadtree built on the object vertices using Karras' algorithm. (c), (e) The vertex quadtree with conflict cells highlighted. Note the label of an quadtree cell in (c) is the concatenation of labels from root R to the leaf cell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

fig:steps

dataset	objects	object Δs ($\times 10^3$)	quadtree depth	quadtree cells ($\times 10^3$)	quadtree 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 quadtree/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; *quadtree depth* - required quadtree depth in order to resolve objects; *quadtree cells* - total number of leaf quadtree cells; *quadtree memory* - amount of memory used by the quadtree; *GVD (sec)* - seconds to perform all steps of GVD computation; *GVD* Δs - number of line segments (2D) or triangles (3D) in the GVD. tab:timings

4 Compute GVD surface

sec:bisector

5 Results and applications

Our implementation[1] 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 ?? 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.

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.

[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.

[BCS02] Boada I., Coll N., Sellares J.: The voronoi-quadtree: construction and visualization. *Eurographics 2002 Short Presentation* (2002), 349–355.

[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.

[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.

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

[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.

[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.

[DBCVK08] De Berg M., Cheong O., Van Kreveld M.: *Computational geometry: algorithms and applications*. Springer, 2008.

[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.

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

[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.

[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.

[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.

[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.

[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.

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

[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.

[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.

[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.

[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.

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

[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.

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

[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.

[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.

[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.

[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.

[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.

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

[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).

[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.

[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.

[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.

[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.