Dynamic Mesh Refinement with Quad Trees and Off-Centers [*]

Umut A. Acar
Toyota Technological Institute at Chicago
umut@tti-c.org

Benoît Hudson
Carnegie Mellon University
bhudson@cs.cmu.edu

Abstract

Many algorithms exist for producing quality meshes when the input point cloud is known *a priori*. However, modern finite element simulations and graphics applications need to change the input set during the simulation dynamically. In this paper, we show a dynamic algorithm for building and maintaining a quadtree under insertions into and deletions from an input point set in any fixed dimension. This algorithm runs in $O(\lg L/s)$ time per update, where L/s is the spread of the input. The result of the dynamic quadtree can be combined with a postprocessing step to generate and maintain a simplicial mesh under dynamic changes in the same asymptotic runtime. The mesh output by the dynamic algorithm is of good quality (it has no small dihedral angle), and is optimal in size. This gives the first time-optimal dynamic algorithm that outputs good quality meshes in any dimension. As a second result, we dynamize the quadtree postprocessing technique of Har-Peled and Üngör for generating meshes in two dimensions. When composed with the dynamic quadtree algorithm, the resulting algorithm yields quality meshes that are the smallest known in practice, while guaranteeing the same asymptotic optimality guarantees.

[*]This work was supported in part by the National Science Foundation under grants ACI 0086093, CCR-0085982 and CCR-0122581 and by a gift from Intel.

1 Introduction

In many applications, we need to *mesh* or a *triangulate* a domain consisting of points and features by splitting it into triangles such that all elements of the domain are covered by a union of triangles. Meshes are typically used to interpolate a continuous function for any of various purposes such as finite element simulations or graphics. A substantial amount of research has been performed on the *static meshing problem* [Che89, BEG90, MV92, Rup95, She98, ...] which assumes that the input domain is known a priori. We are interested in the *dynamic meshing problem* which permits the input to be changed. For the purpose of this paper, we assume that the input consists of points and that the input can be changed by inserting new points and deleting existing points.

To be broadly applicable, a dynamic meshing algorithm must satisfy the properties satisfied by state-of-the art *static* meshing algorithms and some more required by the dyanmic setting. First the algorithm must yield *conforming* meshes, *i.e.*, all points in the input must appear as a corner of a triangle in the output. Second, the output must be *good quality*, *i.e.*, the internal angles of the triangles in the output must be bounded away from 180 $^\circ$. Third, the output must be *size competitive*, *i.e.*, the number of triangles in the output must be at most a constant factor larger than is optimal on that input. Fourth, the algorithm must be *(work) efficient*, *i.e.*, it should preprocess the input quickly. Fifth, the output mesh should be *size-conforming* in the sense of Talmor [Tal97]: output points should be spaced in relation to the local feature size, and no smaller. Finally, the algorithm must be *responsive*, *i.e.*, it should respond to insertions and deletions by updating its output quickly. The various quality properties are required by applications. For example, in Finite Element Method (FEM) of scientific computing [Joh87, for example], mesh quality determines the simulation error [BA76]; the size of the mesh determines the simulation runtime; the size of the smallest element defines the length of the timestep.

The first meshing algorithms to guarantee good quality and optimal size emerged in the early 1990s, with work from Bern, Eppstein, and Gilbert [BEG90]. Their algorithm was later extended to three and higher dimensions by Mitchell and Vavasis [MV92, MV00]. Both these solutions run in time $O(n \lg n + m)$, where n is the number of input points, and m is the number of output elements in the optimal result. The technique uses a quadtree subdivision and maintains a *grading criterion* [1] between sizes of the cells. This criterion ensures that the quadtree subdivision can be mapped to produce a mesh with good quality and size guarantees by applying a postprocess. A number of postprocesses have been proposed, the most recent of which produces the smallest meshes [HPÜ05]. Quadtree subdivisons have also been used in other geometric search problems. In particular, Eppstein et al. gave a dynamic data structure, called skip quad trees, for maintaing quadtree subdivisions under dynamic changes to support point location, compressed quad trees, and other search problems such as approximate range searching [EGS05]. Skip quad trees, however, do not maintain the grade criteria between cells. Therefore they cannot directly be used to generate good quality meshes. For example, in skip quad trees, a single-point insertion will only create one new cell, whereas in meshing, a single insertion can create logarithmically many new triangles.

Dynamic meshing comes up in adaptive mesh refinement (where the mesh must be further refined according to ephemeral features such as eddies cast off an airfoil), in surgical simulations (scalpel cuts create new features), and in crack or fracture simulations. The closely related *kinetic* meshing problem is increasingly in use for fluid-solid interaction problems in the realm of computational fluid mechanics. There are two main approaches taken in the literature. The easiest is to remesh from scratch [KFCO06, BWHT07]. Unfortunately, it is of course quite wasteful to entirely throw away an almost good mesh. Worse, the two

[1]The original paper refers to this as the *balance criterion*. We prefer to speak of grading, to avoid confusion with the only tangentially related usage of the term balance with respect to binary search trees.

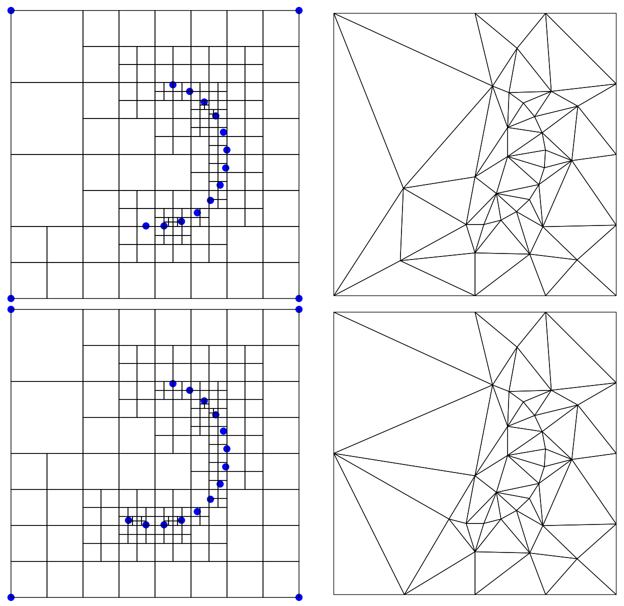

Figure 1: Illustration of dynamic stability. Top row: 13 points in part of a circle of 25 points. Bottom row: 14 points. Left: the quad-tree generated by our algorithm. Right: the mesh generated by off-center refinement. Note how few quadtree cells or mesh triangles change.

meshes may differ in every triangle, causing reinterpolation error when copying values from the old mesh to the new. Another approach is to locally remesh [CCM+04, MBF04, CGS06]. It is easy to implement locally improving the quality. However, to be size-conforming (and size-optimal), we must also *coarsen* the mesh [MTT99, LTÜ98]. Sadly, it is unclear how to make mesh coarsening a local operation; coarsening runs in the same $O(n \lg n)$ time as a full remesh. Our approach can be viewed as a hybrid: we simulate remeshing from scratch, but we only use local operations, most of the mesh does not change, and for small changes we run in sublinear time. Thus, we get better than the best of all prior worlds.

Our results. To state our results, we start with a few definitions for characterizing the input. As usual, n is the number of points in the current input, m is the the number of vertices in the smallest possible mesh, L is the size of a bounding box in which all n points fit, and s is the distance between the closest pair, so that L/s is the *spread* of the input. Finally, d is the dimensionality of space, which we consider to be a constant. Generally, d will be 2 or 3; however, our results apply in any fixed dimension.

We give a mesh refinement algorithm that runs in $O(n \lg L/s)$ time to preprocess a static point set with n points. The preprocessing step yields a quality mesh, as described above. After the preprocessing step, the input can be changed by inserting new points and deleting existing points. To each such change, the algorithm responds in $O(\lg L/s)$ time by updating the output mesh. This response time is optimal in two senses: first, the output mesh may, in the worst case, change by $O(\lg L/s)$. Second, under the assumption that the input has polynomial spread, the response time is $O(\lg n)$, matching the lower bound based on sorting. In addition to optimal response times, the algorithm guarantees that the output conforms to the input, has good quality, is size-optimal, and size-conforming. To satisfy these properties we give a dynamic algorithm that is *history independent* with respect to the preprocessing step in the sense defined by Micciancio [Mic97]: it guarantees that the output mesh is identical to a mesh that would have been obtained by performing the preprocessing step with the current input from scratch. As a result, the algorithm satisfies the same input-output relationship properties as the preprocessing step. We believe that the proposed algorithm is the first optimal-time dynamic mesh refinement algorithm with output quality and size guarantees.

Our approach to obtain optimality while guaranteeing various quality properties is to dynamize the well-graded *quad-tree* algorithm [BEG90] and the post-process that produces the mesh from the quad-tree. For the case two dimensions, we further improve the constant factors involved by dynamizing the algorithm of Har-Peled and Üngör [HPÜ05]. This post-process generates the smallest known meshes. We note that this size guarantee relies on history independence in two critical ways: 1) without history independence the output size guarantee of the processing technique would not apply to the dynamized algorithm, and 2) the Har-Peled and Üngör requires that the quad-tree be locally determined by the input, which would not hold without history independence. To obtain the optimal run-times, we make some small but crucial modifications to the original quad-tree algorithm of Bern et al. Our algorithm divides the cells into size classes and processes them in size order (largest first). This allows us to bound the response time to a dynamic changes by showing that that a small change to the input effects constant number of cells in each size class. For the algorithm, we show that the following *dynamic stability property*: consider two executions of the algorithm with two inputs that differ by a single point, the operations performed by the executions differ by $O(\lg L/s)$. We show similar results for the post-processing step.

The dynamic stability bound implies that the algorithms can be dynamized in a history-independent fashion to yield an efficient dynamic algorithm. We use recent advances on self-adjusting computation to perform this dynamization [Aca05]. The approach relies on a change-propagation algorithm that keeps track of dependences in an execution in such a way that the output can be updated by only performing those operations that differ between two execution. If the static algorithm satisfies certain properties, then change propagation takes the same time as the dynamic stability bound. We establish our results by showing that our modified quad-tree algorithm satisfies these properties. Self-adjusting computation helps abstract away from the details of the dynamization process, which can be very messy, by offering an abstract way (based on dynamic stability) to analyze the performance of dynamic algorithms, while guaranteeing history independence. The technique has been applied to various problems before including both dynamic and kinetic problems including previously unsolved problems such as three-dimensional kinetic convex hulls [ABT07, ABTV06, ABBT06, ABH$^+$04].

We note that the algorithms can be dynamized using other techniques. For example, the change-propagation algorithm employed by self-adjusting computation can be specialized to this problem to obtain the same bounds. Also, deletions can be handled lazily by delaying the removal of the deleted point until a sufficiently large (near-linear) number of points are deleted, and then remeshing from scratch. Such an algorithm can be made to be size-optimal and, in an amortized sense, has near-optimal response time. However, this approach will not properly coarsen and thus the crucial property of size-conformality will be lost. In any case, we will need a way to process insertions. Another class of dynamization techniques include those for order-decomposable search problems [Ove81]. This approach, however, only applies to divide-and-conquer algorithms.

2 Well-Graded Quadtrees

We describe an algorithm for generating well-graded quadtrees that, via dynamization, yields an responsive dynamic algorithm. Well-graded quadtrees yield a hierachical subdivision of the space into cells (i.e., hypercubes in the specified dimension) that can be used to produce a good-quality mesh of the input by applyig a post-processing step.

We define a *cell* as a hypercube in the specified dimension. We say that a cell c is *self-crowded* if it

```
QuadTreeRefine($\mathcal{P}$: point set, $L$: real, $d$: int)
1   Associate $\mathcal{P}$ with the cell $[0, L]^d$
2   If $[0, L]^d$ is crowded then { addWork($[0, L]^d$) }
3   $l \leftarrow \lg L$
4   while ($|W| > 0$) do
5       while ($|W_l| = 0$) do { decrement $l$ }
6       splitAll($W_l$)
7       increment $l$

split($c$: cell)
8   Split $c$ into $2^d$ new, smaller cells $\{c_i\}$
9   for (each point $p$ contained by $c$) do
10      associate $p$ with the $c_i$ that contains it
11  return $\{c_i\}$
```

```
addWork($c$: cell)
1   Append $c$ to $W_{\lg |c|}$

splitAll($W_l$: cell set)
2   newcells $\leftarrow \emptyset$
3   while ($W_l$ not empty) do
4       dequeue $c$ from $W_l$
5       $\{c_i\} \leftarrow$ split($c$)
6       append each $c_i$ to newcells
7   while (newcells not empty) do
8       dequeue $c_i$ from newcells
9       if ($c_i$ is crowded) then { addWork($c_i$) }
10      for (each neighbour $c'_i$ of $c_i$) do
11          if ($|c'_i| \geq 4|c_i|$) then { addWork($c'_i$) }
```

Figure 2: The quadtree refinement algorithm, modified from Bern, Eppstein, and Gilbert [BEG90].

contains two or more input point. A cell c is *crowded by a neighbour* c' if c contains exactly one point, and c' contains are least one point. We say that a cell is *crowded* if it is self-crowded or is crowded by a neighbor. We say that a cell c is *ill-graded* if it has a neighbour c' such that $|c|/|c'| \geq 4$. We say that a quadtree is *well-graded* if every unsplit cell is both well-graded and uncrowded.

Figure 2 shows our algorithm. The algorithm starts with a bounding box (square) of the the point set, with side length L. It maintains a set W of work items, i.e., cells to split, and a mapping from each cell to the set of input points that it contains. The work-set W is partitioned into $\lg L$ buckets such that the bucket W_i is a queue containing the cells of size exactly 2^i. The main loop maintains a finger l in order to quickly find the largest non-empty bucket. The algorithm proceeds in rounds. In each round, it chooses the set of the largest cells on the workset and splits all of them using the splitAll function. The splitAll first splits each cell in the bucked by calling split. The split function splits the cell into 2^d sub-cells and updates the cell-to-points mapping. The splitAll function then enqueues the newly-created crowded or ill-graded cells into the work set by calling addWork, which is only a function in order for us to easily refer to it throughout the paper. At the end of one round of split operations performed by splitAll the algorithm increments the main loop's finger, since splitAll may have made some cells ill-graded that are larger than the cells previously being processed. The key difference between this algorithm and the original quadtree algorithm [BEG90] is that it uses a size-based ordering of cells by dividing them into size classes. This is critical to our dynamic stability bounds because it allows us to show that a small change to the input affects a constant number of cells in each size class, by relying on a packing argument.

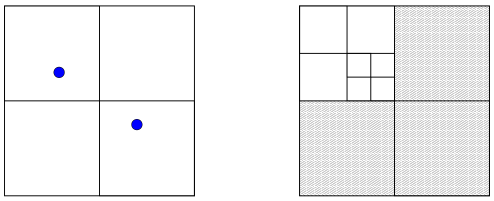

Figure 3: **Left**: two cells that crowd each other. The upper-left cell is also self-crowded. The lower-right cell, note, is not crowded. **Right**: unbalanced cells. The shaded cells are four times larger than one of their neighbours. Note neighbourhood is through vertices.

2.1 Structural Results

Lemma 2.1 *During the algorithm, unprocessed crowded cells (if any exist) are all of the size of the smallest cells in the mesh.*

Proof. Initially, this is trivially true (there is only one cell in the mesh). Later, consider the cell c^+ that was split to create a crowded cell c. Clearly, c^+ was itself crowded, and thus by induction was the smallest cell in the mesh. Now, we have destroyed c^+ and all its equally-sized cells, and replaced them with cells of half the size. These new cells must be the smallest cells in the mesh. Until we split these crowded cells, any further splits must all be grading splits. A cell can only be ill-graded if it is four times larger than its neighbour, thus grading splits cannot reduce the size of the smallest cell. ■

Lemma 2.2 *After a round of splitting crowded cells, until the next round of splitting crowded cells, l increases by exactly one every round.*

Proof. When splitting the crowded cells, we know that all cells in the mesh are well-graded: there are no smaller cells, and any larger cells, if ill-graded, would imply a work set $W_{l'}$ with $l' > l$ was non-empty, a contradiction. The crowded cells may cause ill-graded cells, with size corresponding to $l + 1$, but not of size $l + 2$ because such cells would already be ill-graded, a contradiction. ■

Lemma 2.3 *At all points in the algorithm, every cell c has at most $O(1)$ neighbours c' of size $4|c'| \leq |c| \leq 0.25|c'|$.*

Proof. The proof that the size does not differ much is immediate from the prior lemma. The proof that this implies a bounded number of neighbours is by a volume packing argument. The constant is precisely $6^d - 4^d$. ■

2.2 Size and quality guarantees

To obtain the size and quality guarantees, we can use any of the standard postprocesses published in Bern *et al.* or Mitchell and Vavasis [BEG90, MV00]. Given that our algorithm is just a specific ordering consistent with the schema given by the prior results, we inherit the size and quality guarantees. For example, we can show that all the simplices have aspect ratio at least some constant that depends only on the dimension, and not on the input point set. Furthermore, we can show that among all Steiner triangulations that respect that aspect ratio bound and in which all the input points appear, the size of the triangulation output by the quadtree algorithm and its postprocess is within a constant factor of optimal. In fact, the bound is stronger: at any point p in the domain, we know that the cell that contains p has size within a constant factor of the local feature size at p (the distance from p to the second-nearest input point): the quad-tree is size-conforming.

3 Dynamic Stability

To establish the runtime of our dynamic algorithm, we determine the *stability* of the output relative to changes in the input. The arguments will be familiar to designers of parallel algorithms – indeed, we draw on packing arguments from prior parallel meshing results [STÜ02, HMP07]. Our runtime is regulated in large part by the data dependence structure of our algorithm. We must show that dependency paths are at most $O(\lg L/s)$ long. Unlike in parallel algorithms, we must also show that the dependences cannot fan out: even constant fanout would give us a runtime of $O(\text{poly}(L/s))$, which is completely unacceptable.

Formalizing the notion of stability, consider a run of our algorithm. It reads in the points, performs some operations, reads and writes to memory, then returns an output. We can define an *execution trace* in the following way: operations and memory locations are nodes; there is an edge from a memory location a to an operation f if f reads a; and there is an edge from operation f to memory location b if f writes b. The *dynamic stability* of one point p is the symmetric difference between the sets of nodes in trace T_1 where p was not present, and the nodes in another trace T_2 where p is present. Note that this is a symmetric difference, so that the stability of adding and removing the same point are equal.

Given a trace, we can make an assignment from data locations to other data locations: we assign to a data location l all of its descendents in the trace. This is exactly the set of nodes whose value depends on the value at l. In particular, consider an input data location that holds the coordinates of a point p. We say a cell c *blames* p if the location that stores c is a trace descendent of p. Clearly, a cell c that is *blames* a point p if p crowds c. Inductively, c also blames p if c is made to be ill-graded because a neighbouring cell c' was created by a split, and c' blames p. Note that a cell may blame its splitting on many points; indeed, it will always blame at least two points.

Clearly, if we consider a given cell c, and a point p that it blames, then the distance in inductive hops from c to p is at most $O(\lg L/s)$: in every hop, we either directly blame p, or we blame p through a neighbour of half the size. Thus the trace is a shallow graph; it remains to be shown that the number of descendents of an input point (the number of cells that blame it) is bounded.

Lemma 3.1 *Assume p is blamed for the split of a cell c. Then $||pc|| \in O(|c|)$.*

Proof. If c is being split for crowding, then p is either within c or is in a neighbour c' of c, and $|c'| = |c|$. Thus $||pc|| \leq |c|$. If instead c is being split for grading, then we can follow the causal chain that leads to a cell c' that was split for crowding by p. Label the chain c_i with $c_0 = c$ and $c_k = c'$. Because of the grading condition, we know that $|c_i| = 2|c_{i+1}|$ and thus $|c| = 2^k|c'|$. The distance we can travel along the chain is maximized if the chain follows the diagonal of the cells, a total distance of $2^k\sqrt{2}|c'|$. Finally, c' either contains p or neighbours an equal-sized cell that contains p. Thus the distance from p to c is at most $(2^k\sqrt{2} + 1)|c'|$. In other words, $||pc|| < (\sqrt{2} + 1)|c|$. ■

Lemma 3.2 *Any point p is blamed for at most $O(\lg L/s)$ splits.*

Proof. Given a size class l, we know that any cell of size 2^l that is blamed on p must have distance at most $O(2^l)$. In dimension d, each cell thus has volume $(2^l)^d$, whereas all cells must fit within a volume of $O((2^l)^d)$; therefore, must thus be only $O(1)$ splits in size class l that are blamed on p. Because the algorithm does not overrefine, there are $O(\lg L/s)$ size classes. ■

To account for point location costs, we need to be a bit more careful about blame. If a split relocates a point, there are two possibilities: the split is due to crowding, or the split is due to grading. Lemma 2.1 implies that splits due to grading only occur on cells with at most one point inside, so paying for the relocation is only a constant extra cost. Splits due to crowding may be very costly, but the presence or absence of a point p only changes the decision about whether to split a crowded cell c if p is exactly the second point in the cell and its neighbours. This allows us to cut the causal chain and only have a point q blame its relocation on p when p is exactly the second point in the cell.

Lemma 3.3 *Only $O(\lg L/s)$ point location decisions blame any given input point p.*

Proof. Every point is reassigned at most $O(\lg L/s)$ times during the algorithm, since after every split the cell size falls by half. What is left is to see how many other points are reassigned because of the presence of p that would not otherwise be reassigned (*i.e.*, their containing cell was split because p was present, but would not have been split were p absent).

There are two reasons a point can be reassigned: either it is in a crowded cell being split, or it is in an ill-graded cell being split. A reassignment due to a crowded cell c can only be affected if the point p was either in the cell c or in a neighbour c' of c. Furthermore, we know that there was exactly one other point in c or c' – otherwise the algorithm would split regardless of the presence or absence of p. On the other hand, Lemma 2.1 implies that any ill-graded cell c must be uncrowded – c therefore only has one point inside. In other words, if a split reassigns any points, it reassigns exactly one point. The set of splits is $O(\lg L/s)$-stable, and thus so is the set of point reassignments. ■

Putting these observations together, we get the main theorem of the paper:

Theorem 3.4 *Our quadtree algorithm is $O(\lg L/s)$-stable under single point insertions and deletions.*

4 Dynamization with Self-Adjusting Computation

The prior section established that the static meshing algorithm is stable. This section shows how that stability can be exploited to produce a fast *dynamic* algorithm. The *self-adjusting computation* (SAC) model [Aca05] enables dynamizing static algorithms automatically by relying on a *change-propagation algorithm* to update the output when the input changes. The asymptotic complexity of change propagation can be bound by analyzing the *trace stability* of the algorithm under an insertion/deletion of a point into/from the input. In this section, we state some definitions that our analysis (Section 3) relies on. For brevity and to draw on the reader's intuition, we paraphrase from the more precise definitions in Acar's presentation [Aca05] and present the main stability or update theorem that change propagation time can be bound by stability and a priority-queue overhead for certain programs.

Definition 4.1 (Traces [Aca05, Definition 8]) *The **trace** is an ordered, rooted tree that describes the execution of a program P on an input. Every node corresponds to a function call, and is labeled with the name of the function; its arguments; the values it read from memory; and the return values of its children. A parent-child relationship represents a caller-callee relationship.*

Definition 4.2 (Cognates and Trace Distance [Aca05, Definition 12]) *Given two traces T and T' of a program P, a node $u \in T$ is a **cognate** of a node $v \in T'$ if u and v have equal labels. The **trace distance** between T and T' is equal to the symmetric difference between the node-sets of T and T', i.e., distance is $|T| + |T'| - 2|C|$ where C is the set of cognates of T and T'.*

Definition 4.3 (Monotone Programs [Aca05, Definition 15]) *Let T and T' be the trace of a program with inputs that differ by a single insertion or deletion. We say P is **monotone** if operations in T happen in the same order as their cognates in T' during a pre-order traversal of the traces.*

The change-propagation algorithm relies on a priority queue to propagate the change in the correct order. The main theorem of Acar [Aca05] states that for monotone programs, the time for change-propagation is the same as the trace distance if the priority-queue overhead can be bounded by a constant. For the theorem, we say that a program is $O(f(n))$-**stable** for some input change, if the distance between the traces T, T' of the program with inputs I and I', where I' is obtained from I by applying the change, is bounded by $O(f(n))$. For monotone programs, this stability notion corresponds to the dynamic stability notion (Section 3).

Theorem 4.4 (Update time [Aca05, Theorem 34]) *If a program P is monotone under a single insertion/deletion, and is $O(f(n))$-stable, and if the priority queue can be maintained in $O(1)$ time per operation, then change-propagation after an insertion/deletion takes $O(f(n))$ time.*

4.1 Analysis

The main theorem showed that quad-tree refinement was $O(\lg L/s)$-stable. The remainder of the analysis is devoted to showing that under single-point insertions and deletions, our version of the algorithm is monotone, and that using a standard priority queue will take $O(1)$ time per PQ operation under these updates.

Before proceeding to establish monotonicity, we must first detour to noticing that the same ill-graded cell can be added to the queue repeatedly, by several neighbours; across traces, it may be added by the same neighbour but in a different round. To sidestep these issues, we tag the the ADDWORK call with distinguishing information: the name of the cell that witnessed the bad grading, and the number of the round.

Lemma 4.5 *The well-graded quadtree algorithm is monotone.*

Proof. Let T_0 and T_1 be two traces of QUADTREEREFINE; u and v are nodes of T_0, with round-pair r and r' respectively and let $\overline{u}$ and $\overline{v}$ be their cognates in T_1 (if any). We need to prove that if $u \prec v$ then $\overline{u} \prec \overline{v}$. We distinguish between two cases.

In the first case u and v are from different rounds. Given that u and $\overline{u}$ are cognates, they share round r; similarly v and $\overline{v}$ share round r'. Thus if u precedes v, then $r < r'$ and thus $\overline{u}$ precedes $\overline{v}$.

In the second case, we have u and v from within the same round r. We show by an inductive argument that it is also monotone: The order of trace nodes within a round is defined by the order of cells on the W_l queue being processed. The order of cells in round 0 is clearly monotone: there is only one initial cell to split. Inductively, assume all cells in all prior rounds were processed monotonically between traces T_0 and T_1. Then their corresponding SPLITS were called in the same order in both traces. Therefore, the children generated by the splits were processed (in SPLITALL) in the same order in both traces. Finally, their corresponding ADDWORK calls occurred in the same order in both traces. Note that this last statement uses the fact that we only count as cognates ADDWORK calls with the same causer. ■

Theorem 4.6 (Dynamic Well-Graded Quadtree) *The* QUADTREEREFINE *algorithm, sequentialized and dynamized as described, can maintain a well-graded quad-tree over a point set in any fixed dimension under any sequence of single-point additions and removals. Let $\mathcal{P}_0$ and $\mathcal{P}_1$ be the point sets before and after an update; let $s = \min(s_0, s_1)$ and $n = \max(|\mathcal{P}_0|, |\mathcal{P}_1|)$. Then our dynamic algorithm runs in time $O(\lg L/s)$ and uses a history-independent data structure of size $O(n \lg L/s)$.*

Proof. The Lemmata of the present Section show that QUADTREEREFINE is $O(\lg L/s)$-stable, monotone. To show that change-propagation takes the same time, we need to show that only $O(1)$ trace nodes are in the priority queue at any time. We know from prior proofs that during change propagation, only $O(1)$ trace nodes are processed in any size class. Furthermore, at most 3 size classes are in the queue at any one time: the current size class; ill-graded cells in one size class larger, if any; and crowded cells which may be in a smaller size class. The priority queue can therefore be maintained in constant time. By Theorem 4.4, we conclude that the algorithm responds to dynamic changes in $O(\lg L/s)$ time.

Since change propagation ensure history independence, our algorithm is history independent. Thus, our algorithm is topologically identical to one that results from inserting the n points of $\mathcal{P}_1$ one by one. Give our time bound, we know that we can do n insertions in $O(n \lg L/s)$ time. Since self-adjusting computation never uses more stace than the running time of the from-scratch algorithm, our space bound is $O(n \lg L/s)$. ■

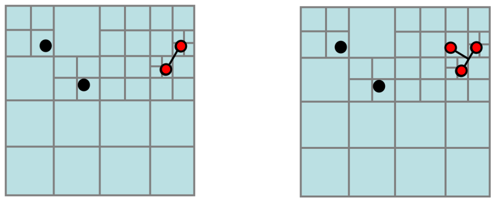
Figure 4: Illustration of the HPU algorithm picking a loose pair, and inserting the off-center since there is no third point nearby. The quadtree is carefully used for point location purposes.

5 Generating small meshes in 2d

The meshes output by the postprocess described in Section 2.2 are within a constant factor of optimal size and of the best possible quality. In practice however, they are substantially larger than than those output by Ruppert refinement [Rup95], and unlike in Ruppert refinement, they do not offer the user of the mesh any control of the desired quality bound. Üngör [Üng04] described a way of choosing what he called an *off-center*: given a bad-quality triangle (one with a small angle), we can insert a Steiner point so that the shortest edge of the triangle forms a triangle with the off-center that exactly achieves the quality threshhold. In theory, off-centers yield optimal-size meshes. In practice, off-center meshes are the smallest known. Har-Peled and Üngör [HPÜ05] then showed how to use off-centers to post-process a graded quad-tree in order to simultaneously achieve the time bounds from quadtree meshing and the small output size from off-center meshing. We show here how to dynamize the Har-Peled and Üngör postprocess. Due to space constraints, we leave the full details to the Appendix A.

The algorithm proceeds as follows: iteratively, in order from smallest to largest quadtree cell, the algorithm considers every input point p in a given cell, then searches neighbouring cells for an input point q. Having found such a pair of points, it checks whether there is a third point r such that pqr is a Delaunay triangle, and pqr has good quality. If there is no such r, then pq is a termed *loose* pair. The algorithm constructs an appropriate r using the off-center, and inserts this r, which is now treated as an input point. See Figure 5. During this routine, the quadtree serves the purpose of performing the point location (for p and q) and range queries (for r, if it exists). As a final post-process, we can again use the technique of starting from the smallest cell to the largest and using the quadtree for point location to compute the Delaunay triangulation in linear time.

We deviate in one important respect from the original algorithm of Har-Peled and Üngör: they left undefined the order of operations pairs within a size class. To establish the monotonicity condition, we require that they be done in FIFO order. This should be reminiscent of our modification of the Bern *et al.* algorithm.

Given that our algorithm performs the same steps as the original algorithm, the correctness, size optimality (and in-practice performance), and static runtime of our modified HPU algorithm immediately follow. Dynamic stability is all that is left to establish. The argument (detailed in the Appendix) is reminiscent of the dynamic stability argument for the quadtree itself: we define a notion of *blame* for off-centers upon input points, and prove a packing lemma:

Lemma 5.1 (Off-centers pack) *Let r be an off-center that blames an input point p. Then $|rp| \in \Theta(NN(r))$ where $NN(r)$ is the nearest neighbour of r when r is inserted.*

Theorem 5.2 *Given a dynamic point set $\mathcal{P} \in [1/3, 2/3]^2$ and a radius/edge ratio $\rho > 1$, we can dynamically maintain a mesh of the desired quality using within a (in practice small) constant factor of the optimal* *number of Steiner vertices. Each addition to or deletion from the input point set can be performed in* $O(\lg L/s)$ *time.*

Proof. Using self-adjusting computation, run the dynamically-stable quadtree algorithm described earlier, and use that as input to the dynamically-stable HPU postprocess described in this section. Upon a point addition or deletion, we know from Theorem 4.6 that the quadtree updates in $O(\lg L/s)$ time. Each cell is only read $O(1)$ times by the postprocess, so propagating the quadtree changes through the postprocess is fast. Finally, HPU is itself $O(\lg L/s)$-stable, by the previous packing lemma. We omit the monotonicity and priority queue arguments for brevity. ■

6 Conclusions

The main algorithmic contribution of this paper is a dynamic algorithm for maintaining a quality mesh in arbitrary dimension. The Har-Peled Üngör result that we dynamize is almost certainly general-dimensional; off-centers generalize [Üng04], and few of the static algorithm's proofs depend on dimension. Our stability results are independent of dimension (except for a constant, exponential in the dimension, hidden in the big-O; this is unavoidable in mesh refinement).

The main theoretical advance, on the other hand, is the stability result. At least as often as wanting to add features to the mesh, practitioners want to *move* the mesh: to put a velocity field upon the vertices of the mesh, advect them, and then recover a quality mesh. Stability results of the sort we proved here typically also have implications on the performance of this *kinetic* problem.

Compared to the worst case, our algorithm is optimal in memory usage: we use $O(n \lg L/s)$ space, and the mesh has exactly that size. However, the worst case is infrequent: there are broad classes of input where the mesh size m is only of size $O(n)$. The optimal output-sensitive space usage bound is thus $O(m)$. The memory usage is dominated by storing the points, for point location. However, to handle single-point updates, points only need to be stored in cells that have one point in them, as these are the only cells that, upon a point addition, may split themselves. All other cells need only remember counts. This yields the desired space bound, but at the cost of needing to track data dependencies by hand rather than using a self-adjusting computation library.

The algorithm that we give here only handles inputs points but not input-features such as segments or polygons. Even in the static case, handling input features is difficult: the first time-optimal algorithm that can handle features was discovered very recently [HMP06, HMP07]. As with the quadtree algorithm, this algorithm has a data dependency depth of $O(\lg L/s)$. We therefore hope to be able to use the techniques in this paper to dynamize that algorithm and thus handle more complicated geometries.

References

[ABBT06] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[ABH$^+$04] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[ABT07] Umut A. Acar, Guy E. Blelloch, and Kanat Tangwongsan. Kinetic 3d convex hulls via self-adjusting computation (an illustration). In *ACM Symposium on Computational Geometry (SCG)*, 2007.

[ABTV06] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic algorithms via self-adjusting computation. Technical Report CMU-CS-06-115, Department of Computer Science, Carnegie Mellon University, March 2006.

[Aca05] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[BA76] Ivo Babuška and A. K. Aziz. On the Angle Condition in the Finite Element Method. *SIAM Journal on Numerical Analysis*, 13(2):214–226, April 1976.

[BEG90] Marshall Bern, David Eppstein, and John R. Gilbert. Provably Good Mesh Generation. In *31st Annual Symposium on Foundations of Computer Science*, pages 231–241. IEEE Computer Society Press, 1990.

[BWHT07] Adam W. Bargteil, Chris Wojtan, Jessica K. Hodgins, and Greg Turk. A finite element method for animating large viscoplastic flow. *ACM Trans. Graph.*, 26(3), 2007.

[CCM$^+$04] D. Cardoze, A. Cunha, G. Miller, T. Phillips, and N. Walkington. A bezier-based approach to unstructured moving meshes. In *Symposium on Computational Geometry*, pages 71–80, 2004.

[CGS06] Narcis Coll, Marité Guerrieri, and J. Antoni Sellarès. Mesh modification under local domain changes. In *15th International Meshing Roundtable*, pages 39–56, 2006.

[Che89] L. Paul Chew. Guaranteed-quality triangular meshes. Technical Report 89–983, Department of Computer Science, Cornell University, 1989.

[EGS05] David Eppstein, Michael T. Goodrich, and Jonathan Zheng Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *21st Symposium on Computational Geometry*, pages 296–305, 2005.

[HMP06] Benoît Hudson, Gary Miller, and Todd Phillips. Sparse Voronoi Refinement. In *Proceedings of the 15th International Meshing Roundtable*, pages 339–356, Birmingham, Alabama, 2006.

[HMP07] Benoît Hudson, Gary L. Miller, and Todd Phillips. Sparse Parallel Delaunay Refinement. In *19th ACM Symposium on Parallelism in Algorithms and Architectures*, 2007.

[HPÜ05] Sariel Har-Peled and Alper Üngör. A time-optimal Delaunay refinement algorithm in two dimensions. In *21st Symposium on Computational Geometry*, pages 228–236, 2005.

[Joh87] Claes Johnson. *Numerical solutions of partial differential equations by the finite element method*. Cambridge University Press, 1987.

[KFCO06] Bryan M. Klingner, Bryan E. Feldman, Nuttapong Chentanez, and James F. O'Brien. Fluid animation with dynamic meshes. In *Proceedings of ACM SIGGRAPH 2006*, August 2006.

[LTÜ98] X.-Y. Li, S.-H. Teng, and A. Üngör. Simultaneous refinement and coarsening: adaptive meshing with moving boundaries. In *7th International Meshing Roundtable*, pages 201–210, Dearborn, Mich., 1998.

[MBF04] Neil Molino, Zhaosheng Bao, and Ron Fedkiw. A virtual node algorithm for changing mesh topology during simulation. In *SIGGRAPH*, 2004.

[Mic97] Daniele Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 456–464, 1997.

[MTT99] Gary L. Miller, Dafna Talmor, and Shang-Hua Teng. Optimal coarsening of unstructured meshes. *J. Algorithms*, 31(1):29–65, 1999.

[MV92] Scott A. Mitchell and Stephen A. Vavasis. Quality Mesh Generation in Three Dimensions. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, pages 212–221, 1992.

[MV00] Scott A. Mitchell and Stephen A. Vavasis. Quality mesh generation in higher dimensions. *SIAM Journal on Computing*, 29(4):1334–1370, 2000.

[Ove81] Mark H. Overmars. Dynamization of order decomposable set problems. *J. Algorithms*, 2(3):245–260, 1981.

[Rup95] Jim Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[She98] Jonathan Richard Shewchuk. Tetrahedral Mesh Generation by Delaunay Refinement. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 86–95, Minneapolis, Minnesota, June 1998. Association for Computing Machinery.

[STÜ02] Daniel Spielman, Shang-Hua Teng, and Alper Üngör. Parallel Delaunay refinement: Algorithms and analyses. In *Proceedings, 11th International Meshing Roundtable*, pages 205–218. Sandia National Laboratories, September 2002. http://www.arxiv.org/abs/cs.CG/0207063.

[Tal97] Dafna Talmor. *Well-Spaced Points for Numerical Methods*. PhD thesis, Carnegie Mellon University, Pittsburgh, August 1997. CMU CS Tech Report CMU-CS-97-164.

[Üng04] Alper Üngör. Off-centers: A new type of Steiner point for computing size-optimal quality-guaranteed Delaunay triangulations. In *LATIN*, pages 152–161, 2004.

```
DynHPU($\mathcal{P} \in [1/3, 2/3]^2$, $\rho$)
1 Construct a graded quadtree $\mathcal{QT}$
2 Rescale so that the size of the smallest cell is 1; let $L$ be the largest cell.
3 for ($i = 0$ to $\lg L$) do
4     enqueue all cells of size $2^i$ into $Q_i$
5 for ($i = 0$ to $\lg L$) do
6     while ($Q_i$ is non-empty)
7           collect all loose pairs $pq$ where $p$ is an active vertex in a cell on $Q_i$
8           empty $Q_i$
9           for each collected $pq$
10              if $pq$ is no longer loose then skip $pq$
11              compute the off-center $r$ of $pq$
12              add $r$ to the smallest cell $c$ such that (a) $c$ contains $r$, (b) $|c| \geq 2^i$, (c) $c_{\mathrm{low}}|c| \leq ||pr|| \leq c_{\mathrm{up}}|c|$
13              append $c$ to $Q_{\lg |c|}$
14              if $pq$ is still loose, repeat
15
```

Figure 5: A dynamically-stable version of the Har-Peled and Üngör [HPÜ05] algorithm. The key difference is that we define more carefully the ordering of items on the work queue. We also require the use of a dynamically-stable graded quadtree algorithm such as DynQT. Note that Line 14 is triggered only if *pq* is loose from both left and right.

A Generating small meshes in 2d

We use the following terms from Har-Peled and Üngör. Most of the following definitions define an orientation; we write the definitions for the counterclockwise (ccw) orientation and leave the reader to perform appropriate substitutions to define the clockwise (cw) equivalent.

Definition A.1 (Leaf) *Given a pair of points p and q, take a point c such that $|cp| = |cq| = \rho|pq|$, and $|pqc|$ forms a counterclockwise cycle. The* ***ccw-leaf*** *of pq is the disc $D(c, \rho|pq|)$.*

Definition A.2 (Loose pair) *A pair pq is* ***ccw-loose*** *if the* ***ccw-leaf*** *is empty of any points. A pair pq is* ***loose*** *if it is either* ***ccw-loose*** *or* ***cw-loose****.*

Definition A.3 (Crescent) *Given a pair pq, let c be the point on the ccw-leaf of pq that is farthest from p and q. The* ***ccw-crescent*** *of pq is the portion of the disc $D(c, |pc|)$ with the ccw-leaf removed.*

Definition A.4 (Off-center) *Let pq be a ccw-loose pair pq. If the ccw-crescent of pq is empty, then the* ***ccw-offcenter*** *of pq is the point c from the definition of the crescent. If the ccw-crescent is non-empty, take the point p' such that disc that circumscribes p, p', and q is empty. The* ***ccw-offcenter*** *is the center of that disc.*

Definition A.5 (Active point) *A point p is* ***active*** *if it may form a loose pair with another active point. See [HPÜ05, Lemmata 4.8–4.11] for proofs and technical definitions. Only $O(1)$ points are active in any cell of a graded quadtree.*

We present our modification of the Har-Peled and Üngör algorithm in Figure 5. DynHPU takes as input the point set, a radius/edge quality bound $\rho > \sqrt{2}$, and a dynamic quadtree. It produces as output a list of points. We can use a modification of DynHPU to produce the Delaunay triangulation in time linear in the output size: to decide that a pair pq is not loose requires finding a point t in the leaf of pq such that pqt is Delaunay.

The algorithm proceeds as follows: iteratively, roughly in order from smallest to largest loose pair, the algorithm identifies a loose pair and inserts its off-center (or both off-centers, if it is loose from both sides). It uses the quadtree for two purposes: to order the loose pairs (to within a constant factor), and to test for looseness. We deviate in one respect from the original algorithm of Har-Peled and Üngör: they left undefined the order of loose pairs within a size class i (Lines 7–15), whereas to establish Lemma A.8 we require that they be done in FIFO order. In essence, we simulate processing Q_i in parallel.

Given that our algorithm performs the same steps as the original algorithm, the correctness, size optimality (and in-practice performance), and static runtime of our DynHPU algorithm immediately follow. Dynamic stability is all that is left to establish. The argument will be reminiscent of the dynamic stability argument for DynQT: we show that any input point p can only be blamed on $O(1)$ off-center insertions for any value of i.

Definition A.6 (Insertion radius) *The **insertion radius** of an off-center r, denoted* $\mathrm{IR}(r)$*, is the distance from r to its nearest neighbour at the time r was inserted.*

Lemma A.7 (The insertion radius is large) *Consider a loose pair pq and their off-center r. Then the insertion radius of r follows* $2\rho|pq| > \mathrm{IR}(r) \geq \rho|pq|$.

Proof. There are two cases: (1) if there is a vertex t in the crescent, then r is the circumcenter of pqt. By definition, pqt is Delaunay: its circumdisc is empty of any other points. Therefore, $\mathrm{IR}(r) = R(pqt)$. Also, because pq is loose, pqt must have bad radius/edge ratio: $R(pqt)/|pq| > \rho$, or equivalently $\mathrm{IR}(r) > \rho|pq|$.

If instead the crescent is empty, then r is the farthest point on the flower of pq, and we know that the crescent of pq is empty of points. The crescent of pq has radius $|pr|$, which shows that $\mathrm{IR}(r) = |pr|$. From the Pythagorean theorem, we can compute $\mathrm{IR}(r) = |pr| > \rho|pq|$.

In either case, r, p, and q all lie on a circle of radius at most $\rho|pq|$, and thus can be separated by no more than twice that distance. ■

Lemma A.8 (Loose pairs grow geometrically) *After every iteration of the* DynHPU *while loop, the size of the smallest remaining loose pair in iteration i of the for loop grows by a factor at least ρ.*

Proof. Let s_{ij} be the length of the shortest loose pair at the beginning of the jth iteration of the while loop in iteration i of the for loop. Consider a loose pair seen at the end of iteration ij, but not seen at the beginning of the iteration. Such a loose pair must include at least one new off-center r; if it is a pair made of two new off-centers, let r be the newer one. That off-center issued from a loose pair of length at least s_{ij}. By Lemma A.7, the nearest neighbour of r is at distance at least ρs_{ij}; in particular, its partner in the loose pair must be at least that far. ■

Lemma A.9 (Loose pairs don't grow too fast) *All loose pairs processed in iteration i of the for loop have length in* $\Theta(2^i)$.

Proof. The upper and lower bounds were proven before [HPÜ05, Lemmata 4.3, 4.7]. ■

Definition A.10 (Blame for off-centers) *An off-center r **directly blames** a point p if r issues from a loose pair around p. Transitively, r **indirectly blames** those that p blames.*

Lemma A.11 (Off-centers pack) *Let r be an off-center that blames a point p. Then $|rp| \in \Theta(\mathrm{IR}(r))$.*

Proof. That $\mathrm{IR}(r) \leq |rp|$ is trivial: the insertion radius of r is empty of points.

If r directly blames p, then this is restating Lemma A.7.

If r directly blames a point q that transitively blames p, then by the triangle inequality, we have $|rp| \leq |rq| + |qp|$. We know that $|rq| = \mathrm{IR}(r)$ by definition. We can inductively assume that there is a constant k such that $|pq| \leq k\,\mathrm{IR}(q)$. Thus, $|rp| \leq \mathrm{IR}(r) + k\,\mathrm{IR}(q)$. It remains to bound $\mathrm{IR}(q)$ in terms of $\mathrm{IR}(r)$; this follows from Lemma A.7. Thus, $|rp| \leq (1 + k/\rho)\,\mathrm{IR}(r)$. For any $\rho \geq 1$, k is a constant with $k = \rho/(\rho - 1)$. ■

Finally, we can state the overall result:

Theorem A.12 *Under self-adjusting computation,* DynHPU *runs in $O(\lg L/s)$ time per addition to or removal from the input point set.*

Proof. By Theorem 4.6, maintaining the dynamic quad tree takes $O(\lg L/s)$ time per update.

Using Lemma A.11 in an area packing argument, at most $O(1)$ off-centers in iteration i blame any input point p. Therefore, at most $O(\lg L/s)$ off-centers of any iteration blame p. Every off-center insertion reads at most $O(1)$ input or Steiner points, and $O(1)$ cells of the quadtree.

For brevity, we elide the monotonicity argument, which is essentially identical to that in Section 4.1.

Again using the fact that every while loop iteration is $O(1)$-stable, and using the fact (derived from Lemma A.9) that we only affect $O(1)$ iterations of DynHPU at a time, the priority queue costs of DynHPU are $O(1)$ per operation. ■