A Comparison of Two Fully-Dynamic Delaunay Triangulation Methods

Michael D. Adams*

Abstract

Two fully-dynamic Delaunay triangulation methods, which differ in their point-location strategies, are proposed. One method is derived from the (bucket-based) BucketInc scheme of Su and Drysdale. The other is a new method based on a quadtree partitioning of the triangulation domain. The two methods are compared experimentally, and their relative merits discussed.

1 Introduction

For the application of image compression, there has been a growing interest in image representations based on arbitrary sampling (i.e., sampling at an arbitrary subset of points from a lattice) [2, 1]. Frequently, such representations are formed by constructing a Delaunay triangulation (DT) of a subset of the sample points and then generating an interpolant over each face of the resulting DT. Since images are usually sampled on a (truncated) lattice, a means is needed for determining a good subset of sample points to use for representation purposes. Typically, the solution to this sample-point selection problem requires the use of a fully-dynamic DT method. By "fully-dynamic", we mean that the DT method must allow the incremental insertion and deletion of points, where the distribution of the points is not known in advance, can change over time, and can be highly nonuniform.

Although many DT methods have been proposed in the literature [5], relatively few are suitable for use in fully-dynamic situations. For example, many methods require all points to be known in advance (e.g., divide-and-conquer approaches). The work described herein was motivated by a desire to find which fully-dynamic DT methods would be best suited to the above image-compression application. In particular, in this paper, we propose two fully-dynamic DT methods and then compare these methods with the goal of better understanding the relative merits of each. Since, in our application, the points to be triangulated always fall on the integer lattice, points are assumed to have integer coordinates in our work. It is worth noting, however, that our methods trivially extend to the case of fixed-point coordinates. For this reason, our proposed methods are of potential use in a wide variety of applications needing fully-dynamic DT methods.

The remainder of this paper is structured as follows. The general framework employed by our proposed DT methods is introduced in Section 2, and then in Section 3, each of our proposed methods is presented in the context of this general framework. Next, Section 4 compares the various methods in terms of their runtime performance. Finally, Section 5 concludes this paper with a summary of our work.

2 General Approach

In our work, we consider a general framework for the DT based on the incremental algorithm described by Guibas and Stolfi [4]. In our application, it is a requirement that, for any given point set, the DT method produce a unique triangulation. Therefore, to ensure the uniqueness of the DT, our framework employs the preferred-directions technique [3]. Our work specifically focuses on effective schemes for point location within the above framework. Before introducing the specific point-location strategies of interest, however, we will provide some additional details on the general framework into which these point-location schemes fit.

We assume the triangulation domain D to be square with power-of-two dimensions. In practice, this constraint can always be satisfied by padding the domain if necessary. Our DT framework provides three basic primitives: 1) `insertVertex`, which inserts a new vertex into the triangulation; 2) `deleteVertex`, which deletes a vertex (that has already been located) from the triangulation; and 3) `findVertex`, which locates a vertex already in the triangulation. Note that `deleteVertex` requires a previously located vertex as a parameter. So, depending on the circumstances, it may be necessary to use the `findVertex` and `deleteVertex` operations in succession in order to delete a vertex.

In our application, successive points to be inserted/deleted can often be nearby one another in the triangulation domain. For this reason, we maintain as additional state what is called the active vertex. Whenever point-location locates a vertex v to use as the starting point for an oriented walk, we substitute in its place the active vertex if the active vertex is closer to the query point than v. This can potentially save time when insertions/deletions exhibit locality.

In more detail, the various DT primitives behave as

*Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, V8W 3P6, Canada. mdadams@ece.uvic.ca

described below. The `insertVertex` primitive: 1) locates a candidate starting point for an oriented walk using the relevant point-location strategy; 2) performs an oriented walk [6] to find the face containing the new vertex; 3) inserts the new vertex into the point-location structure; 4) updates the DT by adding the new vertex and performing edge flips to restore the Delaunay property; and 5) sets the active vertex to the newly inserted vertex. The `deleteVertex` primitive: 1) updates the DT by removing the vertex and performing edge flips to restore the Delaunay property; 2) deletes the vertex from the point-location structure; and 3) sets the active vertex to any vertex that shared an edge with the deleted vertex. The `findVertex` primitive: 1) locates the specified vertex using the point-location structure, possibly in conjunction with an oriented walk; and 2) sets the active vertex to the located vertex.

3 Point-Location Strategies

Bucket Method. The first of our two point-location methods is a modified version of the BucketInc scheme from [5]. Our changes to the original (i.e., BucketInc) scheme were made in order to allow both vertex insertion and deletion to be done efficiently. (The original method only considered efficient vertex insertion.) Due to space constraints, we do not provide a full description of our method. Instead, we simply outline the differences between the original method as described in [5] and our modified version.

In the bucket method, the triangulation domain is partitioned, using a uniform grid, into square regions called buckets. The point-location structure consists of a 2-D bucket array, with one entry per bucket. In the case of the original method, at most one vertex is associated with each bucket-array entry. In our method, however, we track every vertex that falls in a particular bucket. This is necessary to allow both vertex insertion and deletion to be done efficiently. In particular, each bucket-array entry is a doubly-linked list that contains all vertices falling in the corresponding bucket. Each list node has a pointer to its corresponding vertex object in the DT and vice versa.

Adding/removing a vertex from the bucket array is done in a straightforward manner by inserting/removing a node from the appropriate vertex list. The average number η of vertices per bucket is required to satisfy $c \leq \eta < 4c$, where c is a fixed parameter of the method and corresponds to the smallest allowable average number of vertices per bucket. Should the preceding condition become violated (e.g., due to vertex insertion/deletion), the bucket grid spacing is halved or doubled (in both dimensions) as appropriate, changing η by a factor of 4, and then the bucket array is reinitialized. When the grid spacing is decreased (during vertex insertion), we allocate a new larger bucket array, and then move each vertex from its vertex list in the old bucket array to its correct list in the new bucket array. When the grid spacing is increased (during vertex deletion), we allocate a new smaller bucket array, and merge groups of old buckets (four at a time) into new larger buckets by splicing the vertex lists of the old buckets into the new vertex lists. In both cases, rebucketing takes linear time in the number of vertices.

Point location works in the same way as with the original method, that is, by using an outward spiral search for a nonempty bucket starting from the bucket containing the search point. In the case of our method, when a nonempty bucket is found, we simply use the first vertex in the bucket's vertex list for the search result. In the original method, there is no choice involved, as at most one vertex is recorded for each bucket. Lastly, since a bucket may contain a large number of points (for nonuniform point distributions), the `findVertex` primitive employs an oriented walk starting from the first vertex in a bucket's vertex list (instead of doing a linear search over the vertex list).

Tree Method. The second of our point-location strategies is based on a quadtree partitioning of the triangulation domain D. To simplify our subsequent explanation, we assume that the bottom-left corner of D corresponds to the origin. Thus, D is of the form $\{0, \ldots, 2^S - 1\}^2$, where S is a positive integer. With this method, D is hierarchically partitioned, using a quadtree, into square regions called cells. The root cell of the quadtree is chosen as D, and the remainder of the cells are determined by the recursive splitting of the root cell. In particular, a cell is split at its midpoint along each dimension to produce four child cells. In this method, the point-location structure is a tree. Each node in the tree is implicitly (by its position in the tree) associated with a particular cell in the quadtree partitioning of D. In particular, a given node in the tree is associated with the cell in the quadtree partitioning having the same relative position with respect to its root. For example, the root node is associated with the root cell, each child of root node is associated with the corresponding child of the root cell, and so on. Each node in the tree contains the following information: 1) a pointer to the node's parent; 2) pointers to each of the node's four children; and 3) a pointer to a DT vertex that is contained in the node's cell.

Each vertex in the DT is represented by a leaf node in the tree. For a leaf node, the node's vertex pointer is set to the corresponding DT vertex. Also, each DT vertex has a pointer to its corresponding leaf node in the tree. For nonleaf nodes, the vertex pointer is set to any vertex contained in the node's cell, and may be null to indicate an uninitialized state. The tree is maintained such that leaf nodes are always placed as close to the root as possible, subject to the constraint that each leaf node must be associated with a vertex in the node's cell.

Since point coordinates are assumed to be integer, the tree can have at most $S + 1$ levels.

Now, we briefly discuss how the tree method relates to the three DT primitives. The point-location related part of the `insertVertex` primitive needs to insert a new node `n` corresponding to the new vertex `v` into the tree and also determine an already existing vertex for the starting point of an oriented walk. This process is accomplished as follows. If the tree is empty, we simply add the new (leaf) node `n` at the root, and we are done. (Since the DT structure always contains at least three vertices once created, the empty-tree case can only occur during initialization.) Otherwise, we proceed to locate the node `q` furthest from the root whose cell contains `v`. If `q` is not a leaf, we add `n` as a child of `q`; and if the vertex pointer of `q` is null, it is set to a nonnull vertex pointer of any descendant of `q` (excluding `n`). If `q` is a leaf (in which case the cell of `q` contains both `v` and the vertex of `q`), we first move `q` downwards in the tree by adding a linear chain of (new) nodes immediately above `q` until the newly resulting cell associated with `q` no longer contains `v`. Then, we add `n` as a sibling of `q`. The vertex of `q` is used for an oriented walk.

The point-location related part of the `deleteVertex` primitive needs to remove from the tree the (leaf) node `n` corresponding to the vertex `v` to be deleted. This node removal process works as follows. Let `p` denote the parent of `n`. (Since the DT always contains at least three vertices, `n` cannot be the root.) To begin, we delete `n`. If any nodes on the path from `p` to the root have `v` as their vertex pointer, the vertex pointer is set to null. If `p` has only one child `c` remaining and `c` is a leaf, the deletion of `n` has resulted in a linear chain of nodes above `c`. In this case, we move `c` upwards in the tree by deleting this linear chain of nodes. This chain deletion process is essentially the inverse of the operation occurring in the `insertVertex` primitive where a linear chain of nodes is added.

The point-location related part of the `findVertex` primitive locates the node furthest from the root whose cell contains the search vertex. The vertex pointer of this node is the vertex being sought.

Generally speaking, the computational cost of the above algorithms depends on the average tree depth ℓ. Although the distribution of points does effect ℓ, the maximum value attainable by ℓ is bounded by $S + 1$ which, in practice, tends to be relatively small. For example, in our application, we typically handle triangulation domains of size 4096×4096 or less (i.e., $S \leq 12$) in which case $\ell \leq 13$.

4 Experimental Results

We now compare the runtime performance of the bucket and tree DT methods proposed above. Since the performance of the bucket method changes with the parameter c, we have elected to provide results for two values of c, namely 2 and 0.25. For convenience, we refer to these two cases as bucket(2) and bucket(0.25), respectively. Although numerous datasets were employed in our work, we focus our attention on two herein: `planets` and `uniform`. These datasets were chosen in order to cover the case of both uniform and nonuniform point distributions. The `planets` dataset consists of 140025 points distributed in a nonuniform manner over a domain of size 1500×1867, as shown in Figure 1, with the point set having been generated by one of the sample-point selection schemes from our image-compression application. The `uniform` dataset consists of 104861 points uniformly distributed over a domain of size 2048×2048. To ensure a fair comparison, an identical software framework was used to evaluate the bucket and tree methods—only the point-location code was changed.

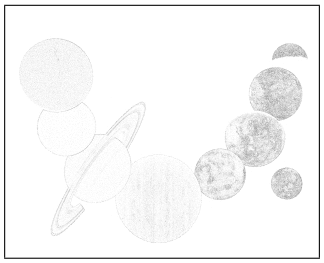

Figure 1: The `planets` dataset (rotated by 90 degrees).

To evaluate the performance of the methods under consideration, we employed a simple application that performs the following steps (in order) for a given dataset: 1) all of the points are inserted into the triangulation via `insertVertex`; 2) all of the vertices are located using `findVertex`; 3) all of the vertices are deleted via `deleteVertex` using the information obtained in the previous step. Since, in some situations, an application may not need to explicitly find a vertex (via point location) before deletion, we have chosen to keep the vertex location and deletion operations separate in steps 2 and 3, respectively. The average time spent for each of the `insertVertex`, `findVertex`, and `deleteVertex` operations was measured (based on statistics from multiple runs). In addition, the following quantities were also recorded: the amount of memory consumed by the DT and point-location structures, and the average number of orientation tests per `insertVertex` operation. The results obtained for the two datasets under consideration are shown in Table 1.

By examining the results of the table, we can make a number of observations. For `insertVertex`, the tree method is anywhere from about 3% slower to 9% faster than the bucket method, depending on the dataset and choice of c for the bucket method. For the nonuniform dataset, which is of most interest in our image-compression application, the tree method is compara-

Table 1: Comparison of triangulation methods for `planets` and `uniform` datasets.

	`planets`			`uniform`		
Quantity	Tree	Bucket(2)	Bucket(0.25)	Tree	Bucket(2)	Bucket(0.25)
avg. `insertVertex` time (us)	8.1534	8.8709	8.0983	8.1359	7.7663	7.9021
avg. `deleteVertex` time (us)	7.9919	9.3084	9.1362	7.7102	8.8306	8.9916
avg. `findVertex` time (us)	0.7221	2.3008	1.5119	0.7246	1.6782	1.1145
DT structure size* (MB)	46.08	43.06	44.06	34.84	32.10	33.98
point-location structure size (MB)	4.95	1.93	2.93	4.06	1.32	3.20
avg. orientation tests/`insertVertex`	5.356	10.33	5.972	5.498	6.621	5.252

*including point-location structure

ble to the bucket(0.25) scheme (within 1%) and significantly faster than the bucket(2) scheme (by about 9%). Overall, for `deleteVertex`, the tree method is consistently faster (by about 14% to 16%) than the bucket method regardless of c. This is due to the cost imposed by rebucketing in the bucket method. For `findVertex`, the tree method is also faster (by about 50% to 200%) than the bucket method. In general, we can see that the performance of the bucket method depends fairly heavily on the choice of c. For this reason, when using the bucket method, it is important that c be well chosen. In [5], it was observed that choosing the c parameter as 2 and 0.25 tended to lead to reasonably good vertex insertion times for uniform and nonuniform point sets, respectively. In our results, we see a similar trend. Another trend that we can observe is that the tree method tends to become faster relative to the bucket method as the point distribution becomes increasingly nonuniform. The tree method is better able to adapt to nonuniform distributions since it uses a partitioning of the triangulation domain with variable-size cells (whereas the buckets in the bucket method all have the same size).

As shown by the results in Table 1, in terms of memory usage, the tree method consistently exacts a higher cost than the bucket method. Also, as one would expect, memory usage for the bucket method increases as the parameter c is decreased. In many applications, it is likely that the increased memory requirements of the tree method are not of critical importance. This is due to the fact that the point-location structure only accounts for a small fraction of the total memory consumed by the DT structure. Furthermore, the memory usage numbers do not capture the fact that the bucket method transiently uses significantly more memory (i.e., when the old and new bucket arrays both exist during rebucketing).

Although no single method performs best in terms of all of the criteria considered, the tree method does present an interesting alternative to the bucket method. The tree method's execution speed is competitive with the bucket method's best performance without the need to choose any special parameters (like c). Not having to chose such parameters can be extremely beneficial if the point distribution is very unpredictable and possibly changing over time, in which case an appropriate choice of c may be difficult to make. Lastly, if vertex deletion (via `deleteVertex`) is performed very frequently, the tree method also has an advantage since it is faster.

5 Conclusions

In this paper, we have proposed two fully-dynamic DT methods, namely the bucket and tree methods. Although neither method is superior to the other in terms of all of the criteria considered, the tree method has some advantages that may make its use preferable in some applications. Perhaps, the biggest advantage of the tree method is that it performs well for a wide variety of point distributions without the need for any special input parameters (like c). This could be quite advantageous in situations where the point distribution is highly unpredictable or changes over time. In any case, by using the results of our work and exercising good judgment, one can hope to build more efficient applications utilizing fully-dynamic DTs.

References

[1] M. D. Adams. An evaluation of several mesh-generation methods using a simple mesh-based image coder. In *Proc. of IEEE International Conference on Image Processing*, pages 1041–1044, Oct. 2008.

[2] L. Demaret and A. Iske. Adaptive image approximation by linear splines over locally optimal Delaunay triangulations. *IEEE Signal Processing Letters*, 13(5):281–284, May 2006.

[3] C. Dyken and M. S. Floater. Preferred directions for resolving the non-uniqueness of Delaunay triangulations. *Computational Geometry—Theory and Applications*, 34:96–101, 2006.

[4] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, Apr. 1985.

[5] P. Su and R. L. S. Drysdale. A comparison of sequential Delaunay triangulation algorithms. *Computational Geometry—Theory and Applications*, 7(5–6):361–385, Apr. 1997.

[6] F. Weller. On the total correctness of Lawson's oriented walk algorithm. In *Proc. of Canadian Conference on Computational Geometry*, 1998.