A Comparison of Two Fully-Dynamic Delaunay Triangulation Methods

Michael D. Adams

Department of Electrical and Computer Engineering
University of Victoria
Victoria, BC, V8W 3P6, Canada
Web: http://www.ece.uvic.ca/~mdadams
E-mail: mdadams@ece.uvic.ca

August 2009
Outline

1. Background
2. Proposed Methods
3. Experimental Results
4. Conclusions
Motivation: Mesh-Based Image Representations

- for image compression, growing interest in image representations based on arbitrary sampling (i.e., sampling at arbitrary subset of points from lattice)
- select small subset of sample points; construct Delaunay triangulation (DT) of subset of sample points and form interpolant over each face of resulting DT

(a) The original image and its (b) corresponding surface; (c) a mesh approximation of the image surface, (d) its corresponding image-domain triangulation, and (e) the image reconstructed from the mesh
since images usually sampled on (truncated) lattice, means is needed for choosing good subset of sample points to use for representation purposes

solution to sample-point selection problem typically requires use of fully-dynamic DT method

fully dynamic: incremental insertion and deletion of points, where distribution of points not known in advance, can change over time, and can be highly nonuniform

although many DT methods proposed to date, relatively few suitable for use in fully-dynamic situations (e.g., some methods require all points known in advance, such as divide-and-conquer approaches)
points to be triangulated assumed to fall on integer lattice (i.e., have integer coordinates), although proposed methods trivially extend to any lattice

triangulation domain \( D \) square with power-of-two dimensions

based on incremental algorithm described by Guibas and Stolfi:


To ensure unique DT produced, preferred-directions technique of Dyken and Floater employed:


proposed methods differ only in point-location strategy
three basic primitives: insertVertex, findVertex, deleteVertex

insertVertex inserts new vertex into triangulation

- locates candidate starting point for oriented walk using point-location strategy
- performs oriented walk to find face containing new vertex
- inserts new vertex into point-location structure
- updates DT by performing edge flips to restore Delaunay property
- sets active vertex to newly inserted vertex

findVertex locates vertex already in triangulation

- located specified vertex using point-location structure, possibly in conjunction with oriented walk
- sets active vertex to located vertex

deleteVertex deletes vertex (that has already been located) from triangulation

- updated DT by removing vertex and performing edge flips to restore Delaunay property
- deletes vertex from point-location structure
- sets active vertex to any vertex that shared edge with deleted vertex

depending on circumstances, may be necessary to use findVertex and deleteVertex in order to delete vertex
Bucket Method

- based on BucketInc method from Su and Drysdale:

- triangulation domain partitioned using uniform square grid into square regions called buckets

- point location structure consists of 2-D bucket array, with one entry per bucket

- each entry in bucket array is doubly-linked list of vertices falling in bucket

- adding/removing vertex from bucket array done in straightforward manner by inserting/removing node from appropriate list

- each list node has pointer to corresponding vertex object in DT and vice versa
average number $\eta$ of vertices per bucket required to satisfy $c \leq \eta < 4c$, where $c$ is a fixed parameter of the method.

If the preceding condition is violated (due to vertex insertion/deletion), the bucket grid spacing is halved or doubled (as appropriate) in each dimension, changing $\eta$ by a factor of 4.

When grid spacing is decreased (during vertex insertion):
1. Allocate a new larger bucket array.
2. Move each vertex from its vertex list in the old bucket array to the correct list in the new bucket array.

When grid spacing is increased (during vertex deletion):
1. Allocate a new smaller bucket array.
2. Merge groups of old buckets (in groups of four) into new larger buckets by splicing vertex lists of old buckets into new vertex lists.

Since a bucket may contain a large number of points, `findVertex` employs an oriented walk starting from the first vertex in the bucket's vertex list.

Point location:
- Outward spiral search for a nonempty bucket starting from the bucket containing the point.
- When a nonempty bucket is found, the first vertex in the vertex list is used as the search result.
Tree Method

- assume triangulation domain $D$ of form $\{0, 1, 2^S - 1\}^2$, $S \in \mathbb{N}$
- triangulation domain $D$ hierarchically partitioned, using quadtree, into square regions called cells
- root cell of quadtree chosen as $D$
- remainder of cells in quadtree determined by recursively splitting root cell
- cell splitting: cell split at midpoint in each of $x$ and $y$ directions to produce four child cells

point-location data structure is tree associated with quadtree partitioning
each node in tree associated with cell in quadtree partitioning having same relative position with respect to root

each node in tree contains pointer to DT vertex contained in node’s cell (as well as pointer to node’s parent and pointers to node’s four children)

for leaf node, cell always contains exactly one vertex

for nonleaf node, cell always contains more than one vertex

one-to-one correspondence between leaf nodes and DT vertices

tree can have at most $S + 1$ levels
Point-Location Part of \texttt{insertVertex} (Complex Case)

insert vertex $v = (6, 6)$

\begin{itemize}
  \item \textbf{1} initial state
  \item \textbf{2} find node $q$ furthest from root whose cell contains $v$
  \item \textbf{3} move $q$ downwards in tree until $v$ not in cell of $q$
  \item \textbf{4} add node $n$ corresponding to $v$ as sibling of $q$
\end{itemize}
Point-Location Part of `deleteVertex` (Complex Case)

1. Initial state
2. Record parent `p` of node `n` corresponding to `v`
3. Delete `n`; record only child `c` of `p`
4. Move `c` upwards in tree
5. Ensure no nodes on path from `c` to root reference `n`
Experimental Results

- compare bucket method for $c = 2$ and $c = 0.25$ and tree method
- identical software framework used to compare methods, only point-location code changed
- simple benchmark application:
  1. all points inserted into triangulation via `insertVertex`
  2. all vertices located using `findVertex`
  3. all of the vertices deleted using `deleteVertex`
- provide results for two datasets:
  1. planets: 140025 points, nonuniformly distributed, domain size $1500 \times 1867$
  2. uniform: 104861 points, uniformly distributed, domain size $2048 \times 2048$

planets dataset (rotated)
Results for *planets* and *uniform* datasets

Comparison of triangulation methods for *planets* dataset.

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Tree</th>
<th>Bucket(2)</th>
<th>Bucket(0.25)</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. insertVertex time (us)</td>
<td>8.1534</td>
<td>8.8709</td>
<td>8.0983</td>
</tr>
<tr>
<td>avg. deleteVertex time (us)</td>
<td>7.9919</td>
<td>9.3084</td>
<td>9.1362</td>
</tr>
<tr>
<td>avg. findVertex time (us)</td>
<td>0.7221</td>
<td>2.3008</td>
<td>1.5119</td>
</tr>
<tr>
<td>DT structure size* (MB)</td>
<td>46.08</td>
<td>43.06</td>
<td>44.06</td>
</tr>
<tr>
<td>point-location structure size (MB)</td>
<td>4.95</td>
<td>1.93</td>
<td>2.93</td>
</tr>
<tr>
<td>avg. orientation tests/insertVertex</td>
<td>5.356</td>
<td>10.33</td>
<td>5.972</td>
</tr>
</tbody>
</table>

*including point-location structure

Comparison of triangulation methods for *uniform* dataset.

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Tree</th>
<th>Bucket(2)</th>
<th>Bucket(0.25)</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. insertVertex time (us)</td>
<td>8.1359</td>
<td>7.7663</td>
<td>7.9021</td>
</tr>
<tr>
<td>avg. deleteVertex time (us)</td>
<td>7.7102</td>
<td>8.8306</td>
<td>8.9916</td>
</tr>
<tr>
<td>avg. findVertex time (us)</td>
<td>0.7246</td>
<td>1.6782</td>
<td>1.1145</td>
</tr>
<tr>
<td>DT structure size* (MB)</td>
<td>34.84</td>
<td>32.10</td>
<td>33.98</td>
</tr>
<tr>
<td>point-location structure size (MB)</td>
<td>4.06</td>
<td>1.32</td>
<td>3.20</td>
</tr>
<tr>
<td>avg. orientation tests/insertVertex</td>
<td>5.498</td>
<td>6.621</td>
<td>5.252</td>
</tr>
</tbody>
</table>

*including point-location structure
Summary of Results

- considering both uniform and nonuniform cases, for `insertVertex`, tree method from 3% slower to 9% faster than bucket method
- for nonuniform case, tree method comparable to bucket(0.25) (within 1%) and significantly faster than bucket(2) scheme (by about 9%)
- for `deleteVertex`, tree method consistently faster (by about 14% to 16%)
- for `findVertex`, tree method faster (by about 50% to 200%)
- performance of bucket method depends fairly heavily on choice of c parameter
- for even more highly nonuniform point distributions (like some in Su and Drysdale paper), tree method performs even better relative to bucket method
Conclusions

- proposed two fully-dynamic DT methods (bucket and tree methods)
- neither method superior to other in all cases
- tree method has some advantages that make its use attractive in some applications
- unlike bucket method, tree method performs well for wide variety of point distributions without need for any special input parameters
- use of tree method advantageous in situations where point distribution highly unpredictable
- as future work, would be worthwhile to compare tree method to other schemes such as Delaunay hierarchy used in CGAL:
  
QUESTIONS?
Supplemental Slides
Point-Location Part of `insertVertex` (Simple Case)

- **Insert vertex** $v = (1, 3)$

1. **Initial state**

2. **Find node $q$ furthest from root whose cell contains $v$**

3. **Add new node $n$ for vertex $v$ as child of $q$**
Point-Location Part of \texttt{deleteVertex} (Simple Case)

1. initial state

2. record parent $p$ of node $n$ corresponding to $v$

3. delete $n$

4. ensure nodes along path from $p$ to root do not reference $n$