A Comparison of Two Fully-Dynamic Delaunay Triangulation Methods

Michael D. Adams

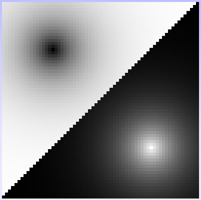
Department of Electrical and Computer Engineering
University of Victoria
Victoria, BC, V8W 3P6, Canada
Web: `http://www.ece.uvic.ca/~mdadams`
E-mail: `mdadams@ece.uvic.ca`

August 2009

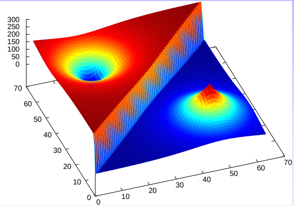
Outline

Motivation: Mesh-Based Image Representations

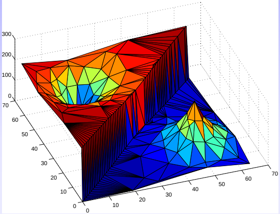
- for image compression, growing interest in image representations based on arbitrary sampling (i.e., sampling at arbitrary subset of points from lattice)
- select small subset of sample points; construct Delaunay triangulation (DT) of subset of sample points and form interpolant over each face of resulting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



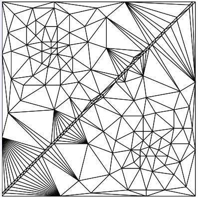
(a)



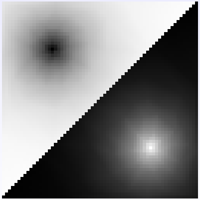
(b)



(c)



(d)



(e)

(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

Motivation (Continued)

- 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)

General Approach to DT

- 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:

  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

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

  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

- proposed methods differ only in point-location strategy

DT Primitives

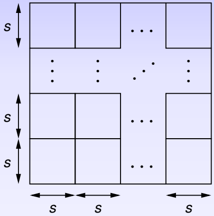
- three basic primitives: `insertVertex`, `findVertex`, `deleteVertex`
- `insertVertex` inserts new vertex into triangulation
  1. locates candidate starting point for oriented walk using point-location strategy
  2. performs oriented walk to find face containing new vertex
  3. inserts new vertex into point-location structure
  4. updates DT by performing edge flips to restore Delaunay property
  5. sets active vertex to newly inserted vertex
- `findVertex` locates vertex already in triangulation
  1. located specified vertex using point-location structure, possibly in conjunction with oriented walk
  2. sets active vertex to located vertex
- `deleteVertex` deletes vertex (that has already been located) from triangulation
  1. updated DT by removing vertex and performing edge flips to restore Delaunay property
  2. deletes vertex from point-location structure
  3. 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:

  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

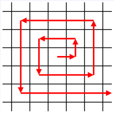
- triangulation domain partitioned using uniform square grid into square regions called buckets



- point location structure consists of 2-D bucket array, with on 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

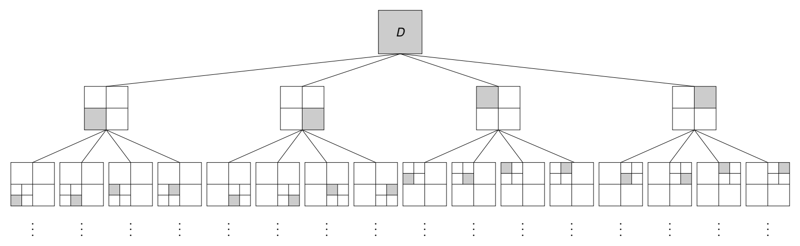
Bucket Method (Continued)

- average number η of vertices per bucket required to satisfy $c \leq \eta < 4c$, where c is fixed parameter of method
- if preceding condition violated (due to vertex insertion/deletion), bucket grid spacing halved or doubled (as appropriate) in each dimension, changing η by factor of 4
- when grid spacing decreased (during vertex insertion):
  1. allocate new larger bucket array
  2. move each vertex from vertex list in old bucket array to correct list in new bucket array
- when grid spacing increased (during vertex deletion):
  1. allocate 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 bucket may contain large number of points, `findVertex` employs oriented walk starting from first vertex in bucket's vertex list
- point location:
  - outward spiral search for nonempty bucket starting from bucket containing point
  - when nonempty bucket found, first vertex in vertex list used as search result



Tree Method

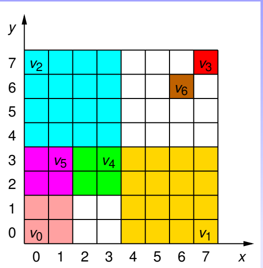
- assume triangulation domain D of form $\{0, 1, 2^S - 1\}^2$, $S \in \mathbb{N}$
- triangulation domain D hierarchically partitioned, using quadtree, into square regions called cells
- root cell of quadtree chosen as D
- remainder of cells in quadtree determined by recursively splitting root cell
-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



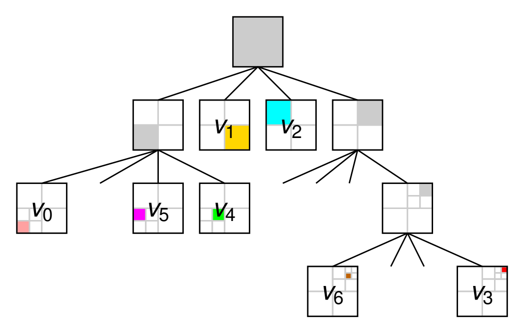
- point-location data structure is tree associated with quadtree partitioning

Tree Example

Vertices



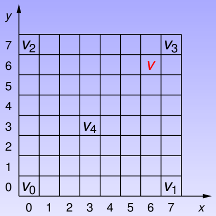
Corresponding Tree



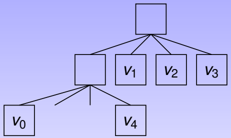
- each node in tree associated with cell in quadtree partitioning having same relative position with respect to root
- each node in tree contains pointer to DT vertex contained in node's cell (as well as pointer to node's parent and pointers to node's four children)
- for leaf node, cell always contains 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 `insertVertex` (Complex Case)

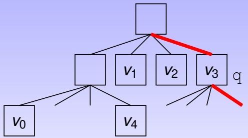
insert vertex
$v = (6, 6)$



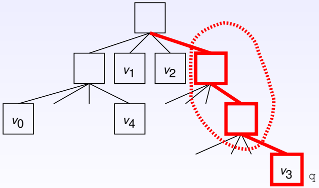
① initial state



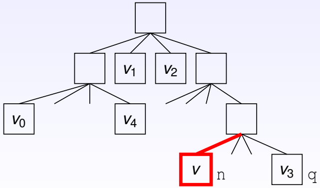
② find node `q` furthest from root whose cell contains v



③ move `q` downwards in tree until v not in cell of `q`

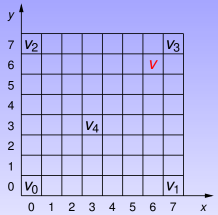


④ add node `n` corresponding to v as sibling of `q`

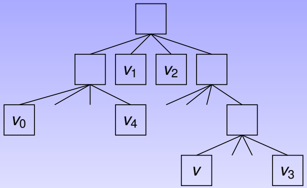


Point-Location Part of `deleteVertex` (Complex Case)

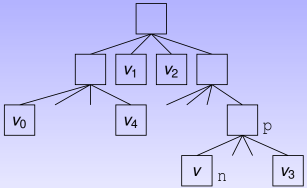
delete vertex v



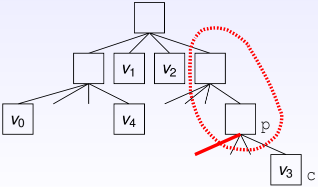
① initial state



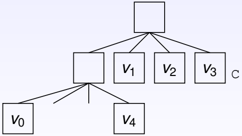
② record parent `p` of node `n` corresponding to v



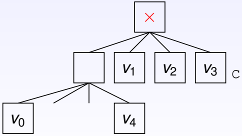
③ delete `n`; record only child `c` of `p`



④ move `c` upwards in tree

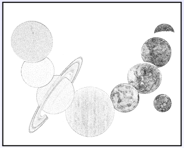


⑤ ensure no nodes on path from `c` to root reference `n`



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 1500×1867
  2. `uniform`: 104861 points, uniformly distributed, domain size 2048×2048



`planets` dataset
(rotated)

Results for `planets` and `uniform` datasets

Comparison of triangulation methods for `planets` dataset.

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

*including point-location structure

Comparison of triangulation methods for `uniform` dataset.

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

*including point-location structure

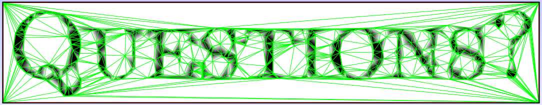
Summary of Results

- 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

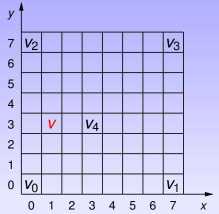
- proposed two fully-dynamic DT methods (bucket and tree methods)
- neither method superior to other in all cases
- tree method has some advantages that make its use attractive in some applications
- unlike bucket method, tree method performs well for wide variety of point distributions without need 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:

  O. Devillers. The Delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13(2):163–180, 2002

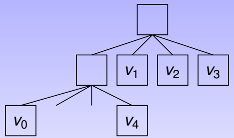

QUESTIONS?

Supplemental Slides

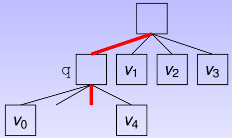
insert vertex
$v = (1, 3)$



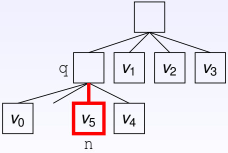
① initial state



② find node `q` furthest from root whose cell contains v

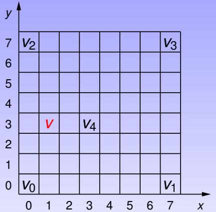


③ add new node `n` for vertex v as child of `q`

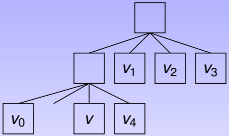


Point-Location Part of `deleteVertex` (Simple Case)

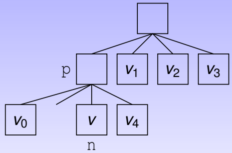
delete vertex v



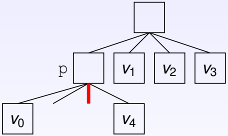
① initial state



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`