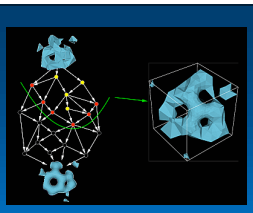
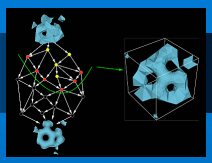


Level-Of-Detail (LOD) Data Structures for Tetrahedral Meshes

Leila De Floriani
University of Maryland, College Park (USA)
and
University of Genova, Genova (Italy)

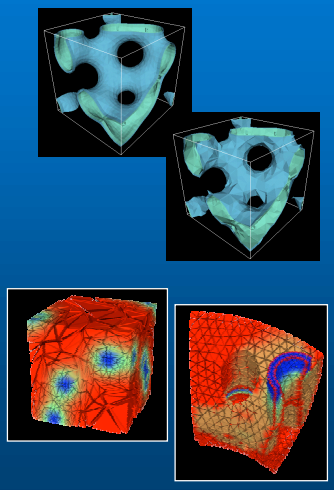
Overview



- Introduction
- Background
- Ingredients for a Level-Of-Detail (LOD) model
- Selective refinement queries on volume data sets
- Data structures for irregular meshes
- Data structures for regular meshes
- Selective refinement algorithms
- Conclusions: results and research issues

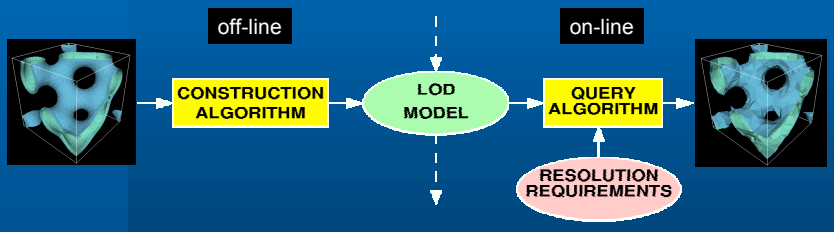
Why Level-Of-Detail (LOD) Models?

- *Aim:* modeling and visualizing *large volume data sets*
- *LOD Model:*
  - reduces the *size* of the output data set
  - allows *varying the accuracy* in different parts of the field domain
  - can be *generated off-line* through an accurate simplification algorithm
- We focus on *LOD models* based on *tetrahedral meshes*

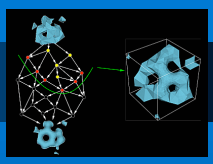


LOD Models

- They are built off-line through a simplification process (top-down refinement or bottom-up decimation)
- They encode all atomic updates performed during simplification
- They are queried on-line to extract variable-resolution meshes



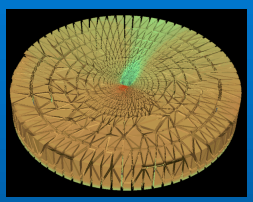
Overview



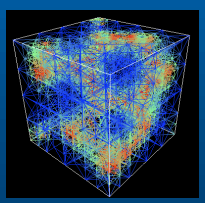
- Introduction
- Background
- Ingredients for a Level-Of-Detail (LOD) model
- Selective refinement queries on volume data sets
- Data structures for irregular meshes
- Data structures for regular meshes
- Selective refinement algorithms
- Conclusions: results and research issues

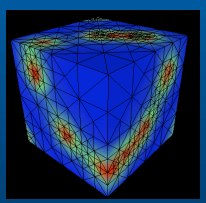
Tetrahedral Meshes

Tetrahedral mesh: connected set of tetrahedra such that
- their union covers the field domain
- any two distinct tetrahedra have disjoint interiors



Regular (structured) mesh: generated by a *recursive subdivision process* based on points on a regular grid

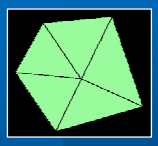




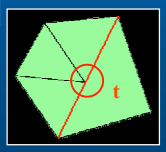
Conforming Tetrahedral Meshes

Conforming meshes: any two intersecting cells meet at a common lower-dimensional cell (face, edge, or vertex)

In 2D

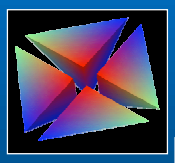


Conforming mesh

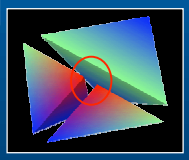


Non-conforming mesh

In 3D



Conforming mesh

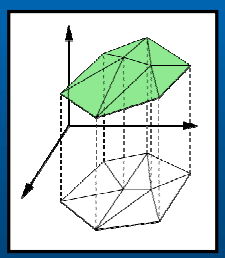


Non-conforming mesh

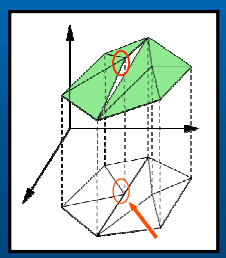
Why Conforming Meshes?

- Tetrahedral meshes are used as decompositions of the domain of a scalar field
- Conforming meshes are a way of ensuring (at least C^0) continuity in the resulting approximation

C^0 surface



Conforming mesh

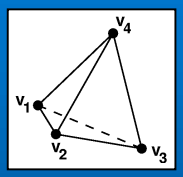


Non-continuous surface

Non-conforming mesh

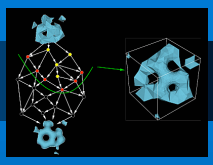
Data structures for tetrahedral meshes

Indexed data structure:
- vertex coordinates
- for each tetrahedron: links to its four vertices



Indexed data structure with adjacencies:
- vertex coordinates
- for each tetrahedron:
  - links to its four vertices
  - links to its four face-adjacent tetrahedra

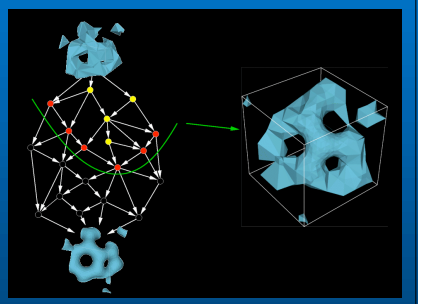
Overview



- Introduction
- Background
- Ingredients for a Level-Of-Detail (LOD) model
- Selective refinement queries on volume data sets
- Data structures for irregular meshes
- Data structures for regular meshes
- Selective refinement algorithms
- Conclusions: results and research issues

Ingredients for a LOD model

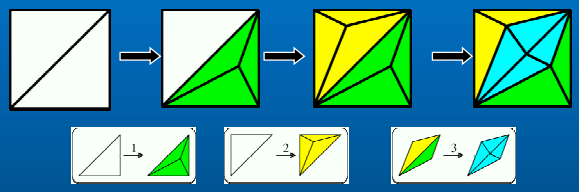
- The coarsest mesh approximation
- The collection of all updates performed on a mesh during simplification (refinement or decimation)
- A dependency relation among updates
- The *dependency relation* drives the extraction of meshes at intermediate resolutions



Ingredients for a LOD model: updates

In simplification - refinement or decimation - an initial mesh undergoes a sequence of updates.

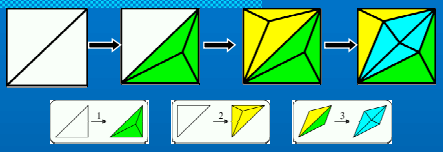
Refinement: from coarse to fine



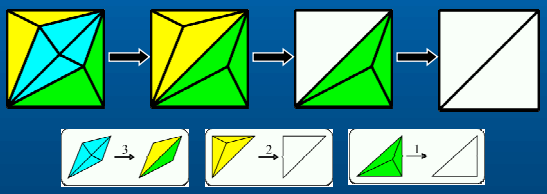
Update = replacement of a set of tetrahedra in a mesh with another set which covers the same portion of the domain

Ingredients for a LOD model: updates

Refinement



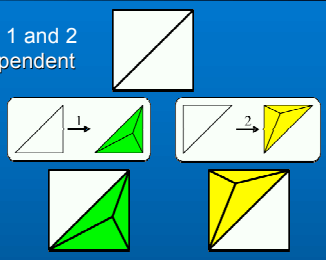
Decimation: from fine to coarse



Ingredients for a LOD model: dependency relation

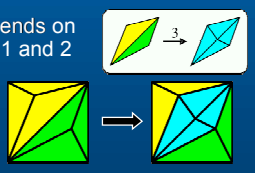
- Dependency relation between pairs of refinement updates:
  an update *B* directly depends on an update *A*
  if and only if
  some tetrahedron introduced by *A* is removed by *B*

Updates 1 and 2 are independent



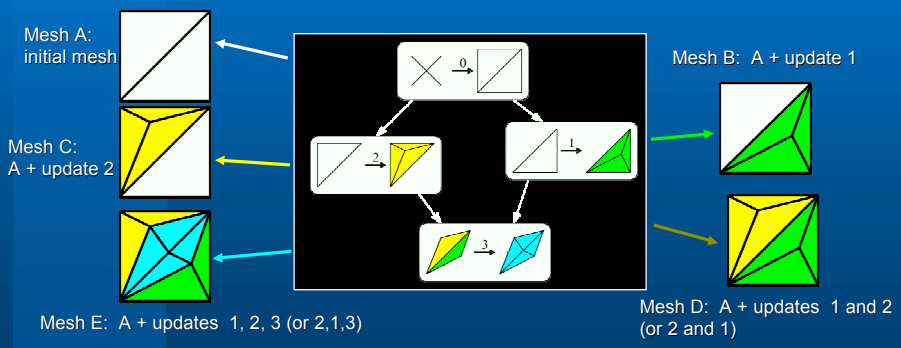
- The transitive closure of the dependency relation defines a partial order

Update 3 depends on both updates 1 and 2

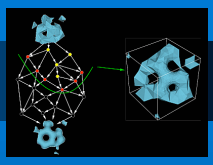


LOD Models: properties

From a LOD model , we can extract any mesh obtained from the coarsest mesh by applying any sequence of refinement updates compatible with the dependency relation



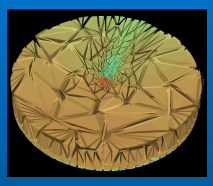
Overview

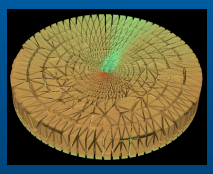


- Introduction
- Background
- Ingredients for a Level-Of-Detail (LOD) model
- Selective refinement queries on volume data sets
- Data structures for irregular meshes
- Data structures for regular meshes
- Selective refinement algorithms
- Conclusions: results and research issues

Selective refinement queries [Cignoni at el., 2003]

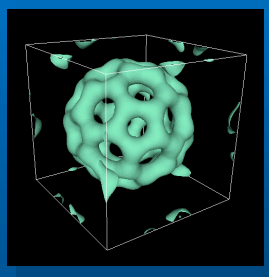
- A set of basic queries for analysis and visualization of a volume data set at different levels of detail
- All instances of selective refinement:
  extract from a LOD model a mesh with the smallest possible number of tetrahedra satisfying some user-defined criterion based on LOD
- LOD depends on approximation error



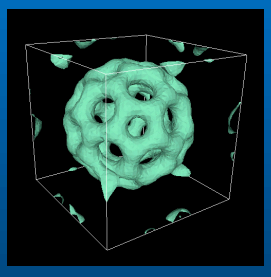


Uniform LOD

Input parameters: an accuracy threshold E
Error(t)□E for every tetrahedron t in the extracted mesh

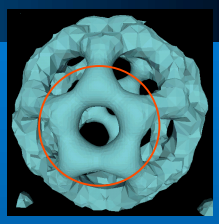


Buckyball
12.5 million tetrahedra



E = 5% of the field range
274,460 tetrahedra

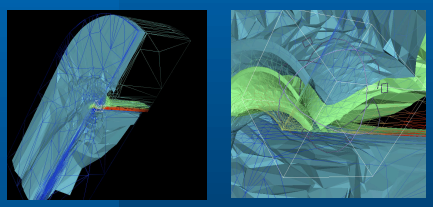
Variable LOD based on spatial location



Input parameters: a *Region of interest* (*ROI*) *R* and *two accuracy thresholds* E_1 and E_2 *(E_1<E_2):*

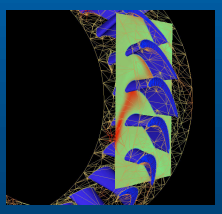
- Error(t) $\leq E_1$ for each tetrahedron t interesecting R
- Error(t) $\leq E_2$ for any other tetrahedron

ROI = box
E=0.01% in the ROI
size = 1/3 of reference mesh



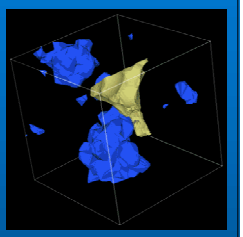
BluntFin: 222,528 tetrahedra

ROI =cross plane
E= 2% in the ROI
size = 7% of reference mesh



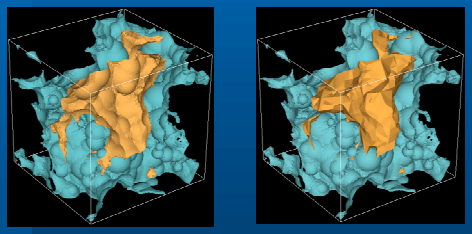
Turbine Blade 576,566 tetrahedra

Variable LOD based on field values



Input parameters: a *collection of field values FV* and *two accuracy thresholds* E_1 and E_2 (E_1<E_2):

- Error(t) $\leq E_1$ for every tetrahedron t interesecting the isosurfaces of values in FV
- Error(t) $\leq E_2$ for any other tetrahedron

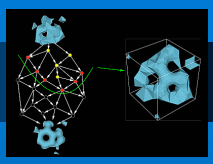


Plasma: 1,500,282 tetrahedra

E = 0.1% along the blue isosurface

25% size of the mesh at uniform LOD with error = 0.1%

Overview



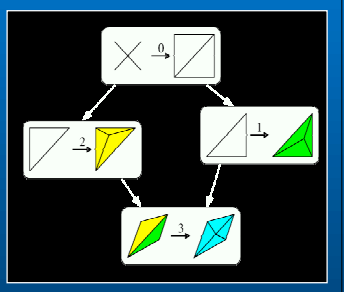
- Introduction
- Background
- Ingredients for a Level-Of-Detail (LOD) model
- Selective refinement queries on volume data sets
- Data structures for irregular meshes
- Data structures for regular meshes
- Selective refinement algorithms
- Conclusions: results and research issues

LOD data structures for irregular meshes

- Graph-based data structures versus tree-based data structures:
  - Graph-based data structures: dependency relation described by a *DAG* (Directed Acyclic Graph)
  - Tree-based data structures: dependency relation described by a *tree* and by a numbering scheme
- Explicit versus implicit representation of the updates:
  - Explicit representations:
    - all tetrahedra affected by an update are encoded *explicitly*
  - Implicit representations:
    - *procedural description* of the update

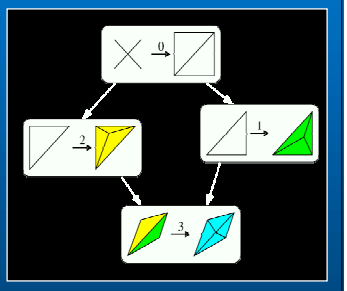
An explicit data structure [De Floriani, Magillo, Puppo, 1997]

- Dependency relation encoded as a DAG
- Each tetrahedron created by any update is encoded once through the indexes to its four vertices
- For each update u:
  - number and indexes of the tetrahedra deleted by u
  - number of tetrahedra created by u and index of the first tetrahedron created by u
- Dimension-independent implementation in a C++ library available in the public domain
  `http://www.disi.unige.it/person/MagilloP/MT/`



An explicit data structure: storage cost

- 360n bytes when built through half-edge collapse
- 3.5 times the cost of encoding the mesh at full resolution in an indexed data structure
- 1.8 times the cost of encoding the mesh at full resolution in an indexed data structure with adjacencies
- Remark: cost of encoding error information has not been considered

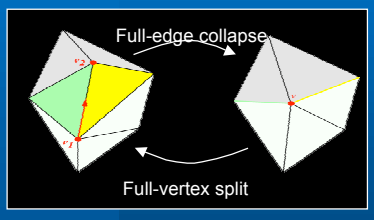


Implicit LOD data structures for irregular meshes

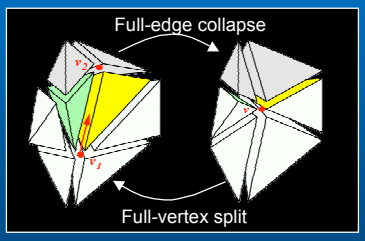
- LOD model built through full-edge collapse (Full-Edge MT) [Cignoni et al., 2003]:
  dependency relation represented as a binary tree
- LOD model built through half-edge collapse (Half-Edge MT) [Danovaro and De Floriani, 2002]:
  dependency relation represented as a DAG

Full-edge collapse/vertex split

- Contract an edge $e = (v_1, v_2)$ to a new vertex v: $e = (v_1, v_2) \rightarrow v$
- (Full-)vertex split: inverse of full-edge collapse



In 2D

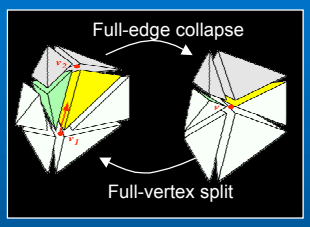


In 3D

On average, 33 tetrahedra are modified and 6 of them are collapsed into triangles

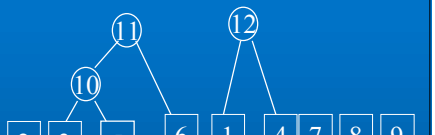
Full-Edge Multi-Tessellation (MT) [Cignoni et al., 2003]

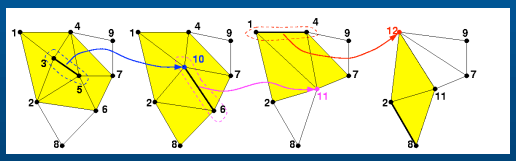
- Based on full-edge collapse
- Updates:
  - full-edge collapses to the edge mid-point and full-vertex splits
- Dependency relation:
  - view-dependent tree [El Sana and Varshney, 1999] - implicit encoding of the dependency relation specific for full-edge collapse



Encoding a view-dependent tree

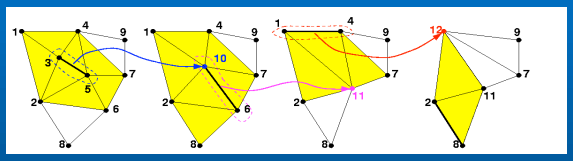
- A *binary tree* used to encode the dependency between vertex v and vertices v_1 and v_2 where $e=(v_1,v_2)$ v
- All dependencies captured through vertex enumeration:
  - vertices of the initial mesh (leaves in tree) are *arbitrarily* numbered
  - the other vertices (internal nodes) are numbered in *progressive order* of their creation

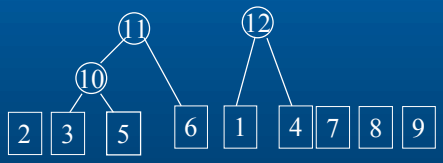




View-dependent tree: vertex-split rule

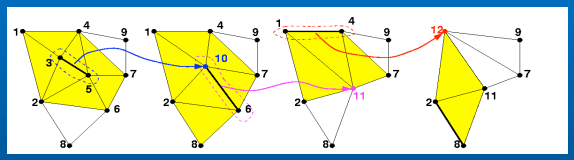
- Vertex v can be split into edge (v_1,v_2) only if all adjacent vertices of v have a label less than v.

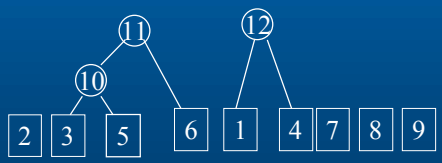




View-dependent tree: edge-collapse rule

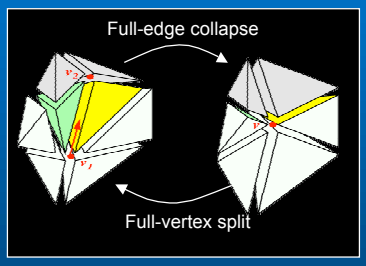
- Edge (v_1,v_2) can be collapsed into a vertex v only if all vertices adjacent to v_1 and v_2 are roots, or their parents have a label greater than v.





Encoding a full-edge collapse/vertex-split

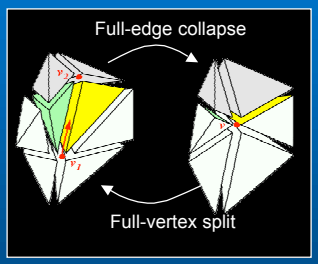
- Geometric information: offset values *for coordinate* and *field values*
- Error information
- Topological information (for performing vertex split): one bit for each tetrahedron t in the star of v :
  - bit=0 if t becomes incident in v_1
  - bit=1 if t becomes incident in v_2
- Any face common to tetrahedra marked differently expands into a tetrahedron incident at edge e=(v_1, v_2)



In vertex split, yellow and green faces become tetrahedra

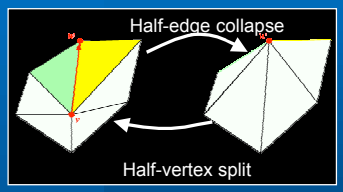
Full-edge MT: storage cost

- Updates:
  - Offset values *for coordinate* and *field values:* 8n bytes
  - *Topological information* 8n bytes for the bitstream (at most 64 tetrahedra in the star of v):
- View-dependent tree: 12n bytes
- Total cost: 28n bytes
- With respect to mesh at full resolution:
  - 33% wrt indexed representation
  - 15% wrt indexed + adjacencies
- Remark: cost of encoding error information has not been considered

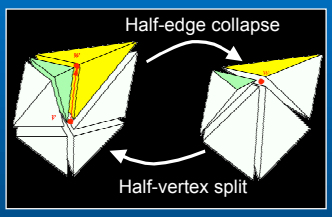


Half-edge collapse/vertex split

- Contract an edge *e =(v,w)* to vertex *w: e =(v,w)* *w*
- (Half-)vertex split: inverse operation insertion of vertex v



In 2D

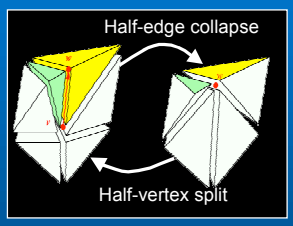


In 3D

On average, 16 tetrahedra are modified and 5 of them are collapsed into triangles

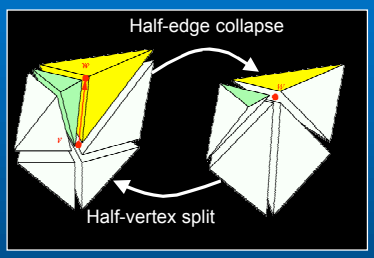
Half-Edge Multi-Tessellation (MT)
[Danovaro et al., 2002]

- Based on half-edge collapse
- Updates:
  half-edge collapses and half-vertex splits
- Dependency relation:
  - compact DAG encoding [Klein and Gumhold, 1998]
  - each arc in the DAG is encoded once



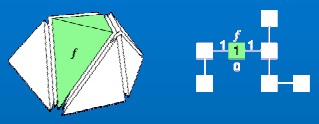
Encoding half-edge collapse and half-vertex split

- Geometric information: coordinates and field value at vertex *v*
- Error associated with update
- Topological information:
  - for *half-edge collapse (v, w)* → *w*
    vertex *w* encoded as:
    - a tetrahedron t in the star of v
    - index of *w* inside t
  - for *half-vertex split w* → *(v, w)*
    the region of influence R: the portion of the star of w affected by half-edge split
    - starting face f: a tetrahedron t in the star of v + index of f in t
    - traversal of the tetrahedral partition of R encoded as a bit stream



Encoding the region of influence

Encoding a breadth-first traversal of the dual graph of the tetrahedral partition of the region of influence

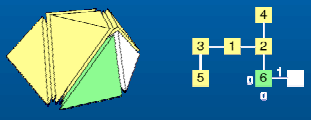


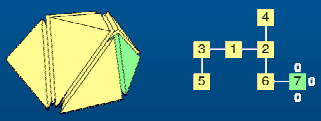
Traversal starts at the encoded tetrahedron t entering t from face f

The three remaining faces f' of each visited tetrahedron t' are labeled as follows:

label(f') =1 if the tetrahedron adjacent to t' along f' must be visited

label(f') =0 otherwise





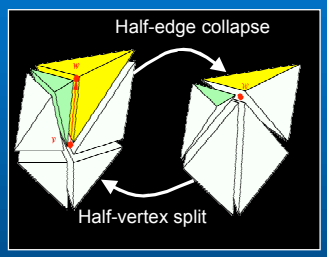
Current tetrahedron: green
Visited tetrahedra: yellow
Tetrahedra to be visited: white

Half-Edge MT: storage cost

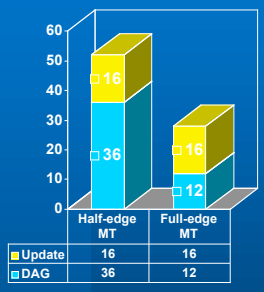
- Update: 16n bytes
  - Coordinates and field of v: 8n bytes
  - Connectivity information: 8n bytes
    - bitstream : 45n bits
    - starting face (7 bits) + vertex w (7 bits): 14n bits
- DAG: 36n bytes
- Total cost: 52n bytes

- With respect to mesh at full resolution:
  - 50% wrt indexed representation
  - 25% wrt indexed + adjacencies

- Remark: cost of encoding error information has not been considered

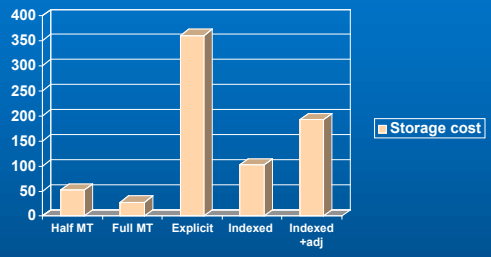


Half-edge and full-edge MT:storage costs



	Half-edge MT	Full-edge MT
Update	16	16
DAG	36	12

Full-edge MT: 53% of space of Half-edge MT



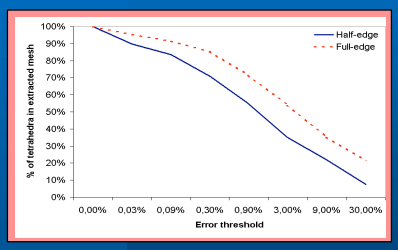
Comparisons with explicit data structure and mesh at full resolution

Half-edge versus full-edge MT: size of the extracted meshes

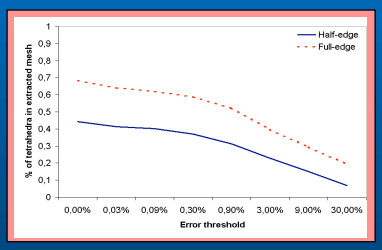
Data from 7 different data sets

On x axis: error thresholds as a percentage of the field range

On y axis: ratio between number of tetrahedra in the extracted mesh and the mesh at full resolution



Uniform LOD

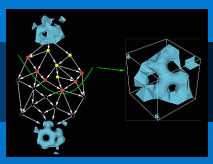


Variable LOD based on field values

Half-edge versus full-edge MT

- Size of the meshes extracted from a Half-edge MT versus size of the meshes extracted from a Full-edge MT on average:
  - *30%* at uniform LOD
  - *68%* at variable LOD
- In a Half-edge MT, updates are of smaller size: on average, 11 tetrahedra affected by a vertex split (half-edge) versus 27 (full-edge)
- Our current work:
  - a modified view-dependent tree for half-edge collapse: about 65% of the space required by our current DAG implementation
  - a new encoding for half-edge collapses (consistent with the view-dependent tree implementation)

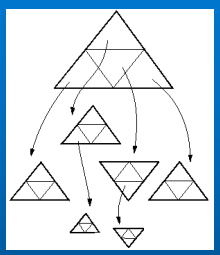
Overview



- Introduction
- Background
- Ingredients for a Level-Of-Detail (LOD) model
- Selective refinement queries on volume data sets
- Data structures for irregular meshes
- Data structures for regular meshes
- Selective refinement algorithm
- Conclusions: results and research issues

LOD models for regular meshes

- Models generated by red/green tetrahedron refinement [Bank'83;Bey'95; Grosso'97;Zhang'95]
  - based on the recursive partition of a tetrahedron into eight tetrahedra
  - *Drawback:* non-conforming meshes are generated and must be made conforming through specific irregular refinement rules



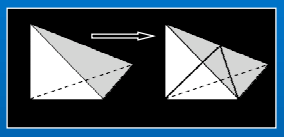
In 2D

LOD Models for regular meshes

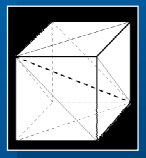
- Models generated by tetrahedron/octahedron refinement [Greiner'00;Chang'03]
  - recursive partition of the space into tetrahedra and octahedra
  - *objective:* only two congruent shapes
- Models generated by regular tetrahedron bisection [Rivara'92; Zhou'97; Ohlberger'97; Gerstner'99-00;Nielson'00; Lee'00; Pascucci'02; Gregorski'02]

Regular tetrahedron bisection

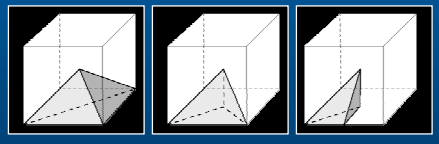
A tetrahedron is bisected along its longest edge



Tetrahedron bisection applied recursively to a *cubic* domain generates *three* classes of *congruent* tetrahedral shapes



1/2 pyramid 1/4 pyramid 1/8 pyramid

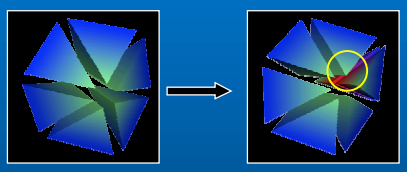


Regular tetrahedron bisection

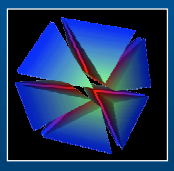
The tetrahedra which are split at the same time form a diamond

Tetrahedron bisection may generate non-conforming meshes

diamond



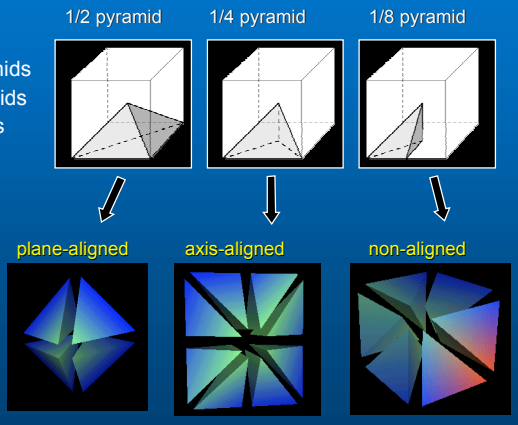
To generate conforming meshes, all tetrahedra sharing an edge must be split at the same time



Diamonds

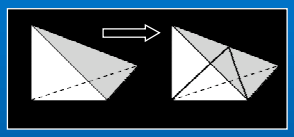
Three kinds of *diamonds*

- plane-aligned: four 1/2 pyramids
- axis-aligned: eight 1/4 pyramids
- non-aligned: six 1/8 pyramids

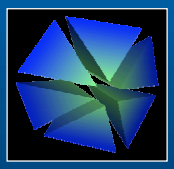


LOD data structures based on tetrahedron bisection

- Hierarchy of Tetrahedra:
  binary forest of tetrahedra encoding the nested structure defined by tetrahedron bisection

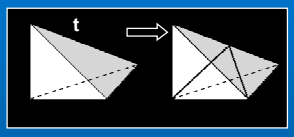


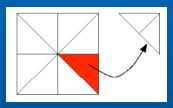
- DAG of diamonds:
  - DAG of tetrahedra clusters which are split at the same time to produce a conforming mesh
  - DAG encodes the dependency relation among diamonds



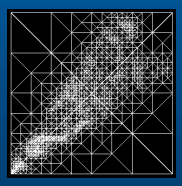
Hierarchy of Tetrahedra (HT)

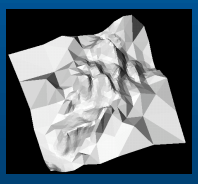
- Each *node* in the hierarchy corresponds to a *tetrahedron* in the nested domain partition
- The children of a node t are the tetrahedra obtained by splitting t along its longest edge





It extends the hierarchy of right triangles used in terrain modeling

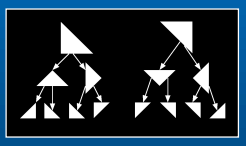




Hierarchy of Tetrahedra (HT): a data structure

- Vertex coordinates and connectivity information are implicit
- Only field values and error information need to be encoded

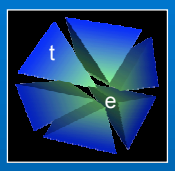
In 2D



- Data structure:
  - a table of *field values*
  - a forest of *six full binary trees* encoded as an array: each element contains the *approximation error(s)*
  - alternatively, errors can be associated with vertices

Computing diamonds from a HT

- When extracting a mesh from a HT we need to compute diamonds on-the-fly
- Problem: given a tetrahedron t and an edge e of t, find all the tetrahedra sharing edge e with t



It requires traversing the diamond of t by navigating through face-adjacent neighbors

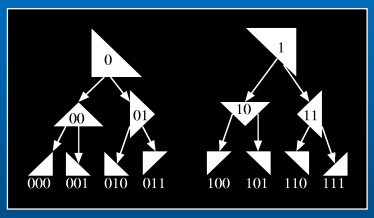
Sub-problem: find the *face-adjacent neighbors* to a given tetrahedron

We apply the traditional approach to neighbor finding in a quadtree combined with the use of location codes

Location codes in a HT

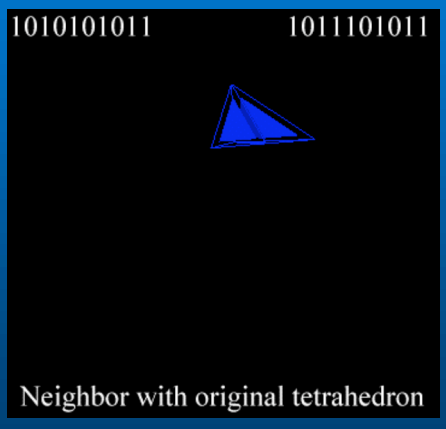
- Location code for a tetrahedron t:
  - *level* of t in the tree
  - *path* from the root to t
- Location codes are *not* explicitly stored, but they are computed on-the-fly when computing diamonds

Location codes for a hierarchy of right triangles



Finding a face-neighbor: an example

Input: location code of a tetrahedron (1010101011) and one of its faces



Output: Location code of the neighbor: 1011101011

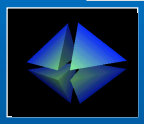
Finding neighbors in constant time
[Lee, De Floriani and Samet, 2001]

- Basic steps of the algorithm: given a tetrahedron t and a face f of t
  - find *nearest common ancestor* of t and of its neighbor along face f
  - *reflect the path* to the input node to obtain the path to the neighbor inversion of *just one bit* corresponding to the child of the common ancestor
- To find the nearest common ancestor:
  - use of bit manipulation and binary arithmetic on the location code of t
  - no need to perform tree traversal

DAG of diamonds [Pascucci'02]

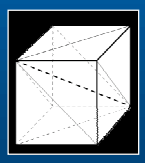
- Nodes: *diamonds*
  - at level 3j non-aligned diamonds
  - at level 3j+1 plane-aligned diamonds
  - at level 3j+2 axis-aligned diamonds
- Root: non-aligned diamond describing the initial cubic subdivision
- Arcs: encode the parent-child relation among diamonds

non-aligned



plane-aligned

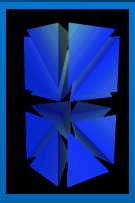
axis-aligned



initial cubic subdivision

DAG of diamonds: parent/child relation

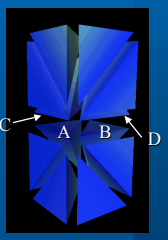
- Parents of a diamond □: diamonds that must be split to create the tetrahedra of □
- Children of □ : diamonds created when □ is split



The two parents of □

Diamond □ : plane-aligned (4 tetrahedra)

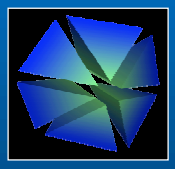
Tetrahedra E, F, G, H in □ are generated by splitting tetrahedra A,B,C and D in the parents



Diamond □

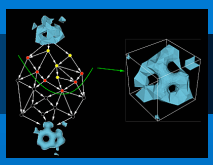
DAG of diamonds: data structure [Gregovski'02]

- A *diamond* is associated with a *vertex in the regular grid* of index (i,j,k)
- Diamond:
  - (i,j,k) index + diamond type
  - approximation error, min and max scalar field values, and gradient encoded for each diamond
  - storage cost: *3 bytes* per diamond



- Parents and children are not stored, but computed on-the-fly

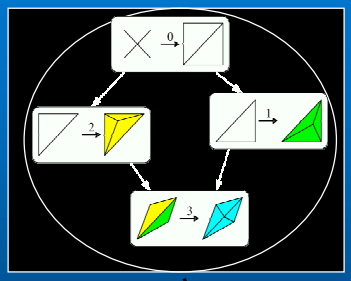
Overview



- Introduction
- Background
- Ingredients for a Level-Of-Detail (LOD) model
- Selective refinement queries on volume data sets
- Data structures for irregular meshes
- Data structures for regular meshes
- Selective refinement algorithms
- Conclusions: results and research issues

Algorithms for selective refinement

- Selective refinement:
  - *extract a mesh* with the *smallest* possible number of tetrahedra satisfying some user-defined *criterion based on LOD*
- Any extracted mesh corresponds to a *consistent set of updates* in the LOD model
- A consistent set S of updates: if an update u belongs to S, all updates that precedes u must be in S



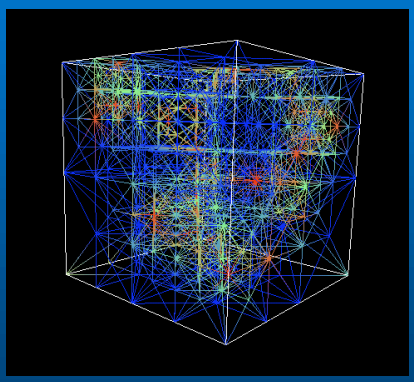
Updates 1, 2 and 3 form a consistent set

Extracted mesh

Algorithms for selective refinement

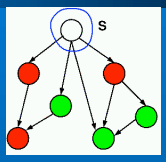
- Based on *traversal of the DAG or of the tree* describing the dependency relation among updates
- They construct a consistent set S of updates
- The mesh associated with such set S is the solution to the query
- Two approaches:
  - top-down: it generates a mesh from scratch
  - incremental: it modifies a previously extracted mesh by locally refining or by locally coarsening it

Top-down approach: an example



Top-down approach

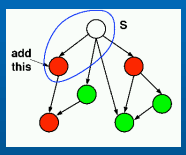
Input: LOD model and LOD criterion
Output: mesh satisfying LOD criterion

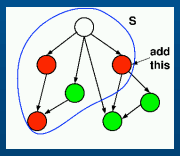


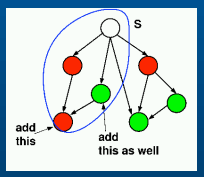
Red updates need to be performed to satisfy the LOD criterion

Initialization step: set S ← coarsest mesh
Generic step:

- Add a *red update* u to S
- If set S ∪ {u} is not consistent, then recursively add all updates preceding u on which u depends



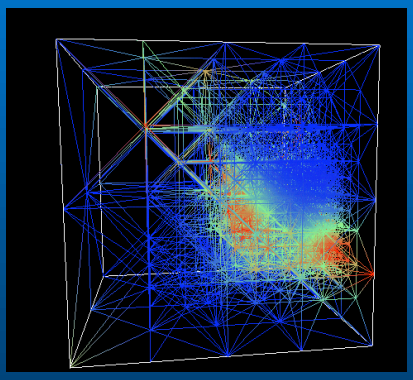




Top-down approach: implementation issues

- Efficiently implemented as a depth-first traversal
- Often implemented by using a priority queue (ordered according to decreasing error values): updates having larger error values are performed first
  - the resulting algorithm is *interruptible* (i.e., intermediate steps produce "balanced" approximations of the resulting mesh)
  - it uses *extra storage* for encoding the priority queue (in our experiments on a hierarchy of tetrahedra: *size of the queue* = 10-16% of the size of the extracted mesh)

Incremental approach: an example



Incremental approach

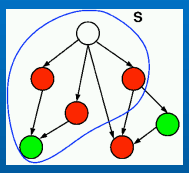
Input: LOD model, previously extracted mesh M, and new LOD criterion

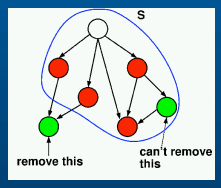
Initialization step: set S □ set of updates corresponding to M

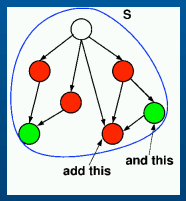
Generic step:

- Add *red updates* to S, and *all* those updates that are necessary to maintain S a consistent set
- Remove *green updates* if possible

Red updates must be performed to satisfy the new LOD criterion
Green updates are not necessary to satisfy the new LOD criterion







Incremental approach: implementation issues

- Direct algorithm: all refinement updates which do not satisfy the new LOD criterion are performed first, and the coarsening updates whenever feasible.
- Priority-based algorithm [Cignoni'03, Gregorski'02]: interleaving refinement and coarsening updates by using
  - a *split queue* containing candidate refinement updates
  - a *merge queue* containing candidate coarsening updates
- A priority-based algorithm is *interruptible* and the size of the extracted mesh can be effectively used as a parameter

Selective refinement on a Hierarchy of Tetrahedra (HT)

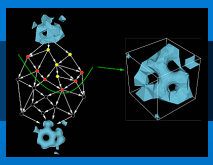
- Selective refinement algorithms on a HT require computing diamonds on-the-fly to generate conforming meshes
- Two approaches:
  - neighbor finding
  - error saturation
- Error saturation:

  the simultaneous split of all tetrahedra in a diamond is forced by the error values assigned to the tetrahedra in the hierarchy

Selective refinement on a HT: some results

- Comparisons on queries at uniform LOD between
  - depth-first algorithm using constant-time neighbor finding
  - depth-first algorithm on a HT with error saturation
- *Results*:
  - size of the extracted meshes: mesh extracted from a HT with error saturation contains 5% more tetrahedra, on average
  - same *number of tetrahedron splits per second*

Overview



- Introduction
- Background
- Ingredients for a Level-Of-Detail (LOD) model
- Selective refinement queries on volume data sets
- Data structures for irregular meshes
- Data structures for regular meshes
- Selective refinement algorithms
- Conclusions: results and research issues

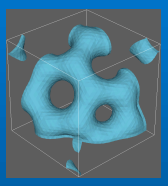
Results: regular versus irregular data structures

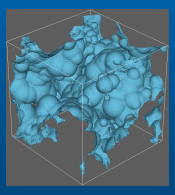
- Storage cost: regular data structures considerably more compact than irregular data structures: coordinates and structural information are implicit

- Selectivity:
  - Half-Edge MT more selective than HT at uniform LOD
  - Half-Edge MT less selective at variable LODs

  Experiments performed on *HT* and *Half-edge MT* with error on tetrahedra and with error on updates

Small-Bucky



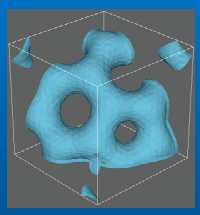


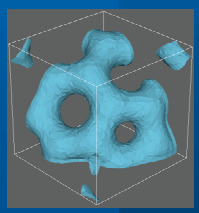
Plasma

Uniform LOD

Meshes at uniform LOD with error = 0.1%

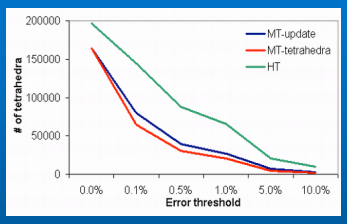
From a *HT*:
45% size of mesh at full resolution

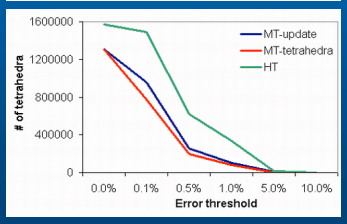




From a *Half-edge MT*:
24% size of mesh at full resolution

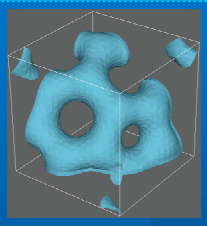
Smallbucky



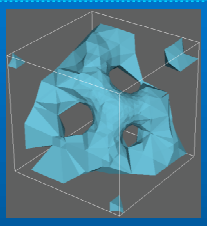


Plasma

Variable LOD based on spatial location



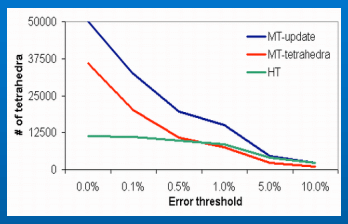
Mesh at uniform LOD with error =0.1%

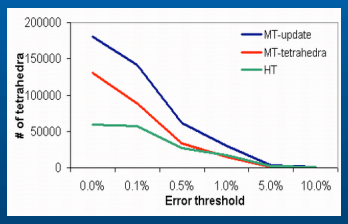


Mesh at variable LOD with error =0.1% only in the box

Size of the extracted mesh on the right = 6.2% mesh at uniform LOD with error 0.1%

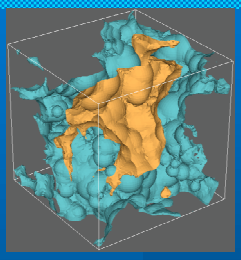
Smallbucky



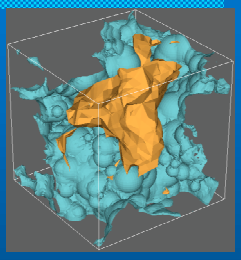


Plasma

Variable LOD based on field values



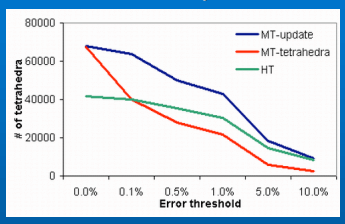
Mesh at uniform LOD with error =0.1%

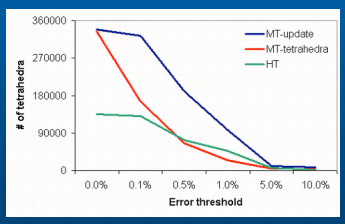


Mesh at variable LOD with error =0.1% only along the blu iso-surface

Size of the extracted mesh on the right = 25% mesh at uniform LOD with error 0.1%

Smallbucky





Plasma

Some research issues

- Dealing with massive irregular meshes:
  - *out-of-core* simplification algorithms
  - *out-of-core* LOD data structures
  - *out-of-core* algorithms for selective refinement
- From *geometry-based* to *structural* LOD models:
  - combining a LOD representation with a morphological description of the field
  - generating LOD models through simplification algorithms guided by the field morphology
- LOD models for *four and higher-dimensional scalar fields*

Acknowledgements

- *University of Genova*:
  Emanuele Danovaro, Paola Magillo and Enrico Puppo
- *Italian National Research Council:*
  Paolo Cignoni and Roberto Scopigno
- *University of Maryland*:
  Michael Lee and Hanan Samet