Computing intersection areas of overlaid 2D meshes

W. Randolph Franklin[1] , Salles V. G. Magalhães[2]
[1]Rensselaer Polytechnic Institute, USA , Universidade Federal de Viçosa, Brazil[2]

Intersecting maps

Algorithm: PAROVER2

- Input: 2 polygonal maps (planar graphs) M_0 and M_1.
- Objective: Efficiently compute the area of every nonempty intersection of a face (polygon) from map M_0 and a face from M_1.
- Applications in CAD, GIS, statistics, etc.
- Example: estimate populations of water resource polygons from state populations.
- Challenges
  - Special cases
  - Big datasets

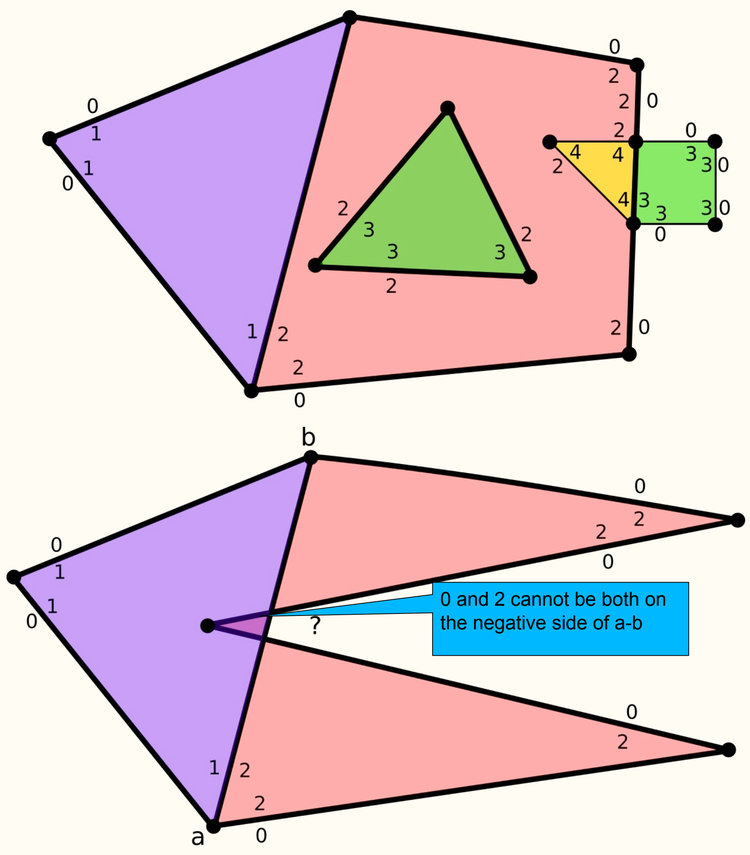
Source: USGS

Novelties

- Expected linear execution time
- Grid indexing: efficient parallel uniform grid
- Computation with local information.
- Parallel: for multi-core computers
- Simple flat data structures: good for GPUs.
- Implemented using a functional paradigm with NVIDIA's Thrust library:
  - High-level implementation
  - Can be targeted to different backgrounds: OpenMP, TBB, sequential, CUDA.

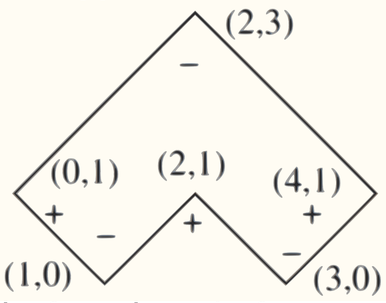
Data representation

- "Soup" of edges:
  - Oriented edges: { $(x_0,y_0,x_1,y_1,f_l,f_r)$ }.
  - f_l and f_r are the ids of the two faces bounded by the edge.
- Supports:
  - ✓ Multiple components
  - ✓ Nested components
  - × Self intersections ▭ contradictions



Local topological formulae

- Power of local topological formulae:
- Area of a face from either:
  - Set of vertex-edge incidences
  - Vertex positions and neighborhood, e.g.
    - Restricting edge slopes: 1 or -1
    - Each vertex: sign bit (neighborhood)
    - Area = $\sum s_i x_i^2$ = 0 - 1 + 4 - 9 + 16 - 4



- Similar formulae exist for general polygons.
- PAROVER2 uses following formula:
  - Each vertex v has two adjacencies
  - $\hat{t}$: the unit direction along the edge
  - $\hat{n}$: unit direction perpendicular to v and $\hat{t}$ pointing to the inside of the edge

$$A = \sum (v \circ \hat{t})(v \circ \hat{n}) \div 2$$

- Requires little information → easy to compute (and parallelize)

Outface area computation

- Performed with a map-reduce
- Process input and compute each output vertex with all its vertex-edge adjacencies. Two types:
  - Adjacency of one of the input maps.
  - Adjacency generated by an intersection of a pair of edges from the two input maps.
- Outface areas are accumulated.

Algorithm: output adjacencies that are input adjacencies

1. Input adjacency: h , w.l.o.g., h in M_0
2. h is adjacent to two faces f_l, f_r
3. Vertex of h: v –– v is in a face f' of M_1
4. h is part of two outfaces: (f_l,f') and (f_r,f')
5. Normal vector of h may need to be negated
6. Area component (*outface-id, area*) computed for each outface
7. Total area of each outface obtained by summing its components

- Nontrivial parts:
  - Special cases
  - Storing area components
  - Point location

Storing area components

- Challenges:
  - Computing and storing components in parallel.
  - Ids of non-empty outfaces: unknown in advance
  - How many components each outface has: unknown in advance

Storing area components

1. Size of the vector: 4x the number of input edges in the two meshes combined.
2. The i-th input edge will create output pairs numbered $4i$ to $4i$+3 → no need for synchronizations.
3. Vector sorted by outface-id and reduced-by-key.

(slowest step: sort – no better alternative found yet)

Point location

- Input pre-processed in linear time. Queries performed in expected constant time.

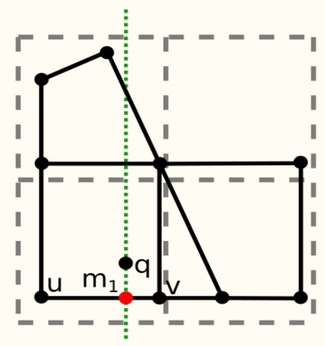
Point location: pre-processing

1. Create a g x g grid as index. g is chosen s.t. a cell could contain the largest face.
2. For each edge e, compute the superset of size 4 of the grid cells that intersect e. (for simplicity: use a bounding-box)
3. In parallel: create a vector of pairs (cell, edge)
4. Sort it by cell id
5. Use a parallel scan to find the start of each cell in the vector

Point location: query

Input: query point q from M_0 (wlog.)

1. Compute grid cell c containing q.
2. Find the edge e from M_1 in c that intersects a vertical ray from q at the closest point.
3. If e does not exist → q is in the exterior.
4. Otherwise, q is in one of the two faces adjacent to e.



Adjacencies that are intersections

- Computing the adjacencies generated by the intersection of two input edges: differs in two ways from previous case.
  1. Intersections of edges have to be found. One intersection: vertex of four outfaces and eight adjacencies.
  2. Point location is not necessary (can be determined by the intersection).

Algorithm: output adjacencies that are intersections of two input edges

1. Compute the maximum possible number of edge intersections per cell and allocate a vector V for all intersections.
2. Populate V with the pairs of possibly intersecting edges. The i-th element in V can be found in O(1) time.
3. Filter V by whether or not the edges do intersect.
4. Map the resulting vector into a vector of octuples of outface adjacencies.
5. Sort, reduce by key, and sum.

Performance experiments

- Test data: overlapping square meshes.
- Dual 14-core 2.0 GHz Xeon, 256 GB of RAM
- NVIDIA's Thrust + OpenMP backend
- Parallel speedup of 6.3x (Turbo Boost reduces the speedup)
- Validate output by looking at computed areas.

# input edges	# output faces	Elapsed time (sec)
220	400	0.023
3,720	3,600	0.032
40,400	40,000	0.082
361,200	360,000	0.47
4,004,000	4,000,000	6.2

Conclusions and future work

- Simple fast algorithm for computing the area of intersections.
- High-level functional programming style: easily (?) portable code.
- Extension to 3D: volume of intersecting polyhedra.

$$V = \sum (v \circ \hat{t})(v \circ \hat{n})(v \circ \hat{b}) \div 6$$

- Small conceptual extension, but practical challenges.
- We`ve been developing other software with similar ideas. E.g. 3D-EPUG-Overlay, Union3, PinMesh

Acknowledgements

CAPES