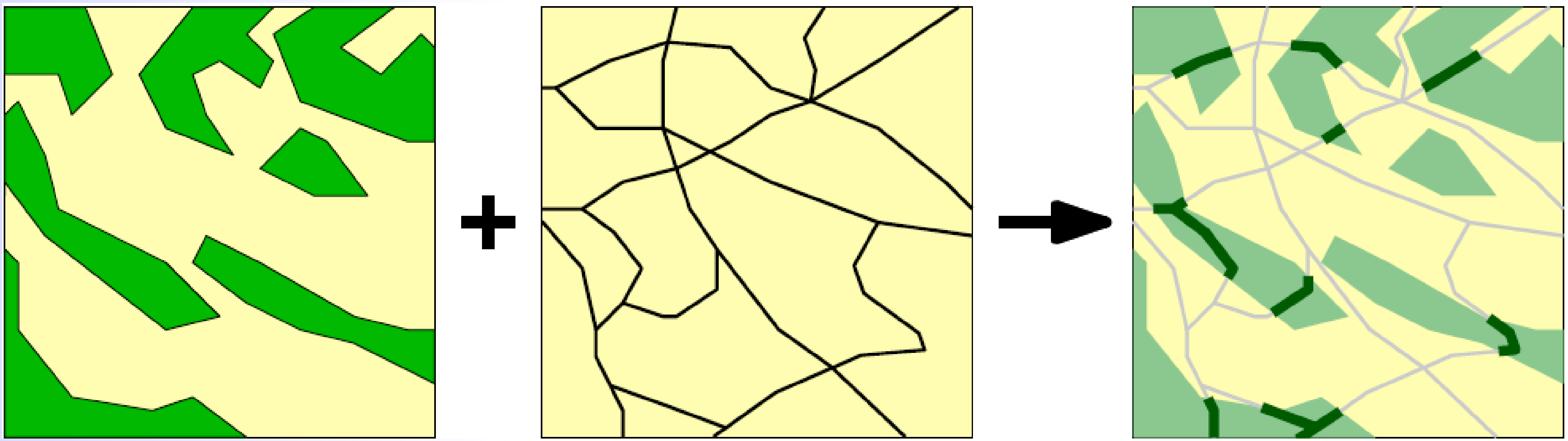
I/O-Efficient Indexes for Fat Triangulations and Low-Density Subdivisions

Mark de Berg Herman Haverkort Shripad Thite Laura Toma

Laura Toma
Bowdoin College
2009

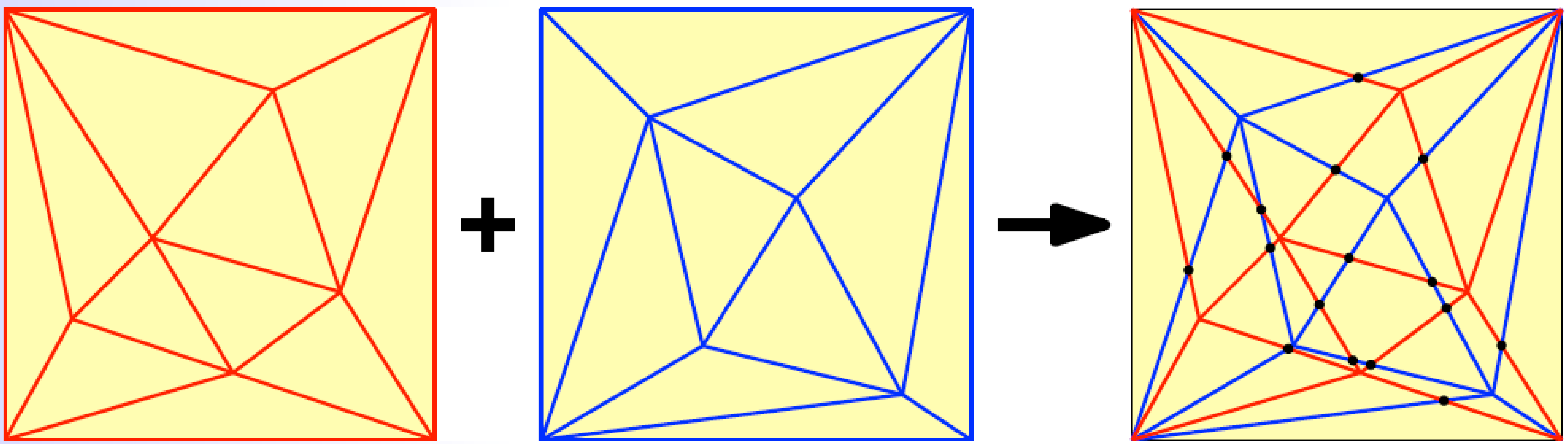
Map Overlay

- Maps: planar subdivisions, sets of non-intersecting line segments,..



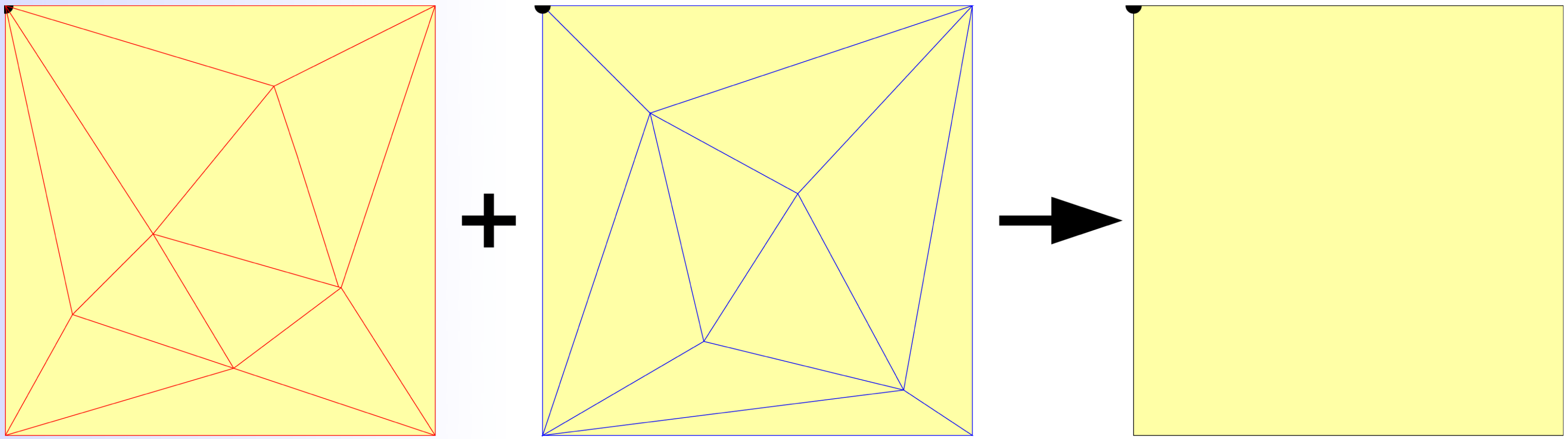
Map Overlay

- Maps: ..., triangulations



Overlaying Triangulations CPU-Efficiently

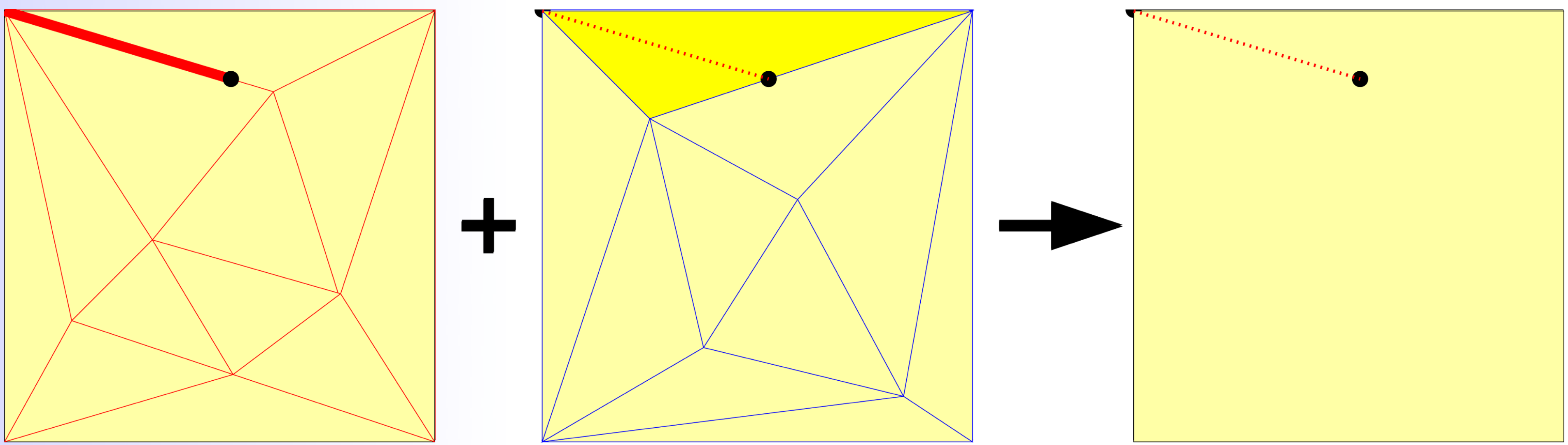
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

Overlaying Triangulations CPU-Efficiently

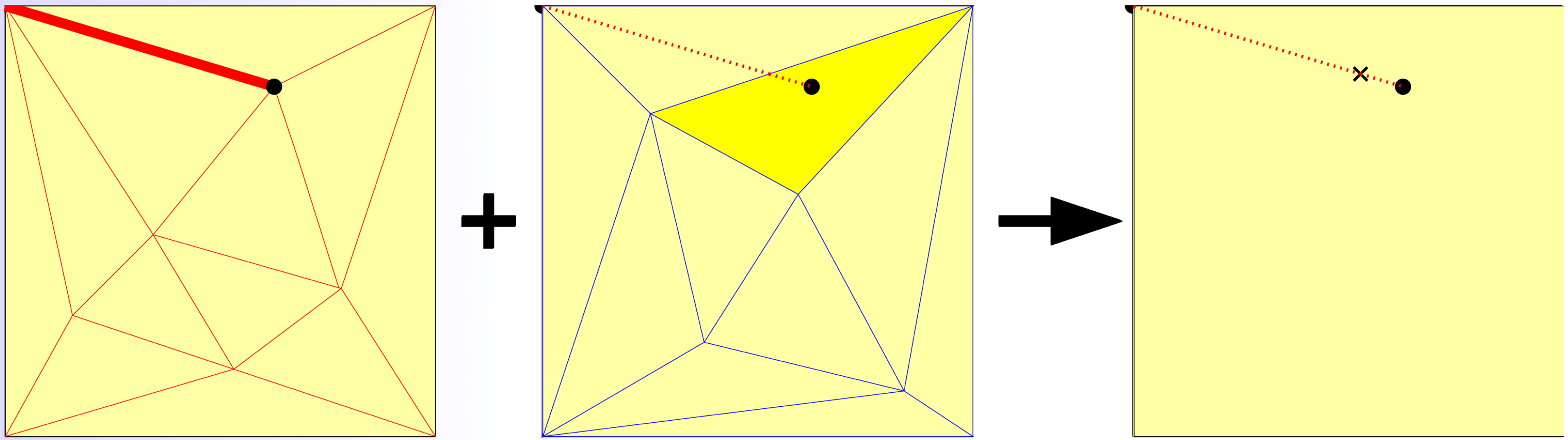
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

Overlaying Triangulations CPU-Efficiently

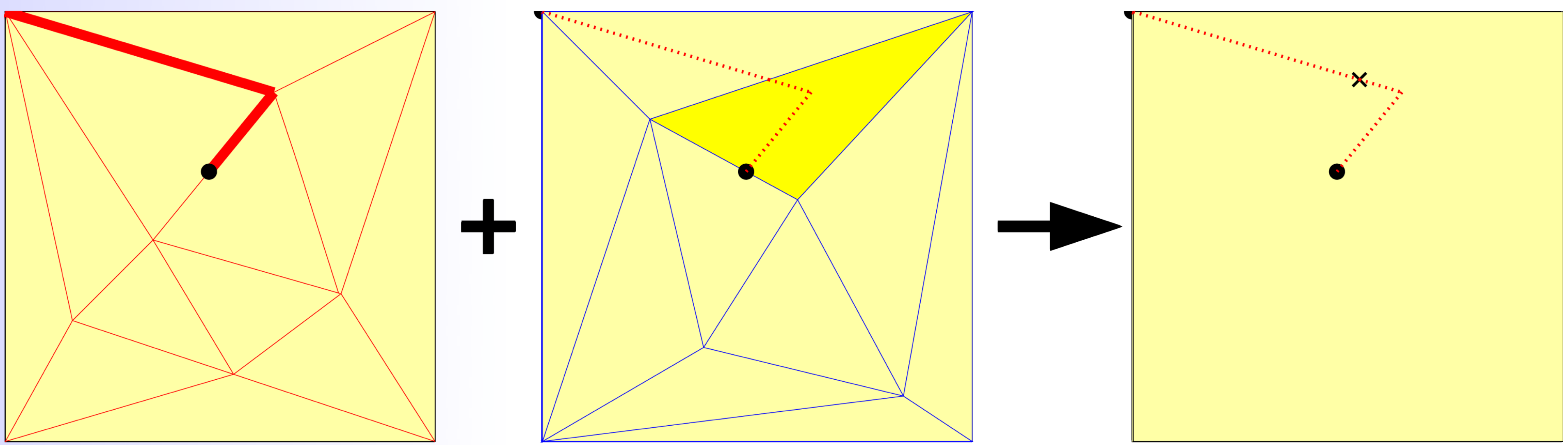
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

Overlaying Triangulations CPU-Efficiently

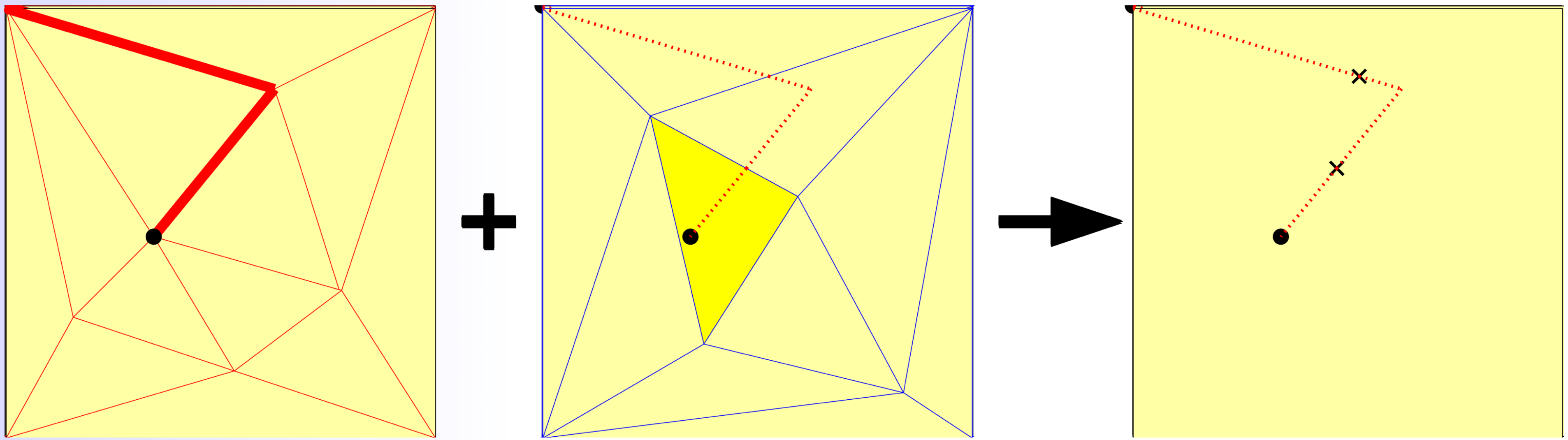
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

Overlaying Triangulations CPU-Efficiently

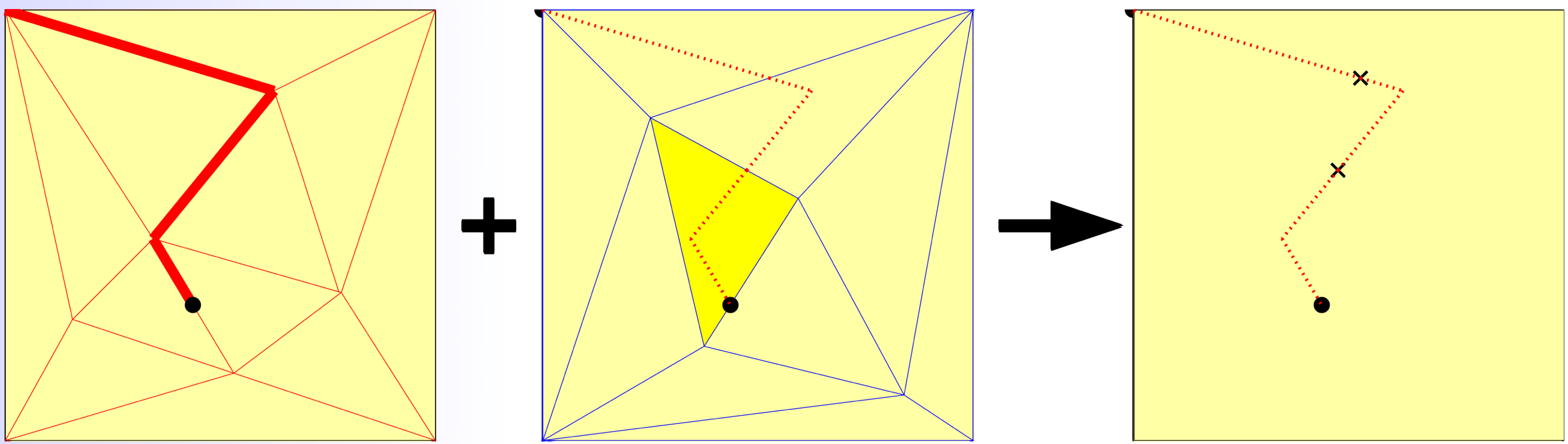
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

Overlaying Triangulations CPU-Efficiently

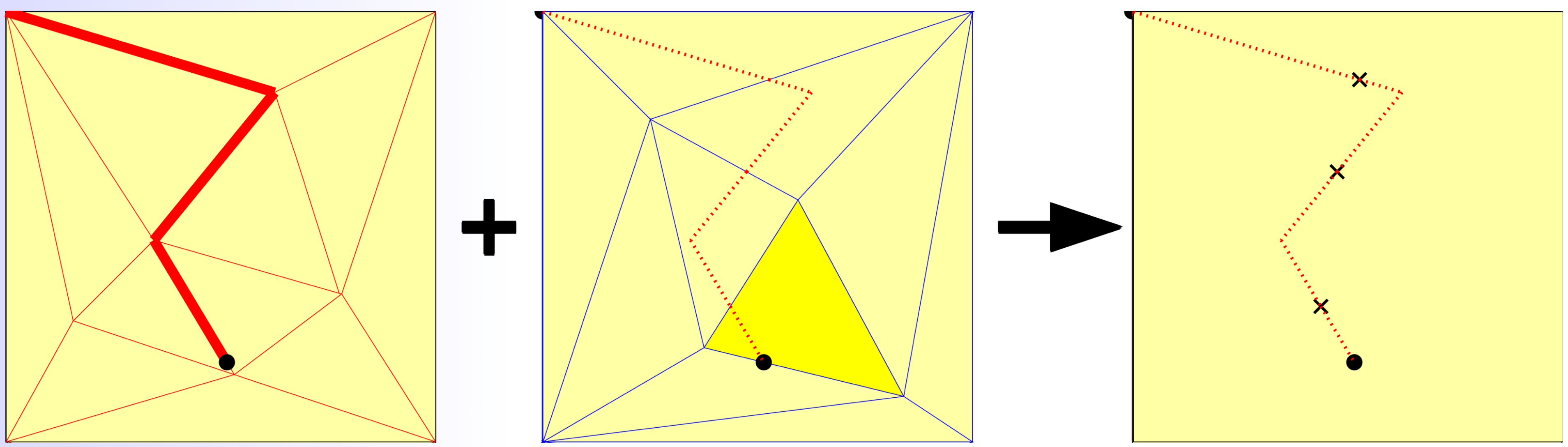
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

Overlaying Triangulations CPU-Efficiently

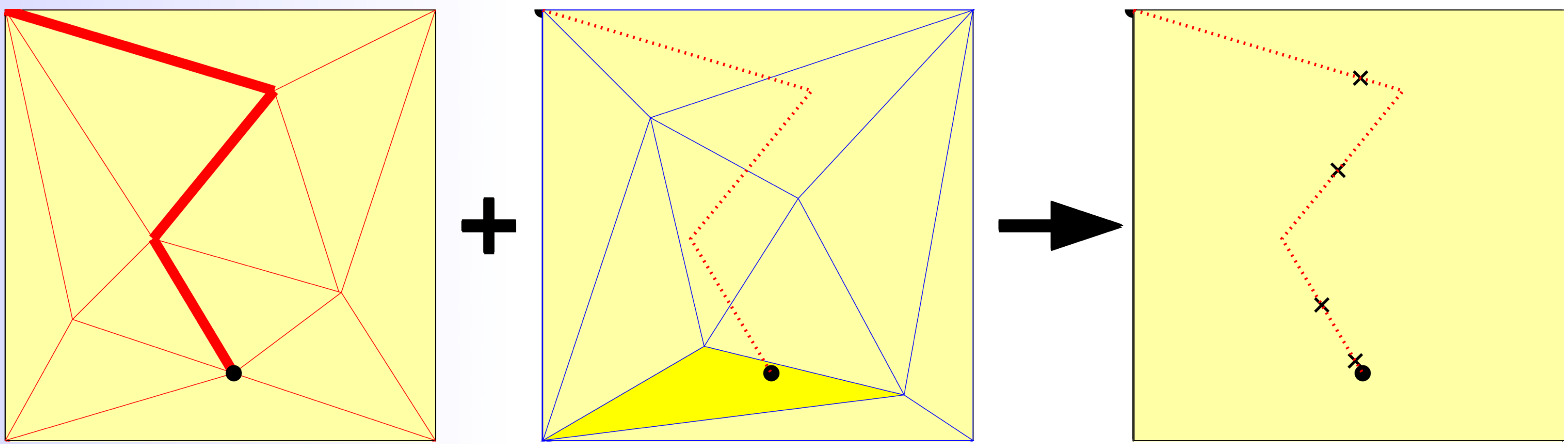
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

Overlaying Triangulations CPU-Efficiently

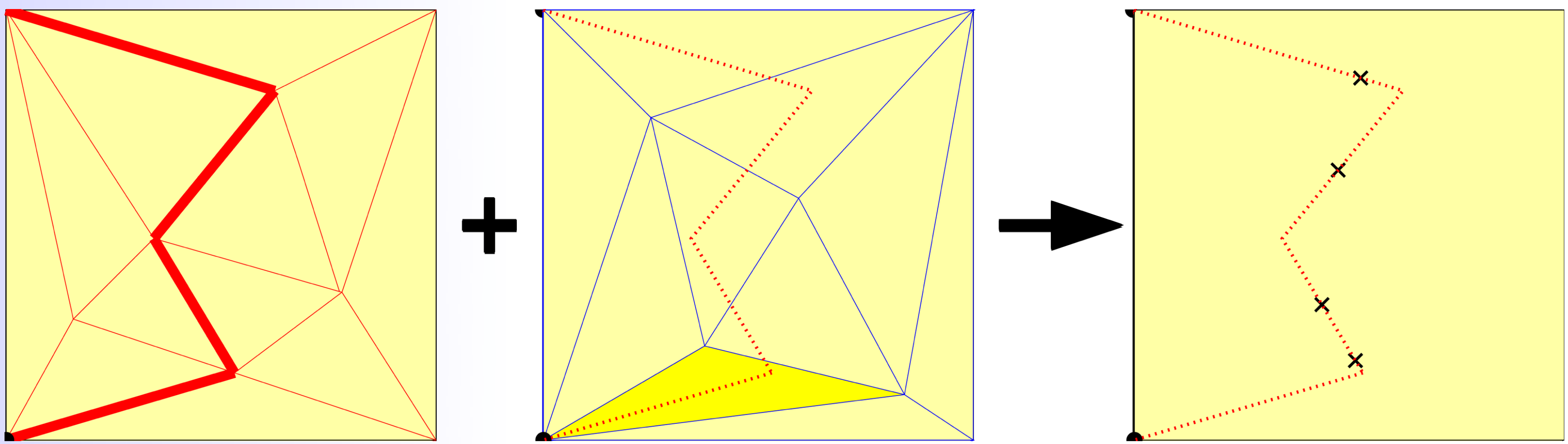
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

Overlaying Triangulations CPU-Efficiently

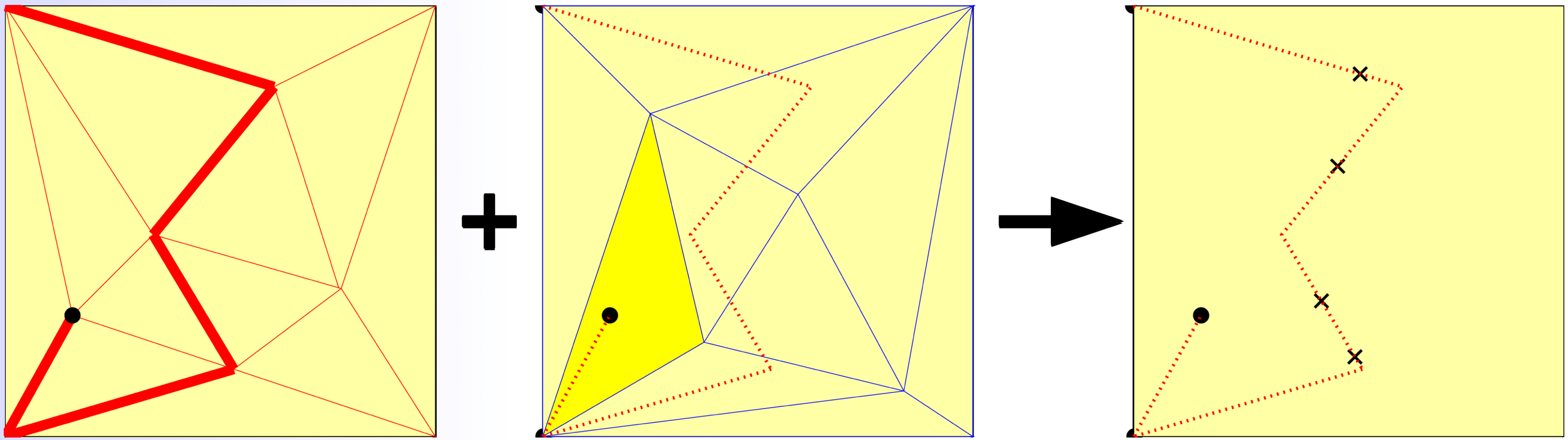
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

Overlaying Triangulations CPU-Efficiently

- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other

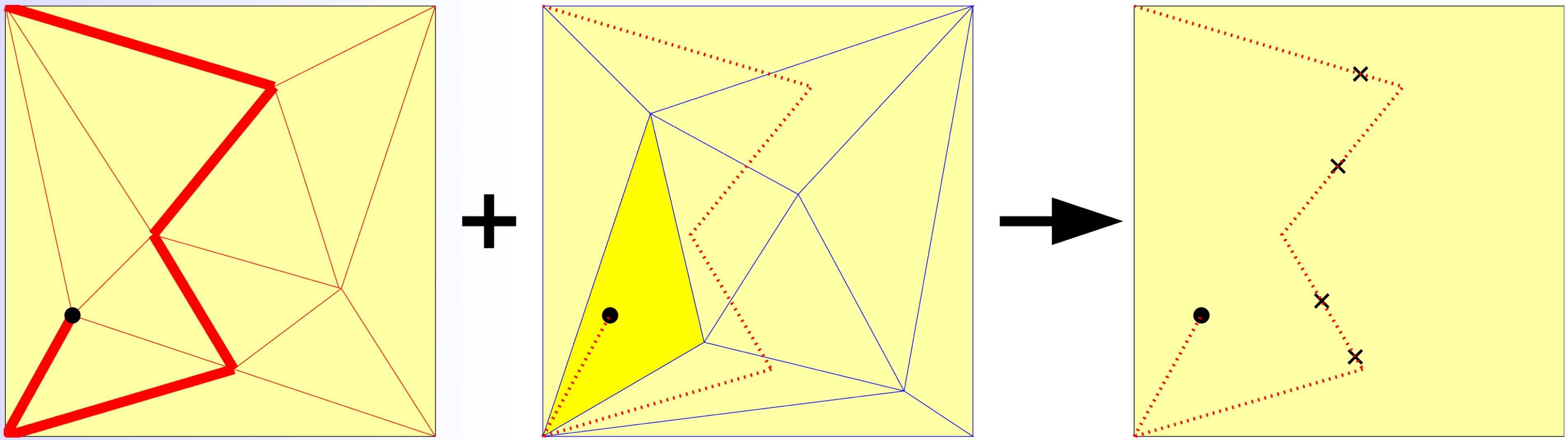
Overlaying Triangulations CPU-Efficiently

- Maps: ..., triangulations

- DFS in one triangulation, traverse triangles in the other
  - O(1) operations per edge
  - O(1) operations per crossing
- Total: O(n+k) CPU-operations (for n triangles, k crossings)

Overlaying Triangulations CPU-Efficiently

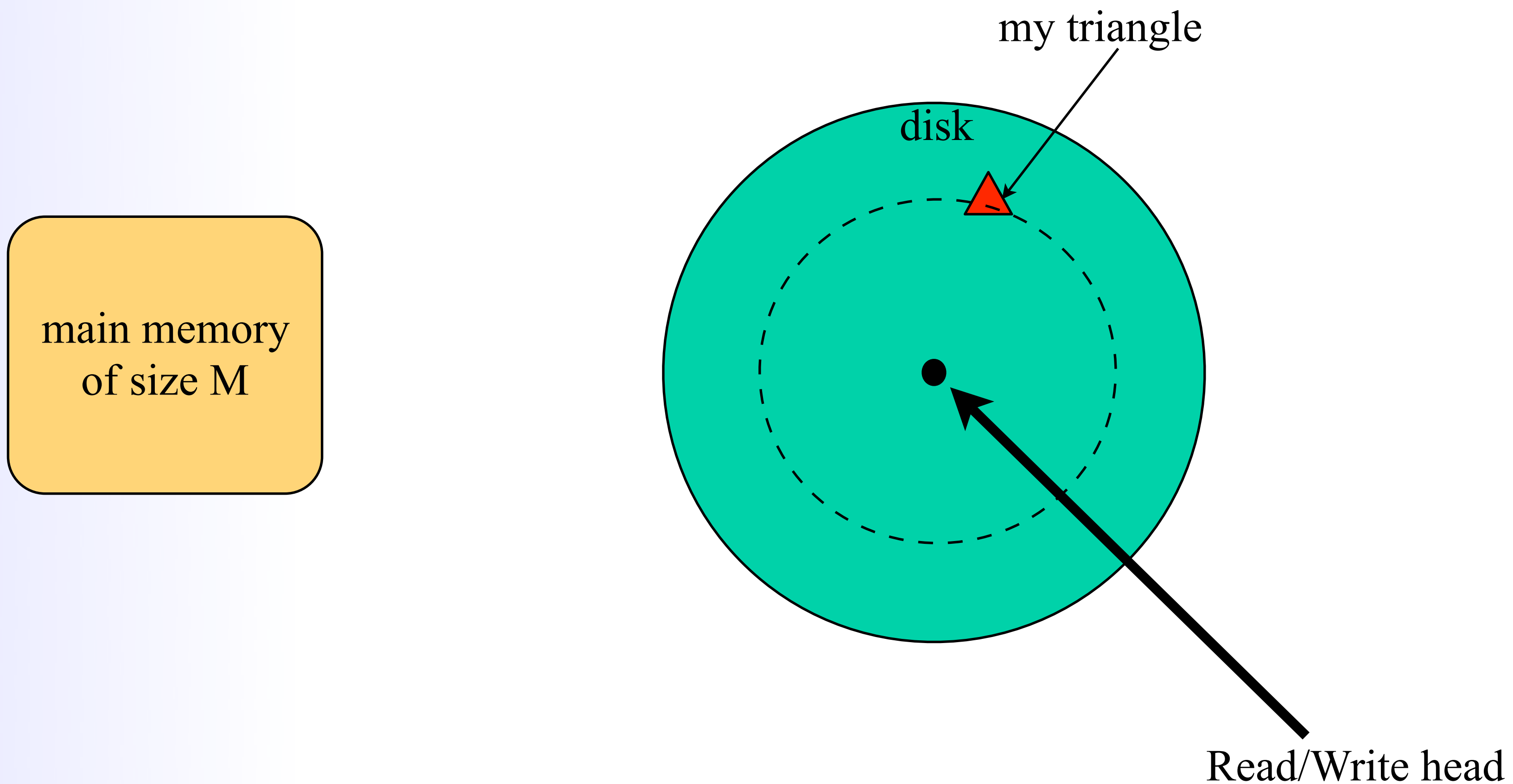
- Maps: ..., triangulations



- DFS in one triangulation, traverse triangles in the other
  - O(1) operations per edge
  - O(1) operations per crossing
- Total: O(n+k) CPU-operations (for n triangles, k crossings)

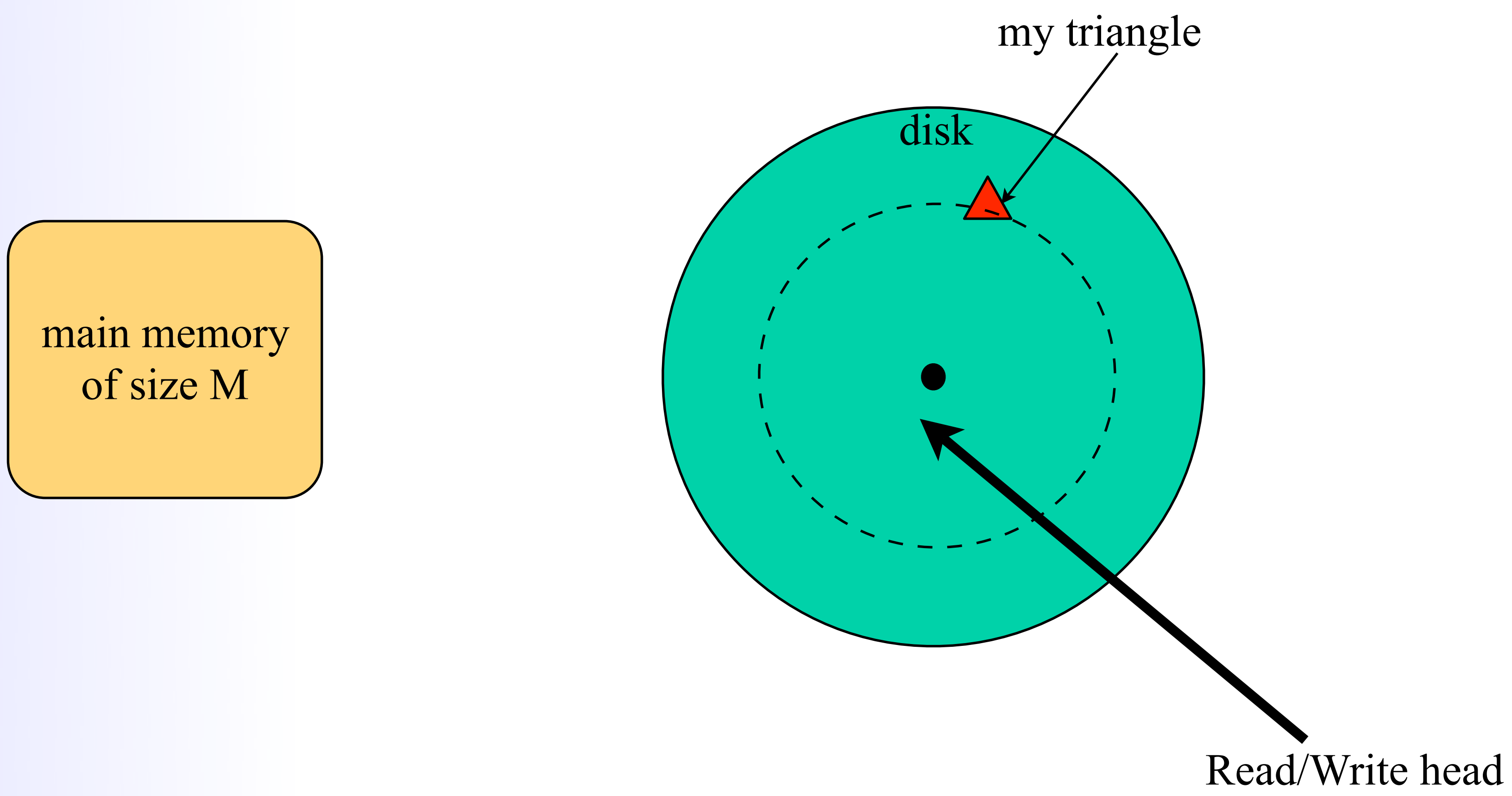
In External Memory

- If main memory is too small to hold all data



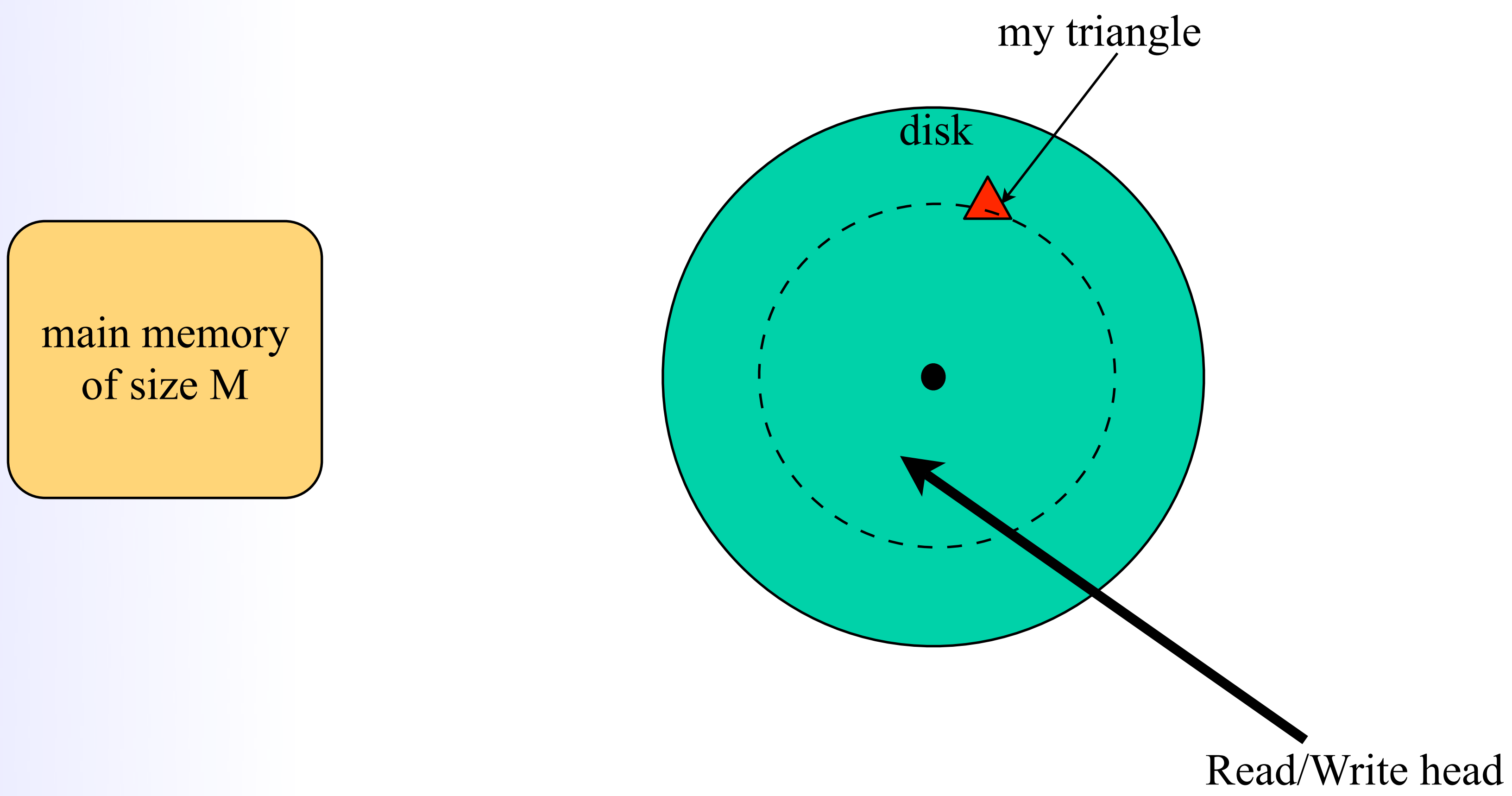
In External Memory

- If main memory is too small to hold all data



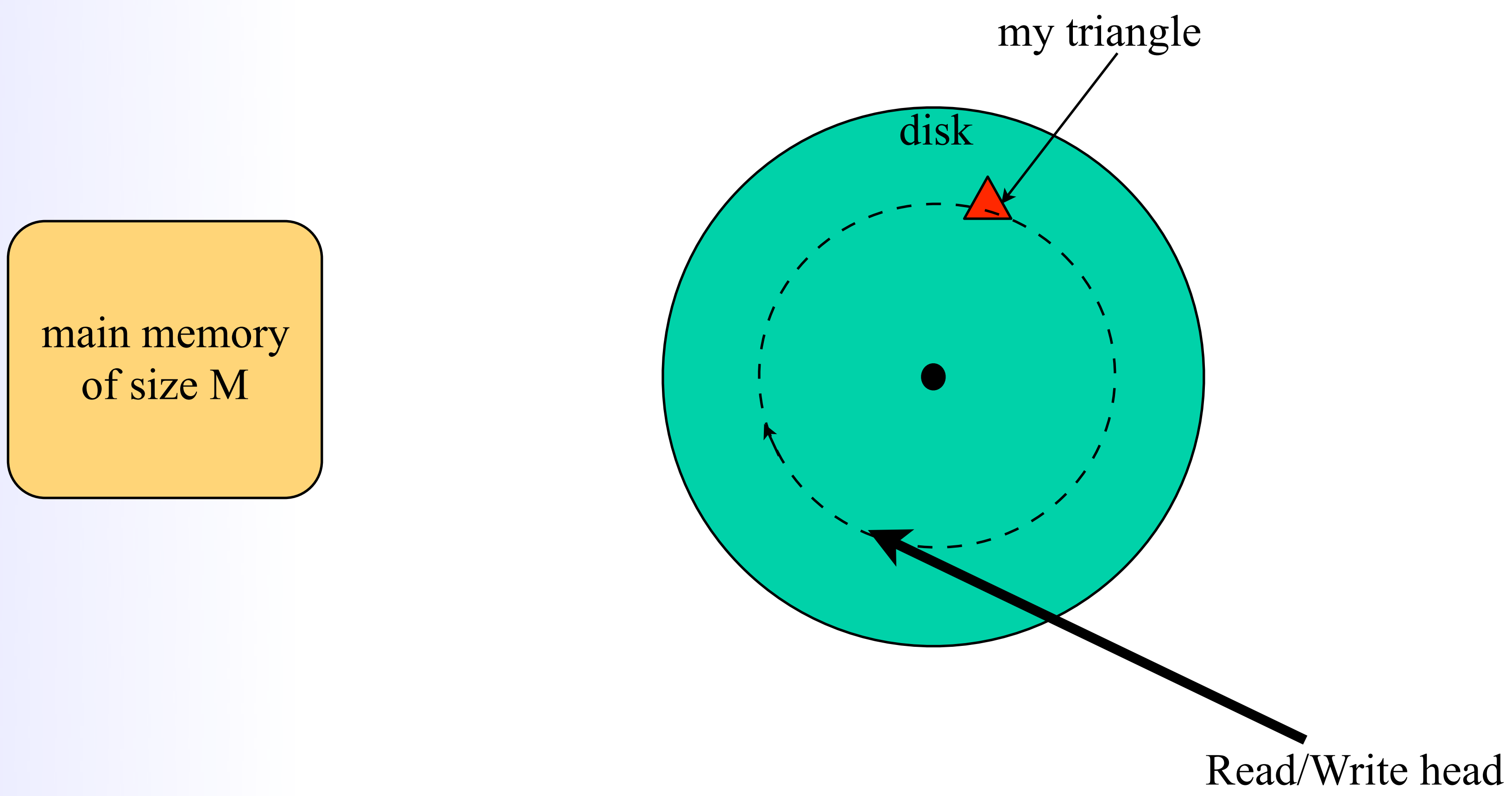
In External Memory

- If main memory is too small to hold all data



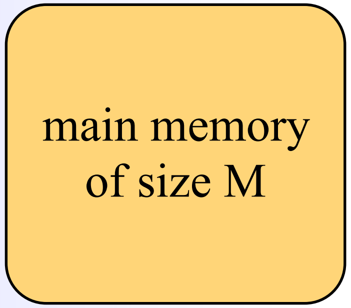
In External Memory

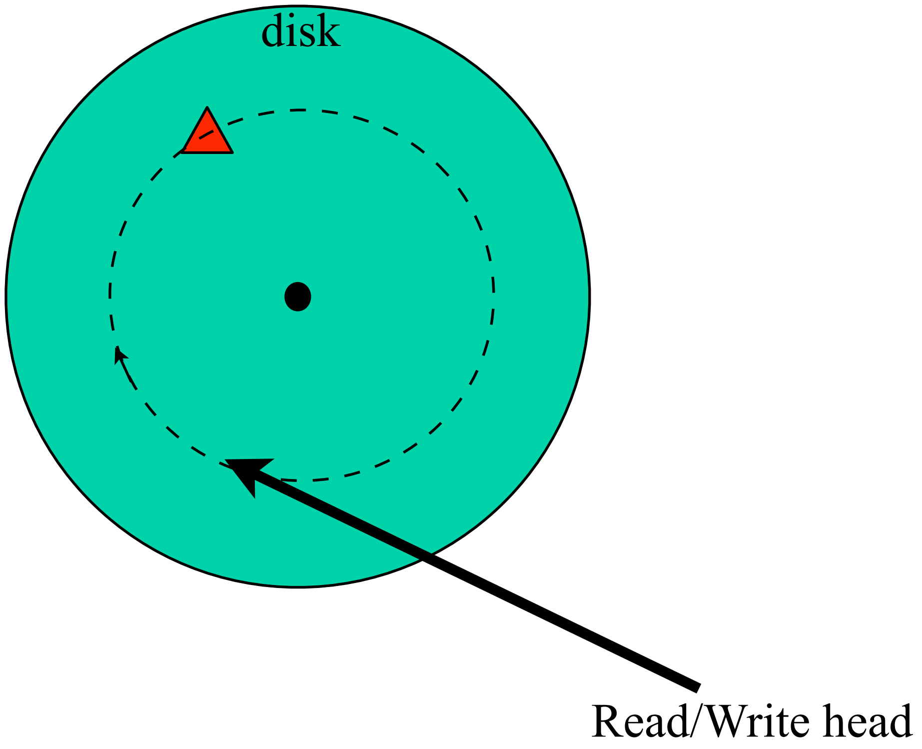
- If main memory is too small to hold all data



In External Memory

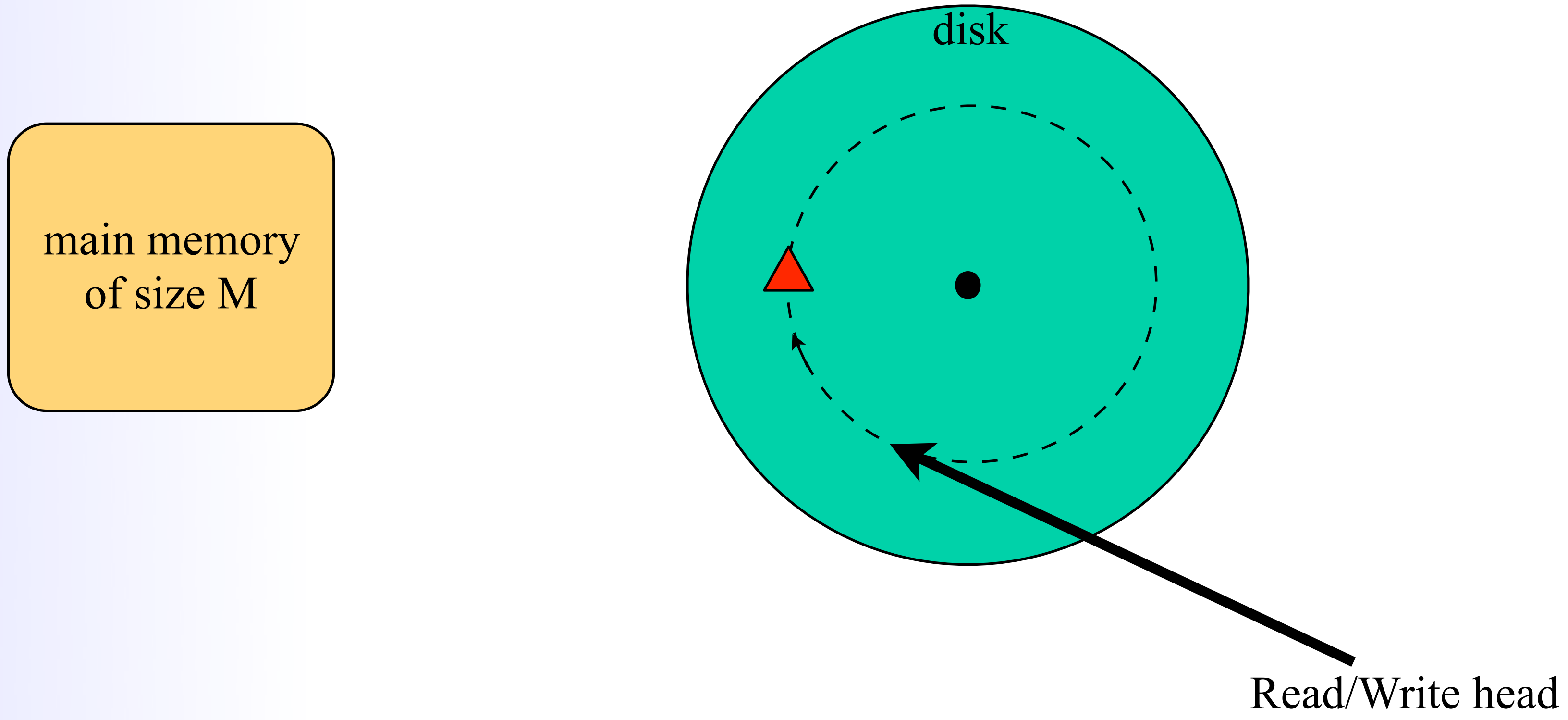
- If main memory is too small to hold all data





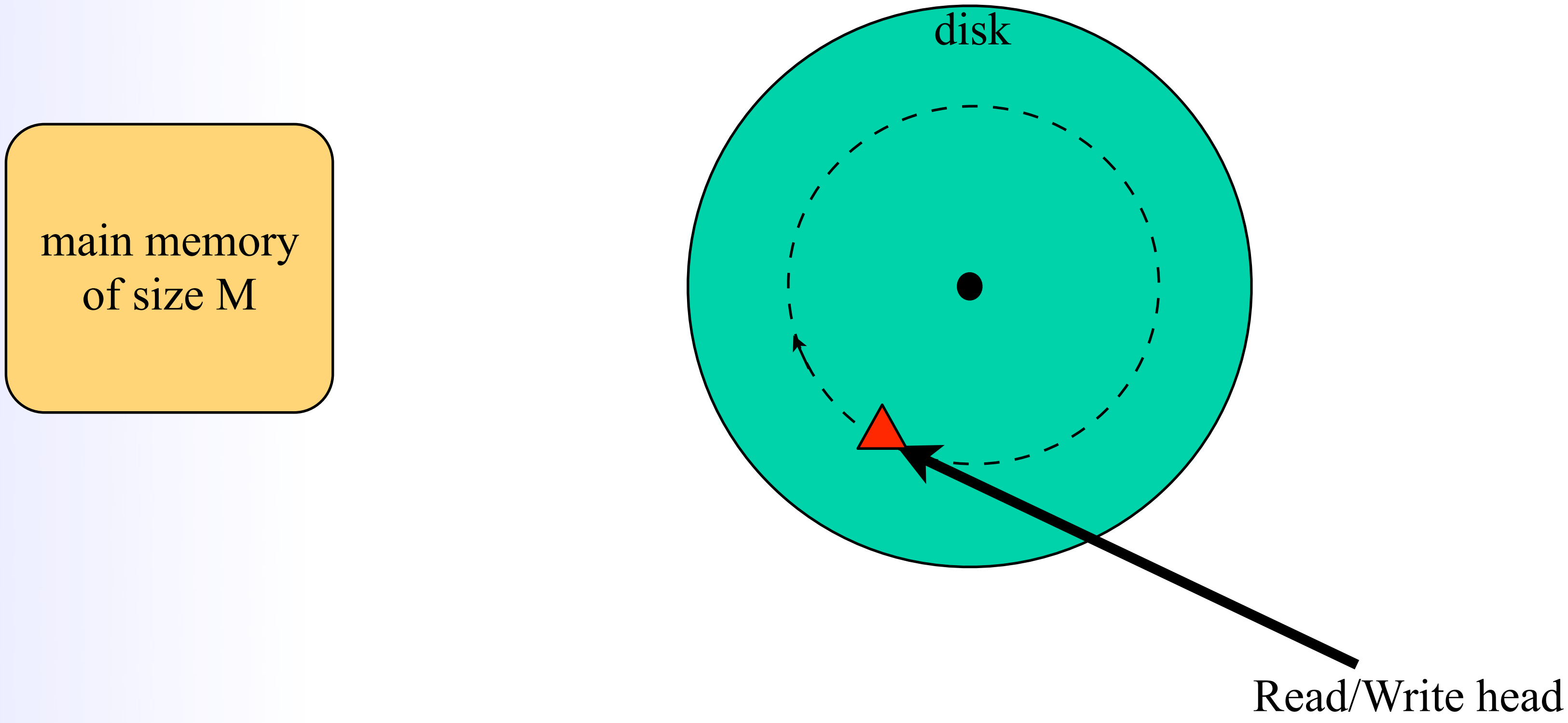
In External Memory

- If main memory is too small to hold all data



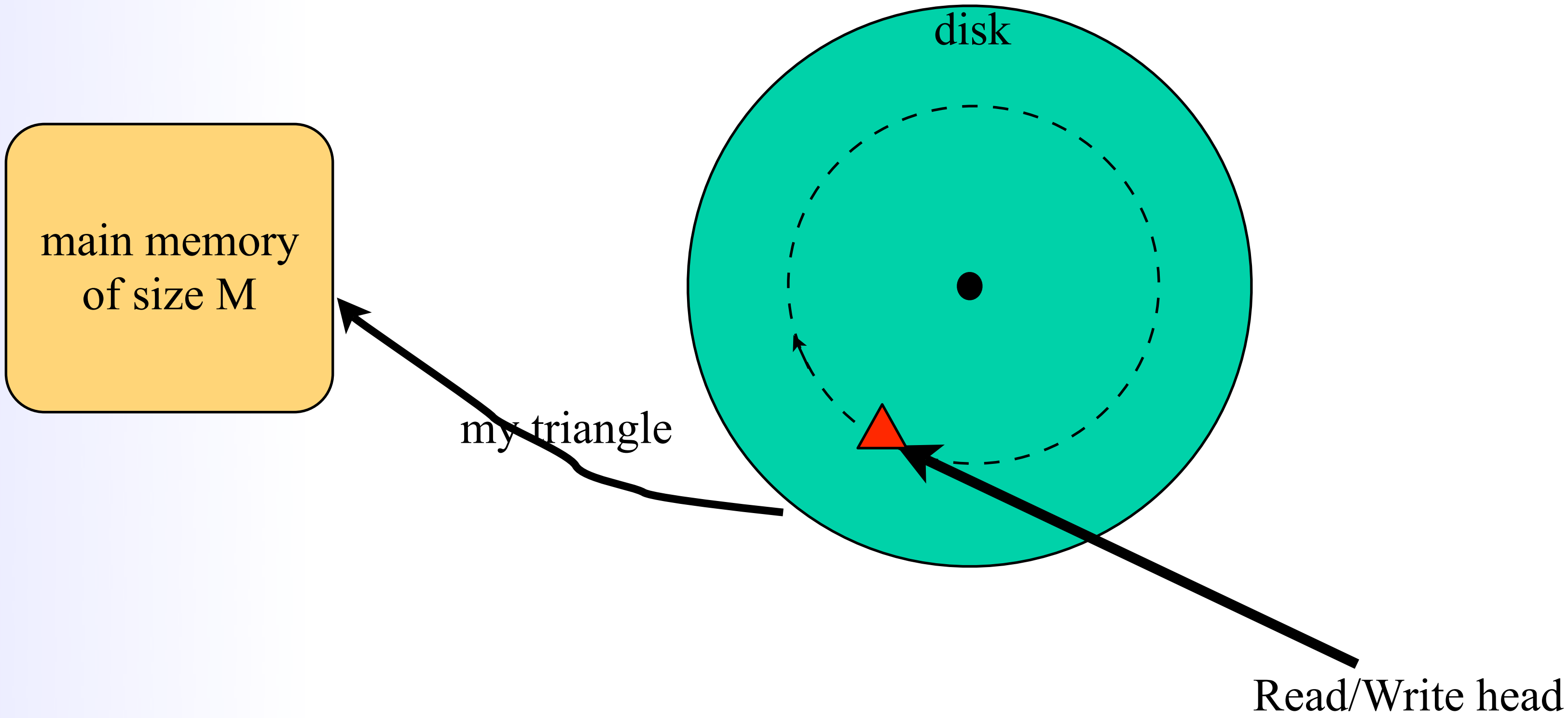
In External Memory

- If main memory is too small to hold all data



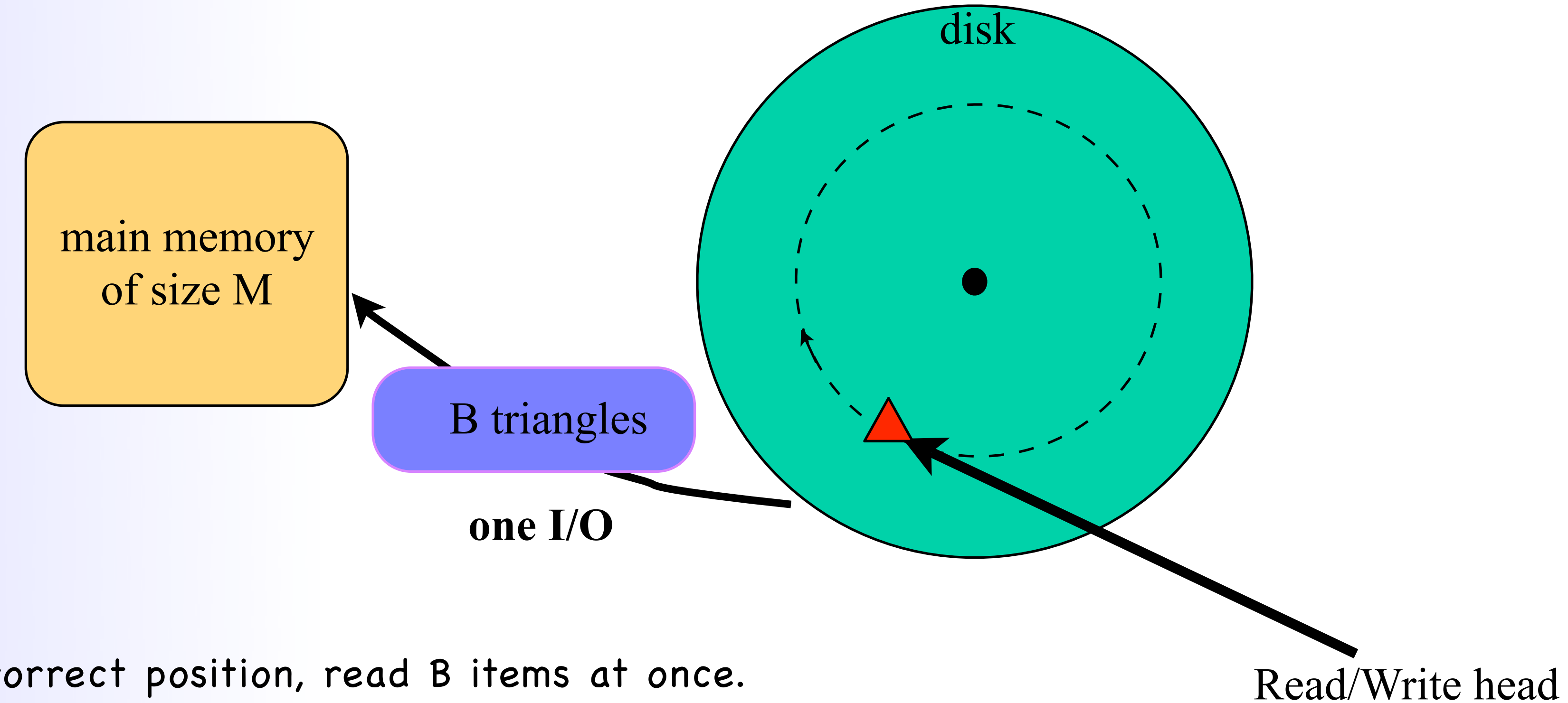
In External Memory

- If main memory is too small to hold all data



In External Memory

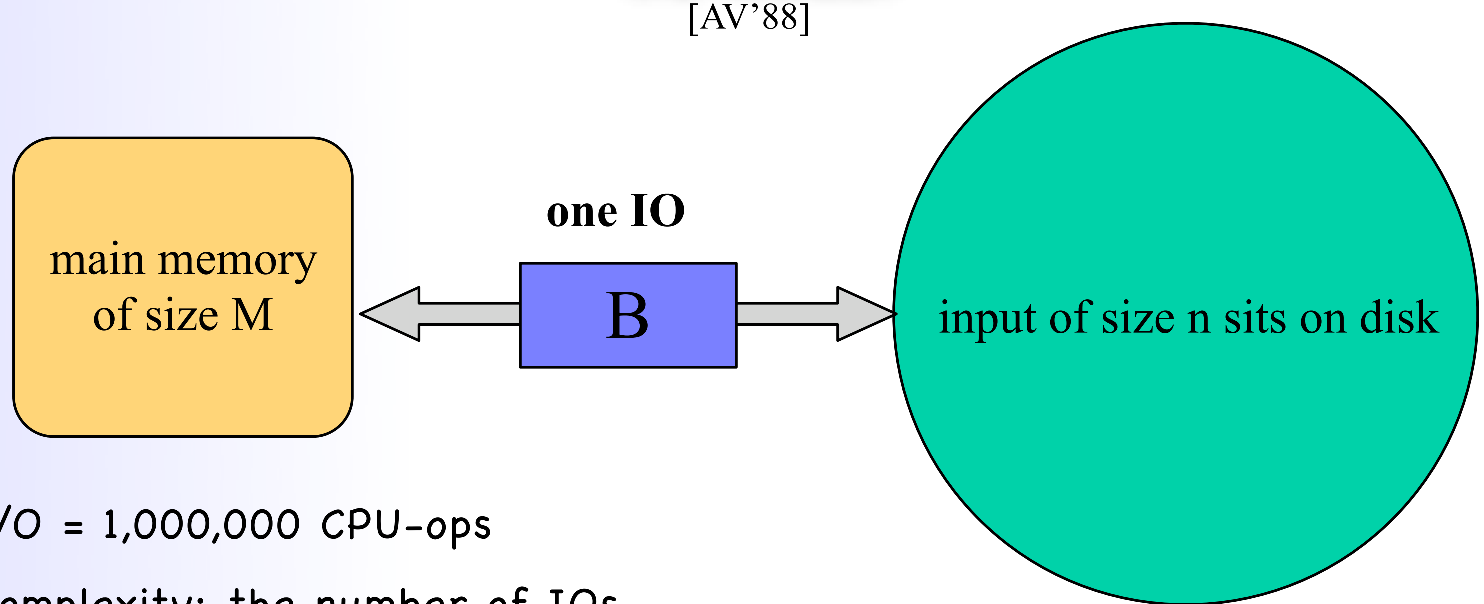
- If main memory is too small to hold all data



- Once in correct position, read B items at once.
  (hope you can keep them in memory until you need them)
- When working with large data, I/Os dominate.

I/O-Model

[AV'88]



- one I/O = 1,000,000 CPU-ops
- I/O-complexity: the number of IOs
- Goal: minimize I/O-complexity

- Basic building blocks and bounds:
  - scanning : $\mathtt{scan}(n) = \frac{n}{B}$ IOs
  - sorting: $\mathtt{sort}(n) = \Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$ IOs [AV'88]

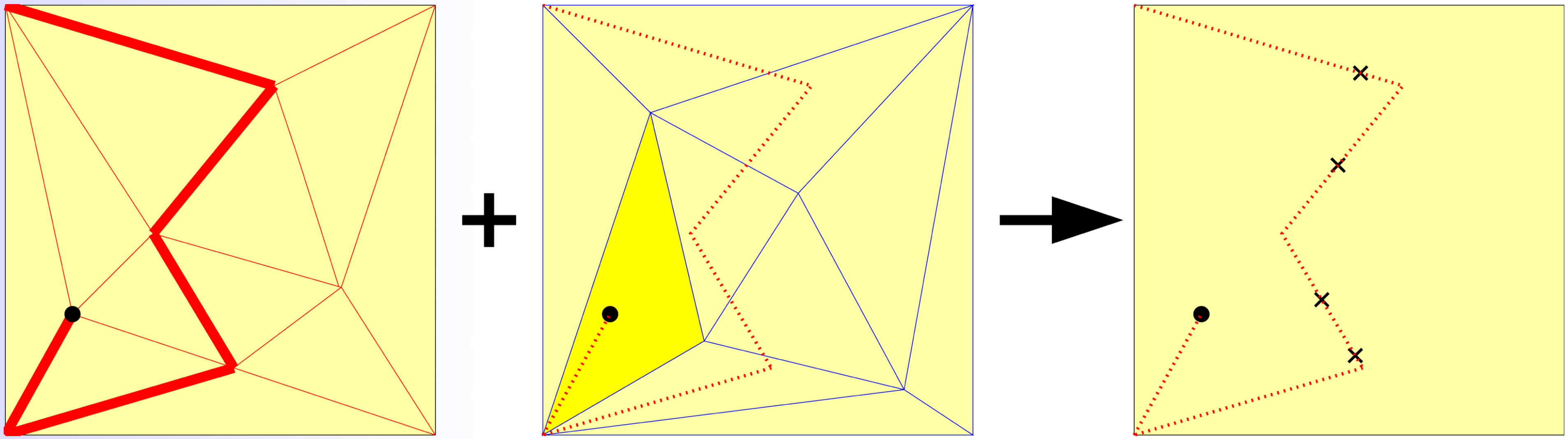
$$\mathtt{scan}(n) < \mathtt{sort}(n) \ll n \text{ IOs}$$

Overlaying Triangulations I/O-Efficiently?

- Imagine data is on disk.

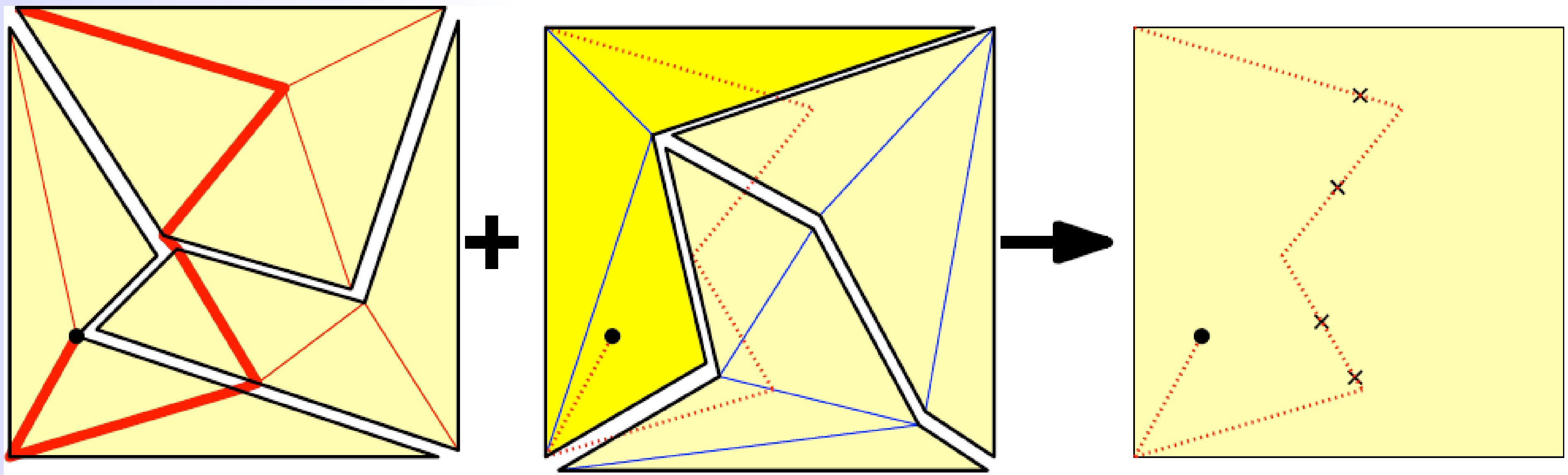
Overlaying Triangulations I/O-Efficiently?

- Imagine data is on disk.



Overlaying Triangulations I/O-Efficiently?

- On disk data is arranged in blocks.



- DFS in one triangulation, traverse triangles in the other:

  $\Theta(n+k)$ CPU-ops (for n triangles, k intersections)

  - O(1) IOs per edge
  - O(1) IOs per triangle
  - Total: $\Theta(n+k)$ IOs ← Not efficient

- $\mathtt{scan}(n) < \mathtt{sort}(n) \ll n$ IOs

Our results

n = input size;
M = main memory size;
B = disk block size

$$scan(n) = \frac{n}{B} \quad < \quad sort(n) = \frac{n}{B} \log_{M/B} \frac{n}{B} \quad << \quad n$$

Previously:

- Arge et al.: map overlay in $O(sort(n) + k/B)$ I/O's (complicated, super-linear space)
- Crauser et al.: randomized, linear space

Our results: in $O(sort(n))$ I/O's we can build a data structure that supports:

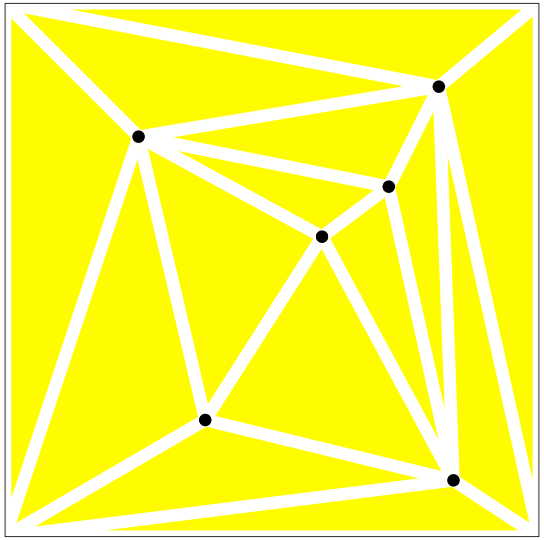
- map overlay in $O(scan(n))$ I/O's;
- point location in $O(\log_B n)$ I/O's;
- range queries in $O(\frac{1}{\varepsilon}(\log_B n) + scan(k_\varepsilon))$ I/O's;
- for triangulations: basic updates in $O(\log_B n)$ I/O's.

Condition: input must be *fat* triangulation (all angles $>$ positive constant), or a *low-density* set of segments (for any circle C, #intersecting segments $> \mathrm{diam}(C)$ is $O(1)$)

Ingredients: quadtrees ...

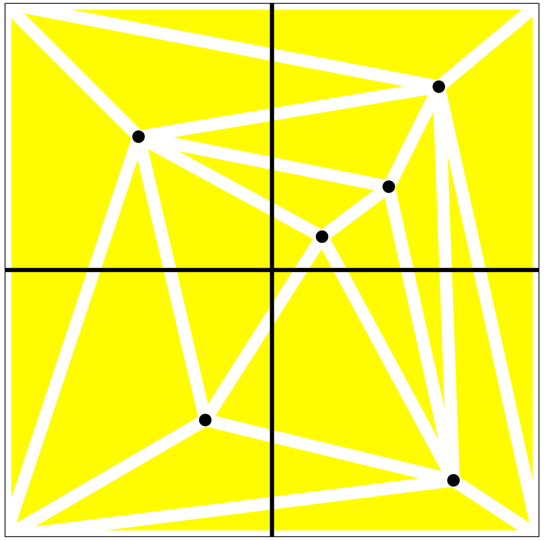
Quadtree: divide unit square into quadrants, refine until amount of data per cell is small.

(for example: until every cell has at most one vertex)



Quadtree: divide unit square into quadrants, refine until amount of data per cell is small.

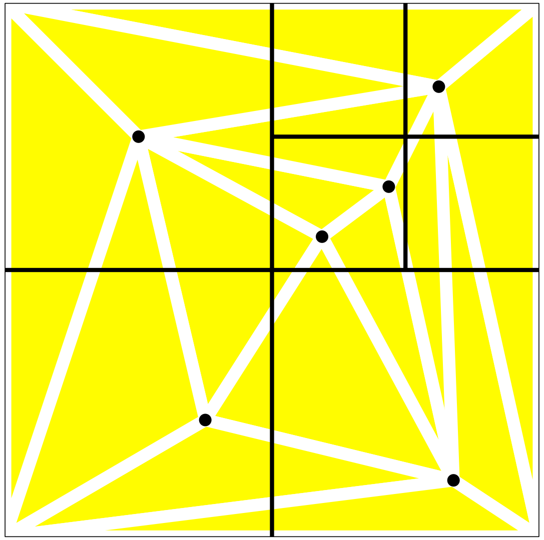
(for example: until every cell has at most one vertex)



Ingredients: quadtrees ...

Quadtree: divide unit square into quadrants, refine until amount of data per cell is small.

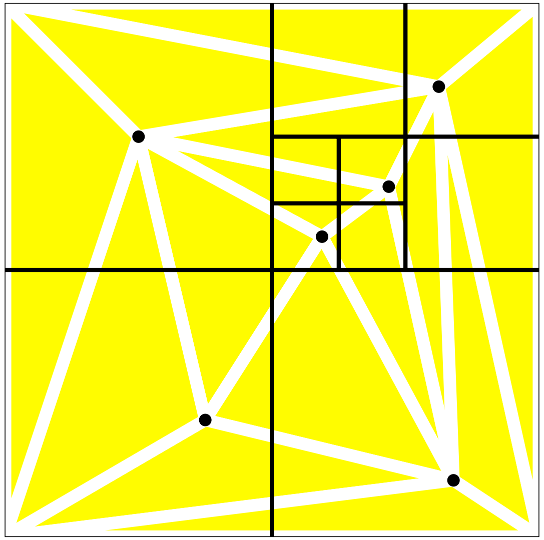
(for example: until every cell has at most one vertex)



Ingredients: quadtrees ...

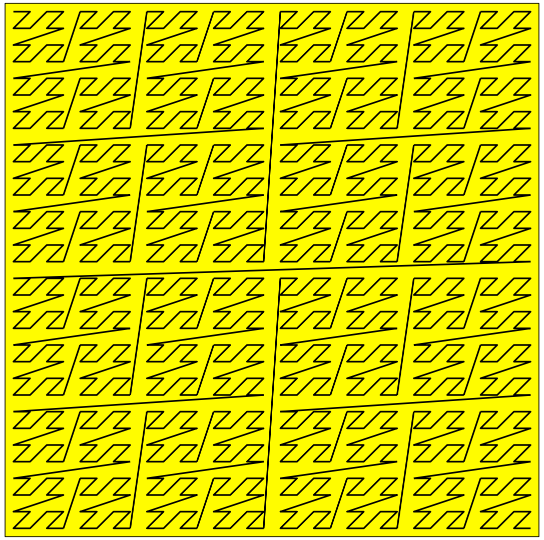
Quadtree: divide unit square into quadrants, refine until amount of data per cell is small.

(for example: until every cell has at most one vertex)



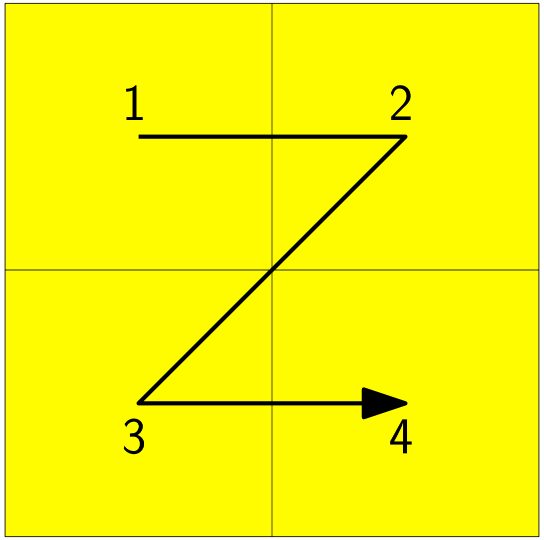
Ingredients: ... and Z-order

Z-order space-filling curve: visit quadrants recursively in order NW, NE, SW, SE



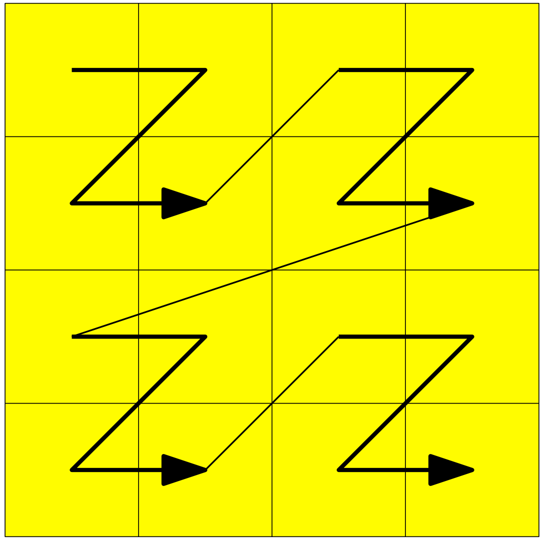
Ingredients: ... and Z-order

Z-order space-filling curve: visit quadrants recursively in order NW, NE, SW, SE



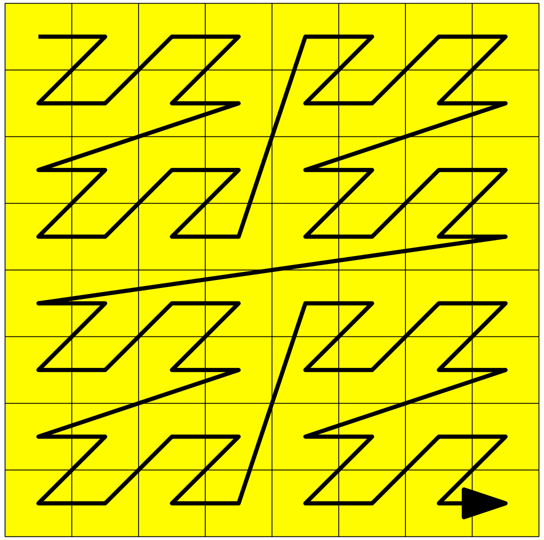
Ingredients: ... and Z-order

Z-order space-filling curve: visit quadrants recursively in order NW, NE, SW, SE



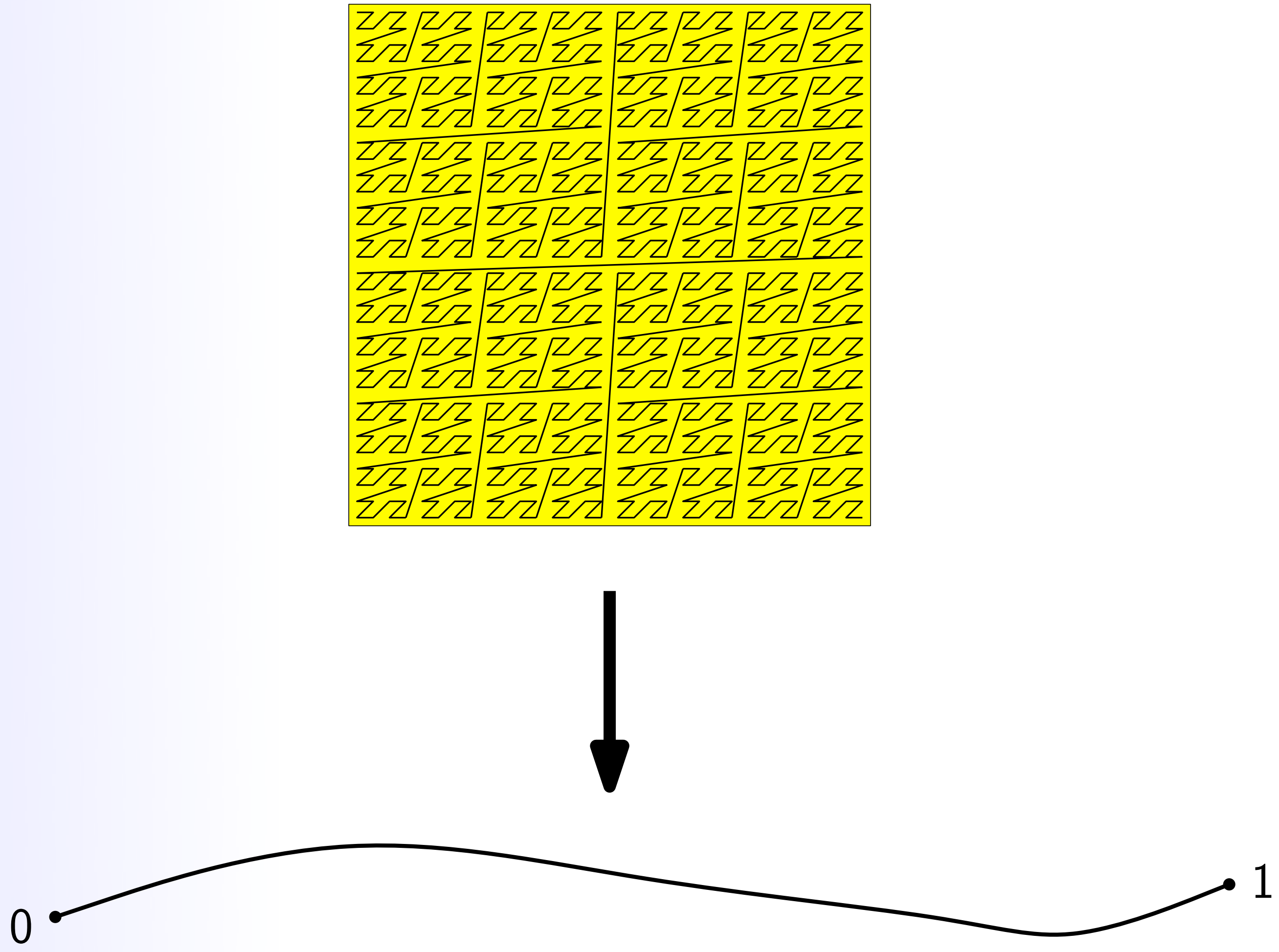
Ingredients: ... and Z-order

Z-order space-filling curve: visit quadrants recursively in order NW, NE, SW, SE



Ingredients: ... and Z-order

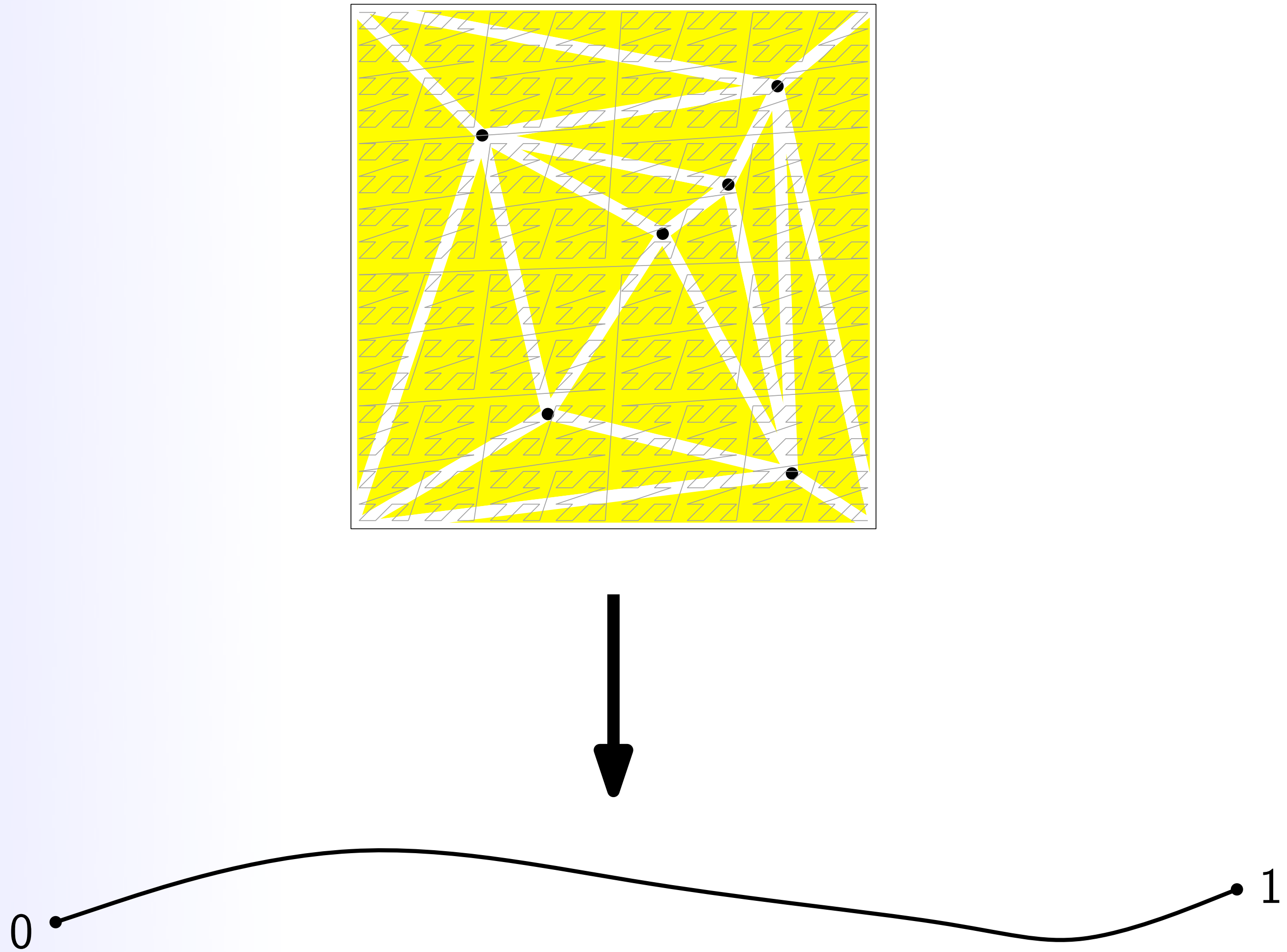
Z-order space-filling curve: visit quadrants recursively in order NW, NE, SW, SE



Ingredients: quadtrees and Z-order

Quadtree cell ≡ interval on Z-order curve

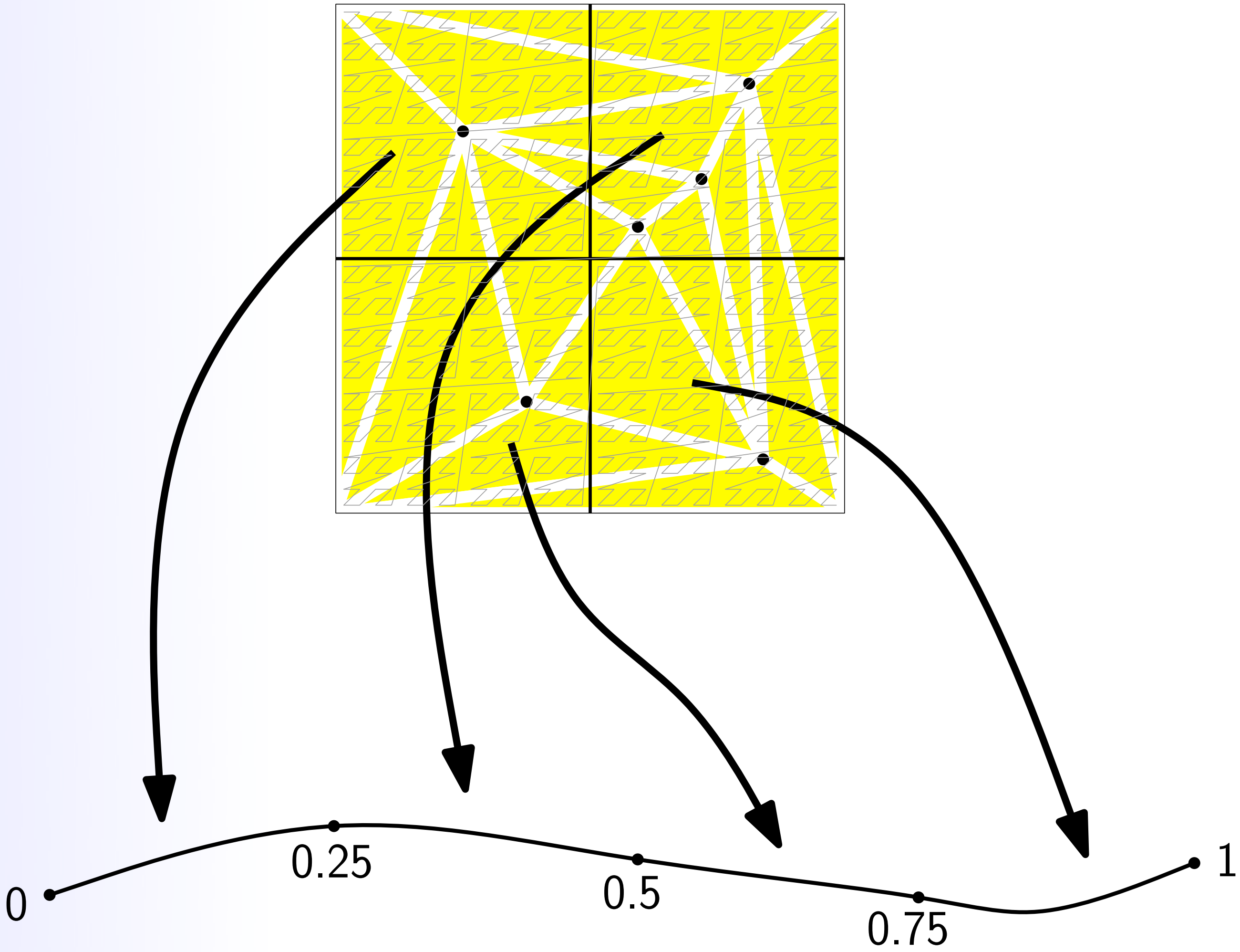
Quadtree subdivision ≡ subdivision of Z-order curve



Ingredients: quadtrees and Z-order

Quadtree cell $\equiv$ interval on Z-order curve

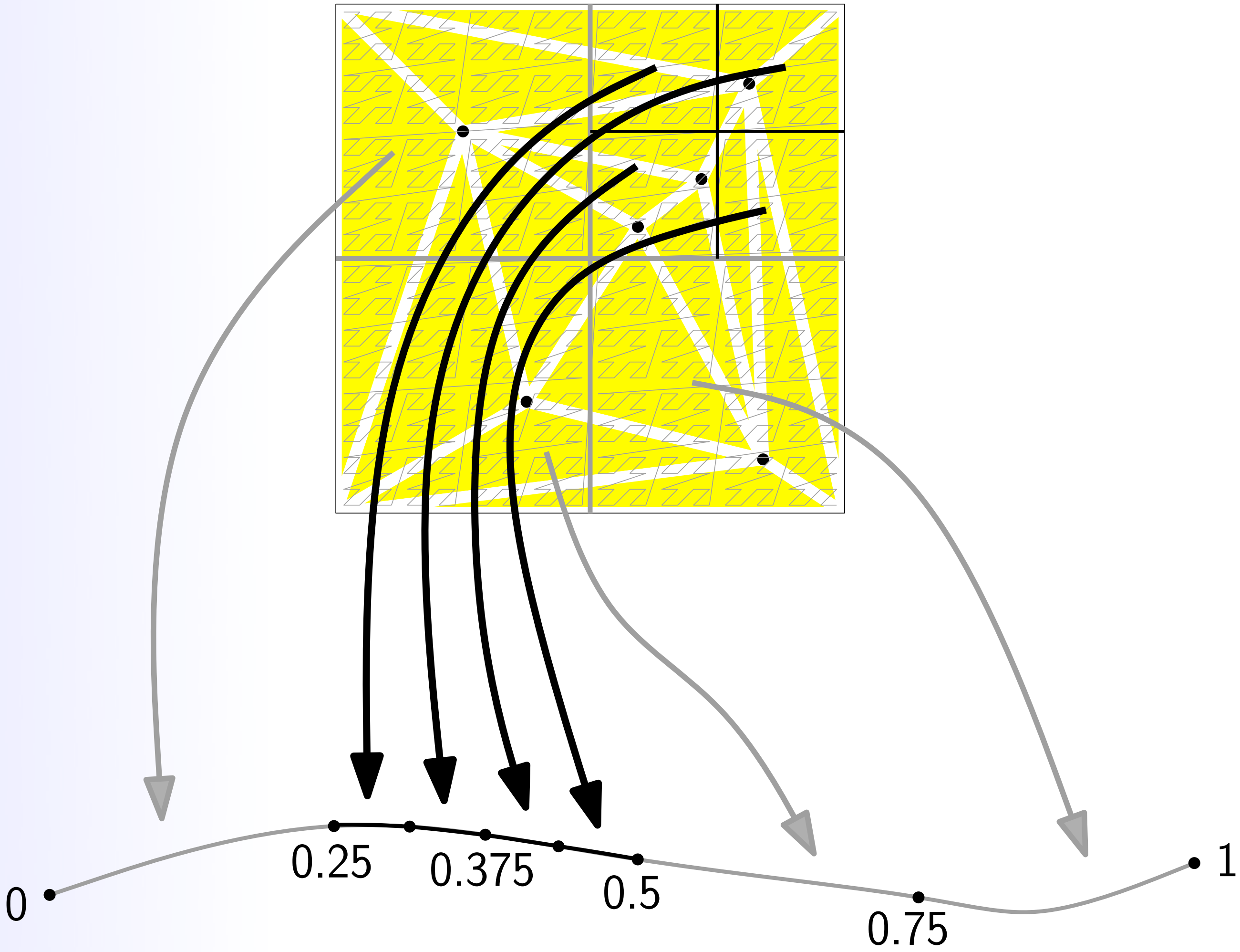
Quadtree subdivision $\equiv$ subdivision of Z-order curve



Ingredients: quadtrees and Z-order

Quadtree cell ≡ interval on Z-order curve

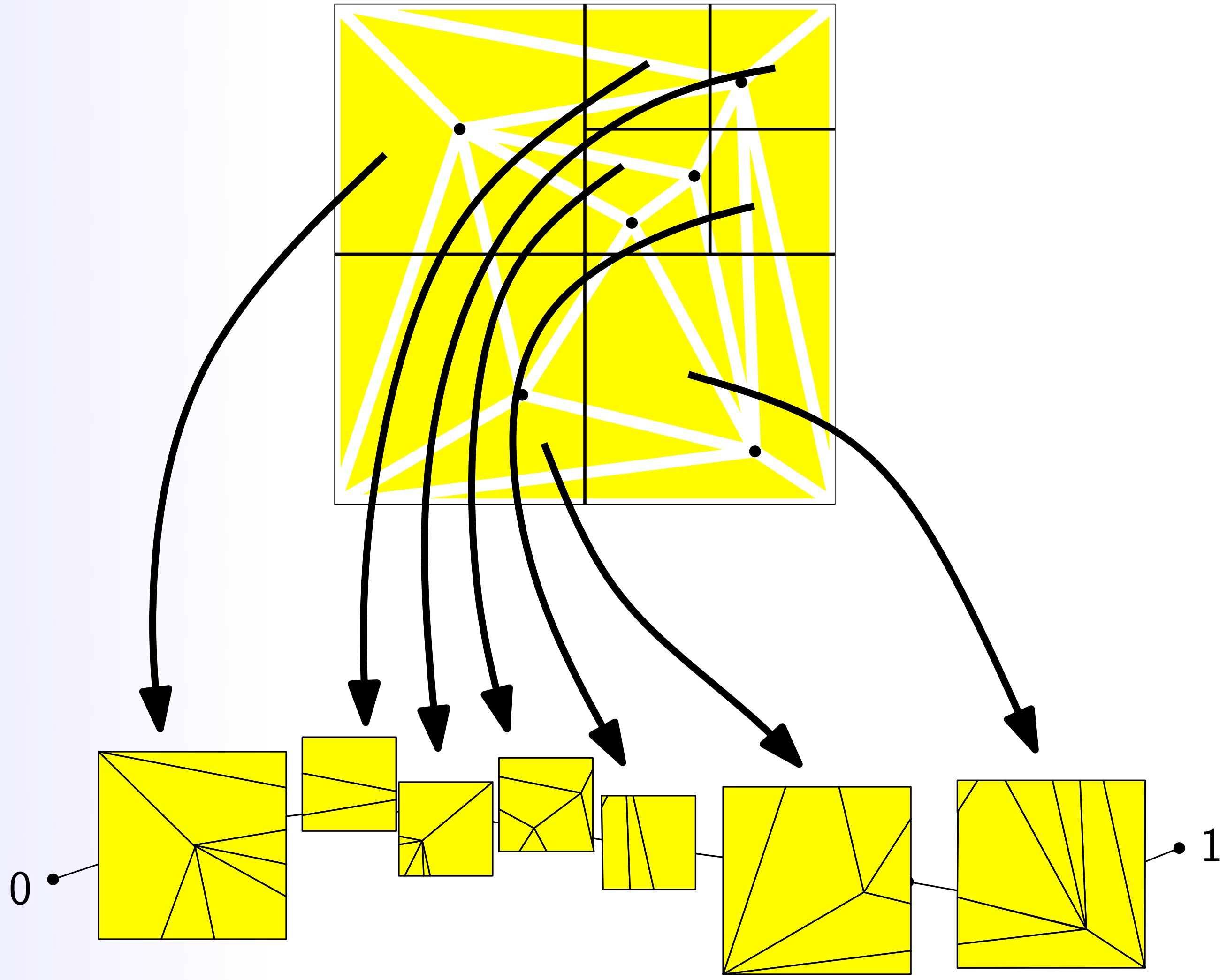
Quadtree subdivision ≡ subdivision of Z-order curve



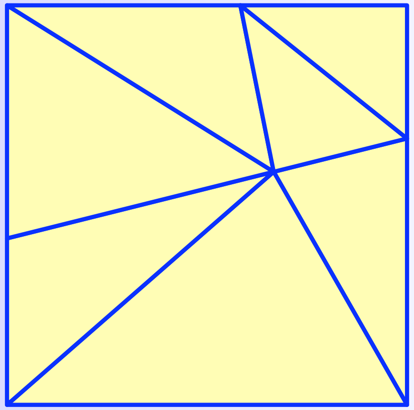
Ingredients: quadtrees and Z-order

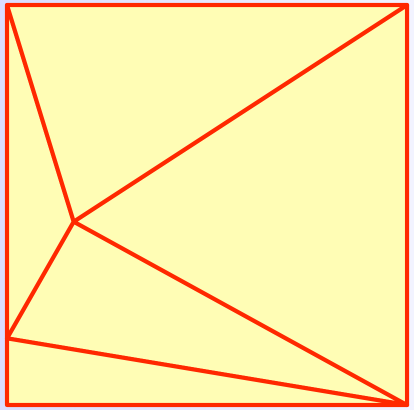
Quadtree cell ≡ interval on Z-order curve

Quadtree subdivision ≡ subdivision of Z-order curve

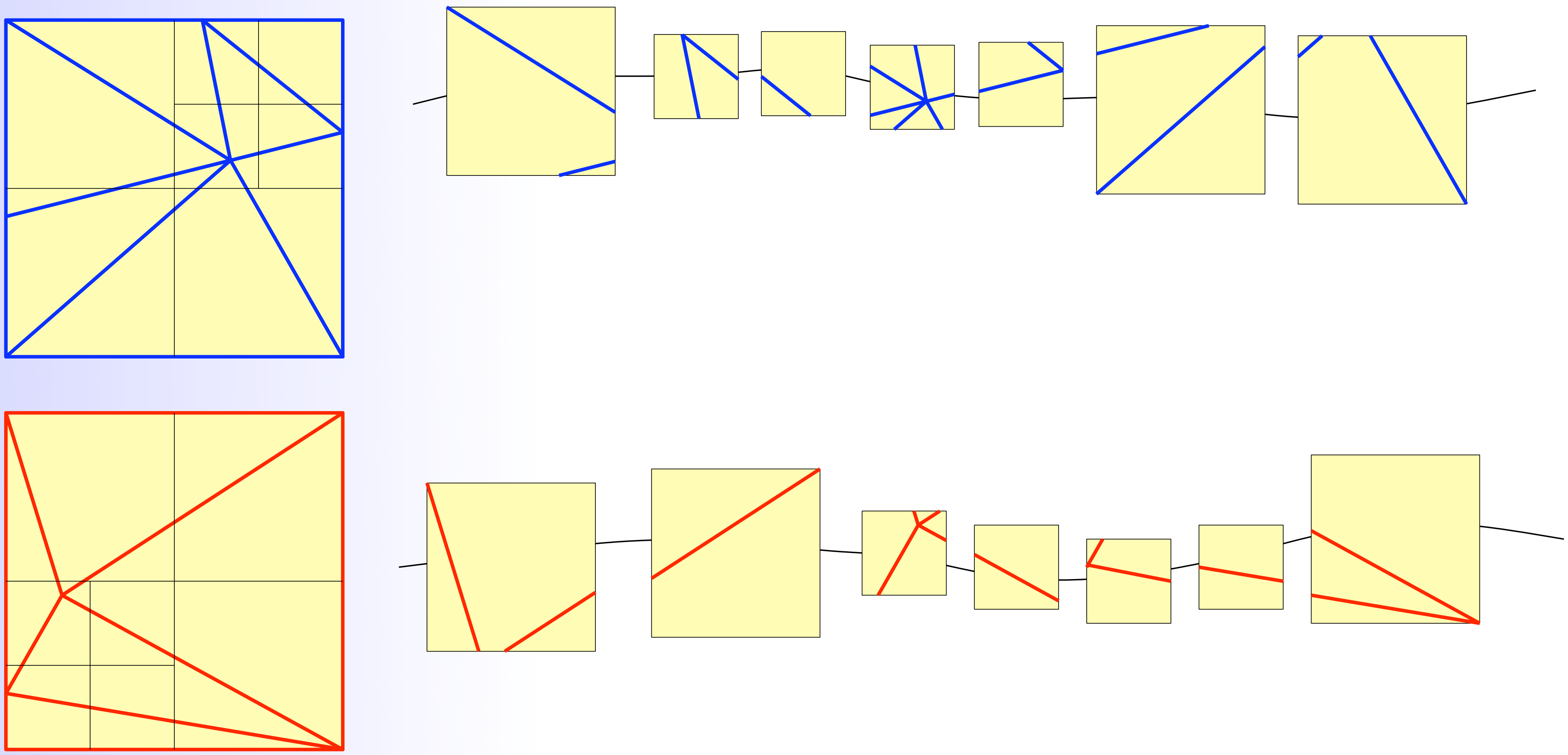


Map overlay with quadtrees in Z-order

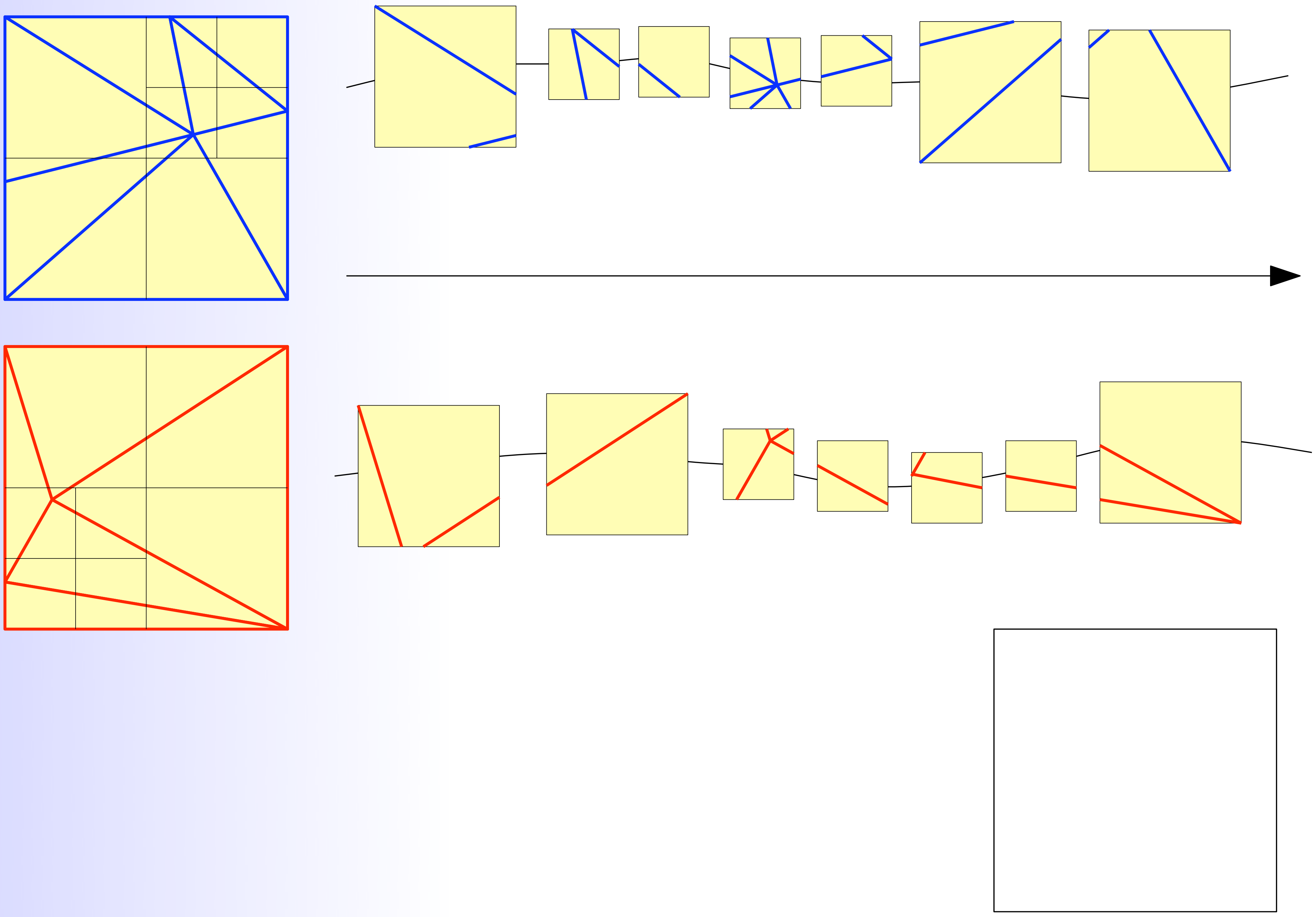




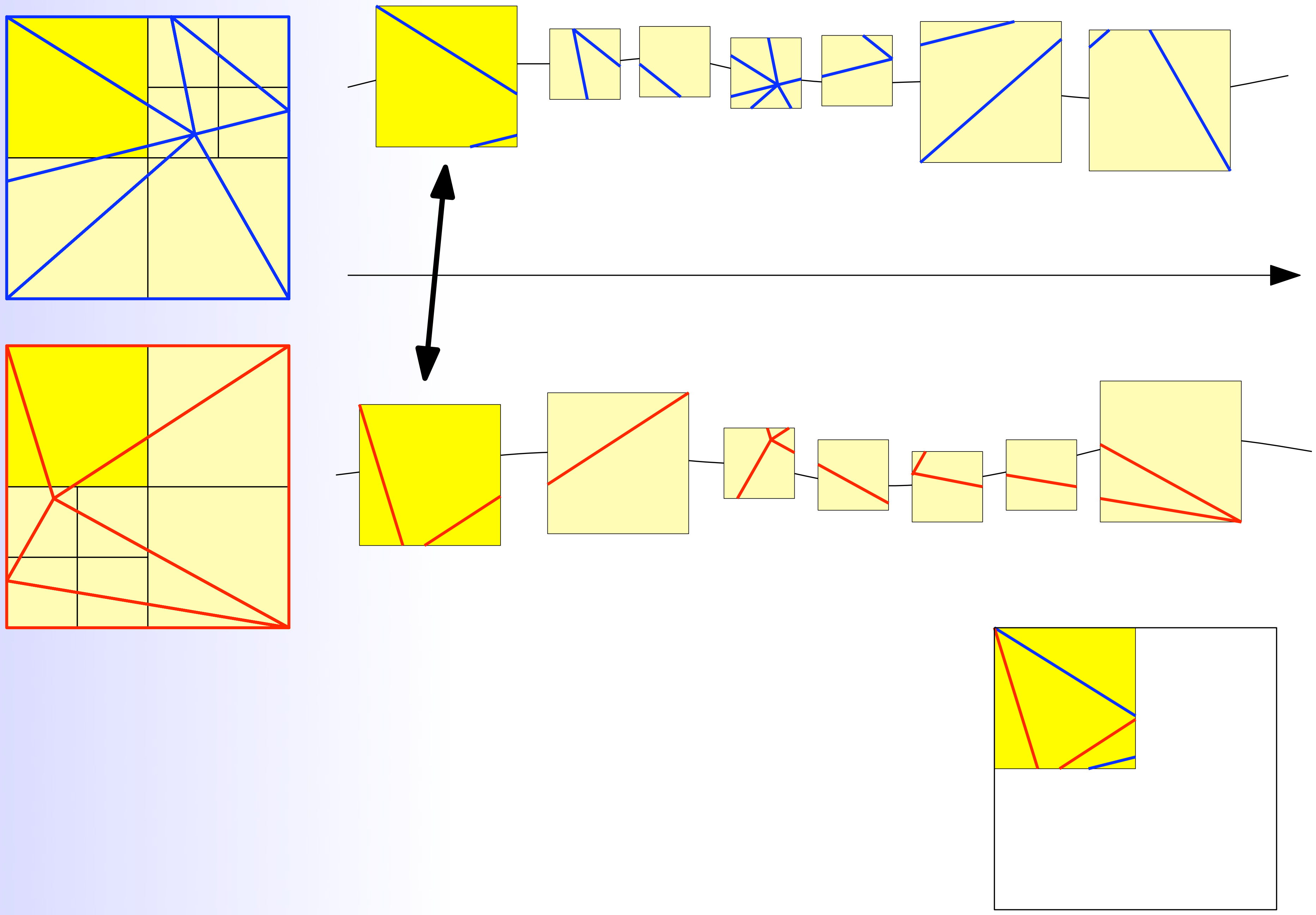
Map overlay with quadtrees in Z-order



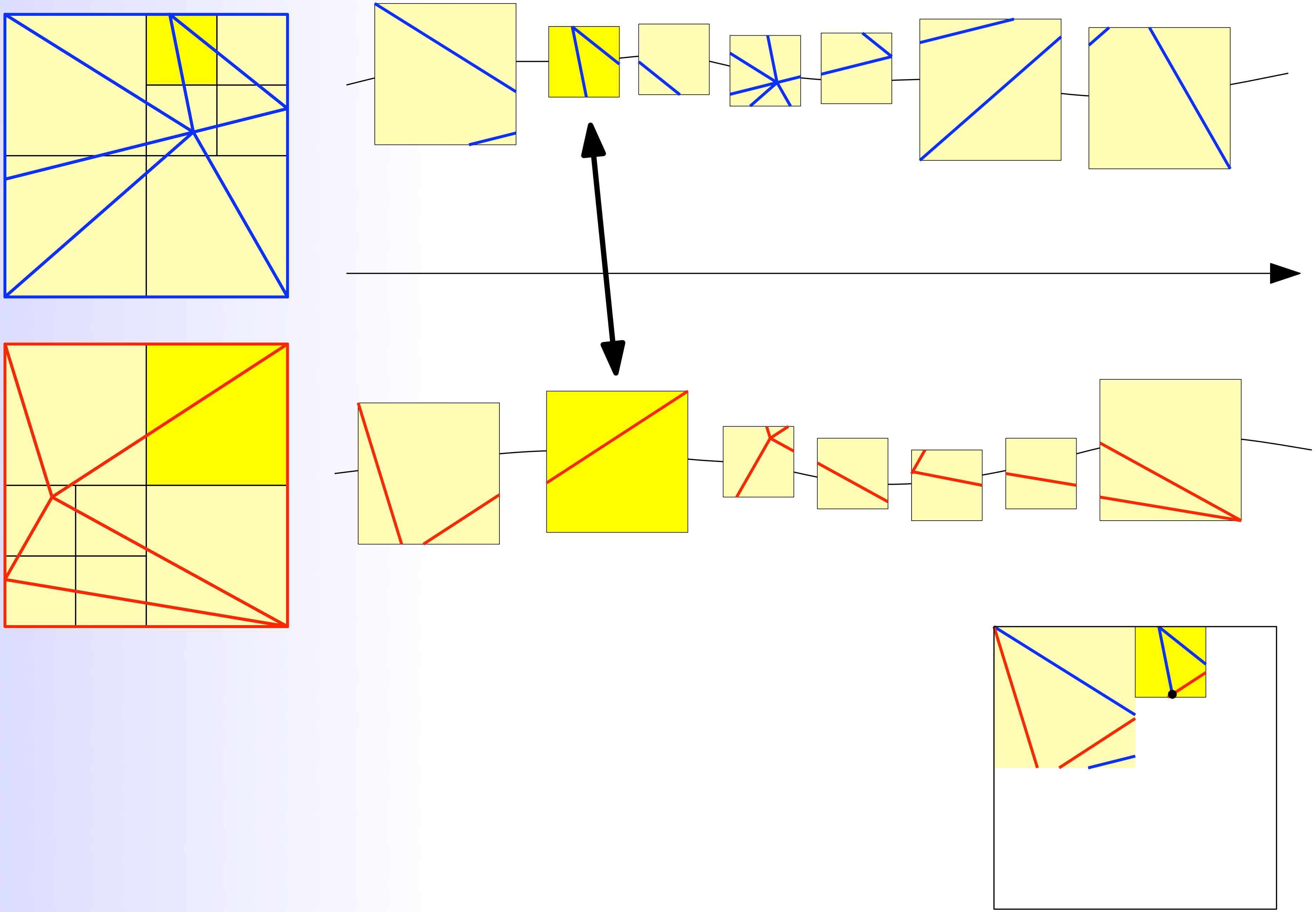
Map overlay with quadtrees in Z-order



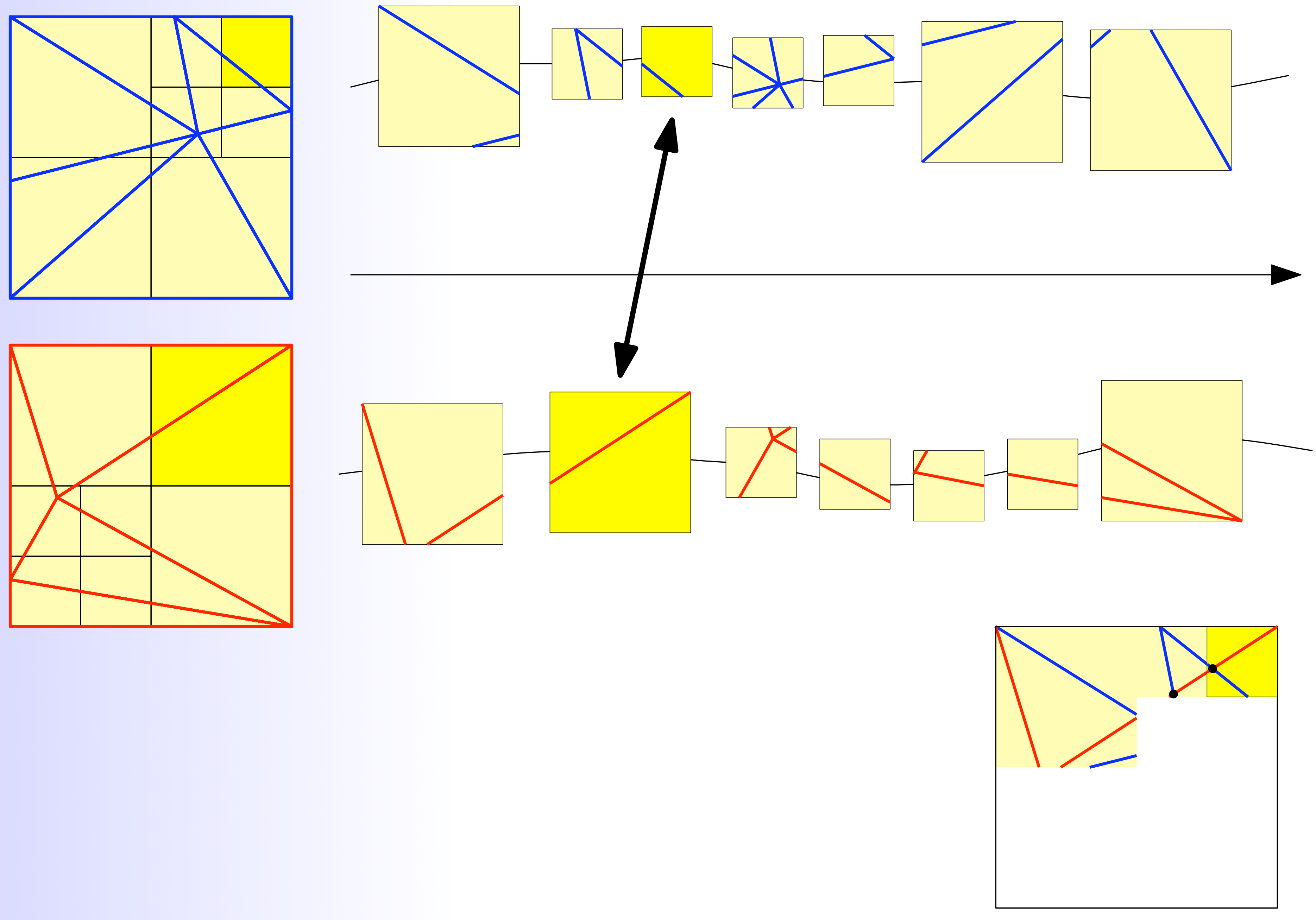
Map overlay with quadtrees in Z-order



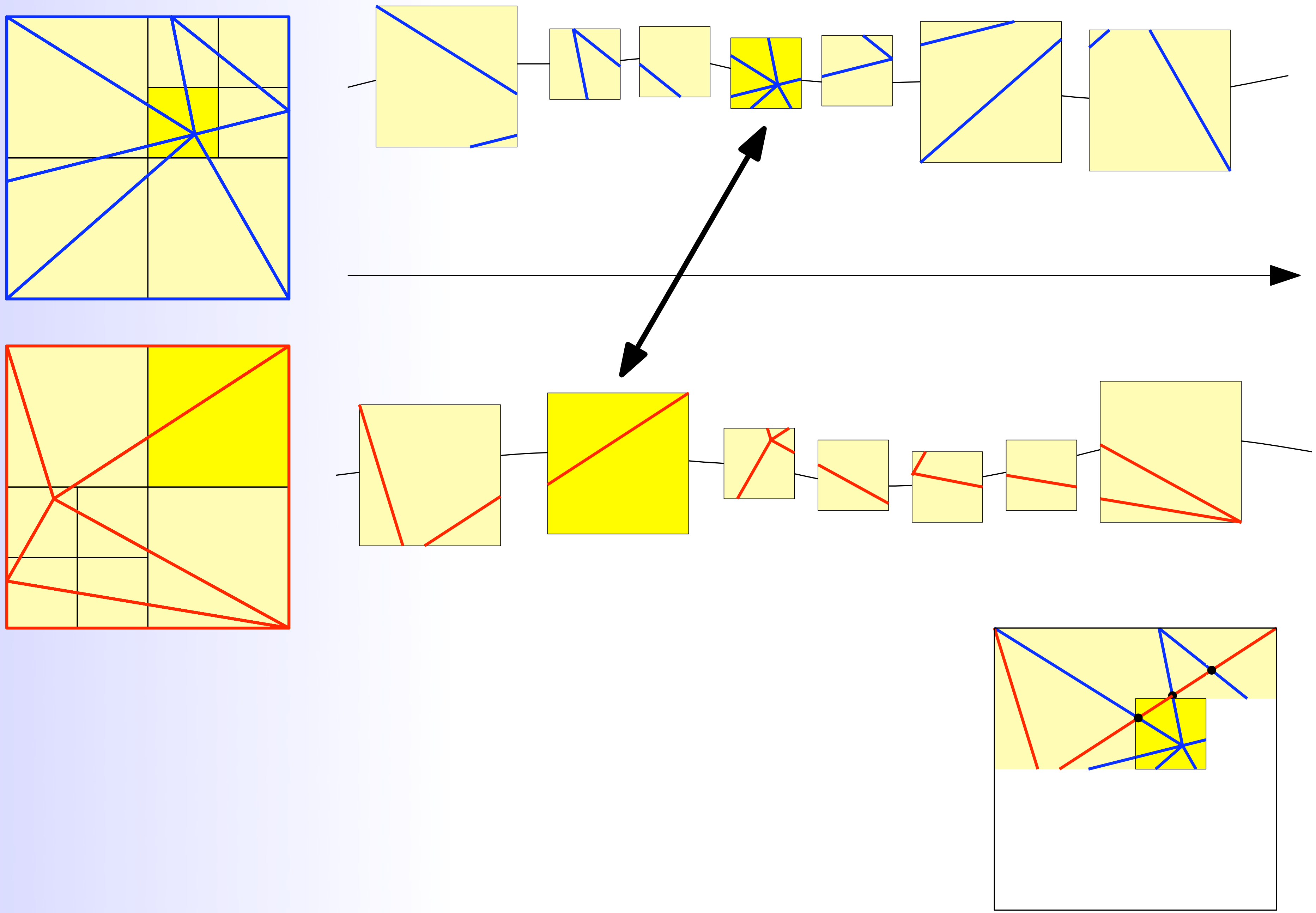
Map overlay with quadtrees in Z-order



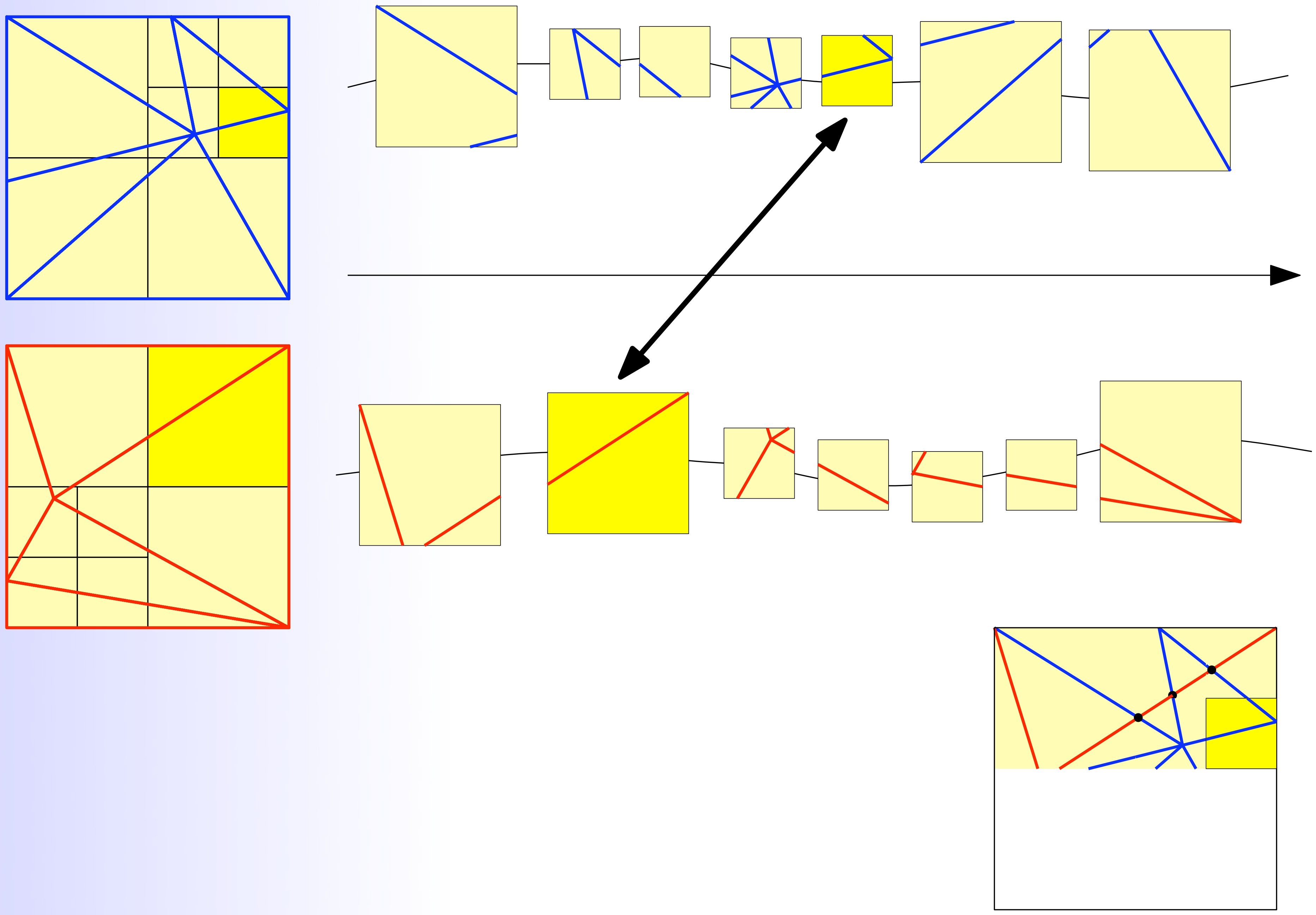
Map overlay with quadtrees in Z-order



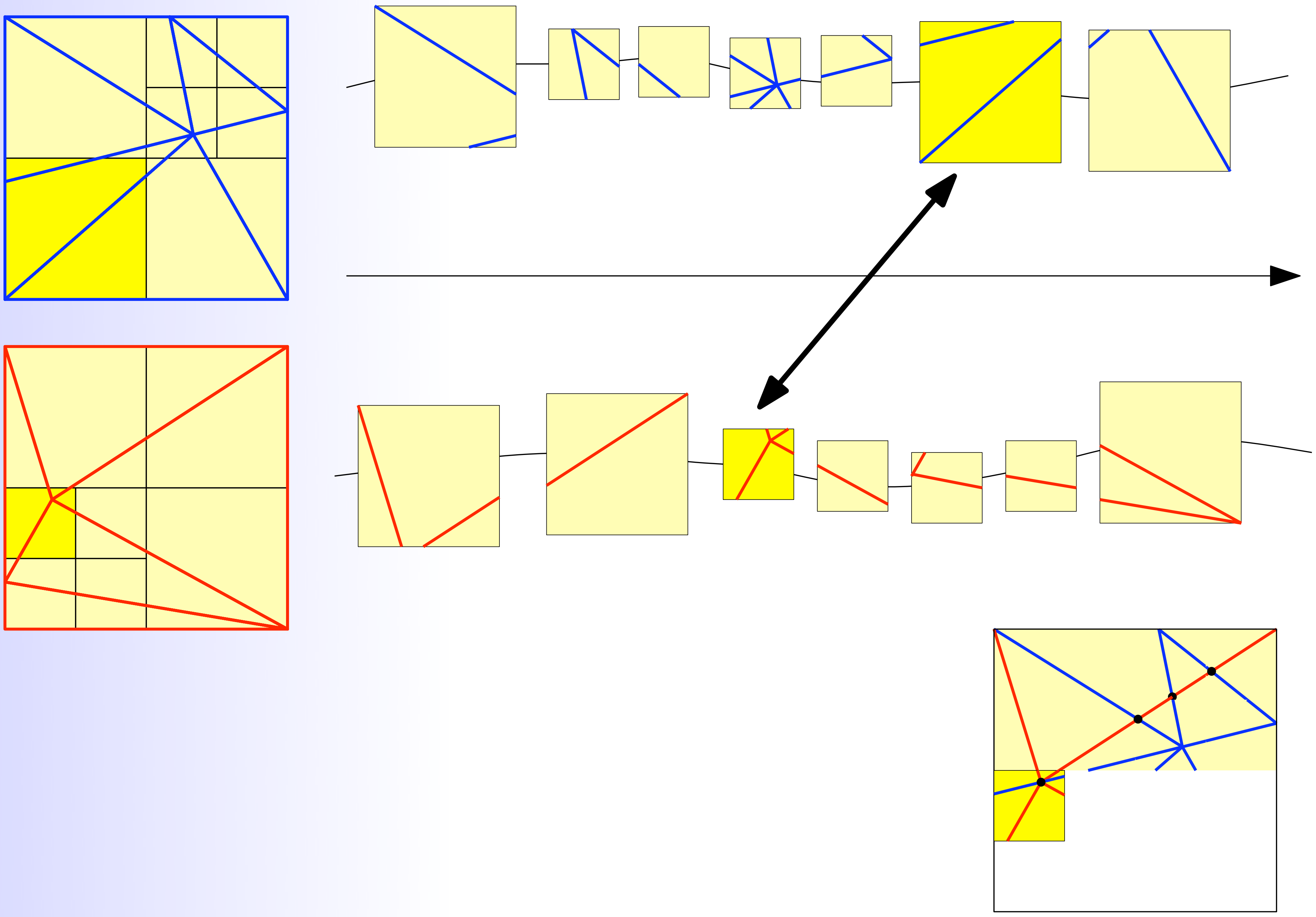
Map overlay with quadtrees in Z-order



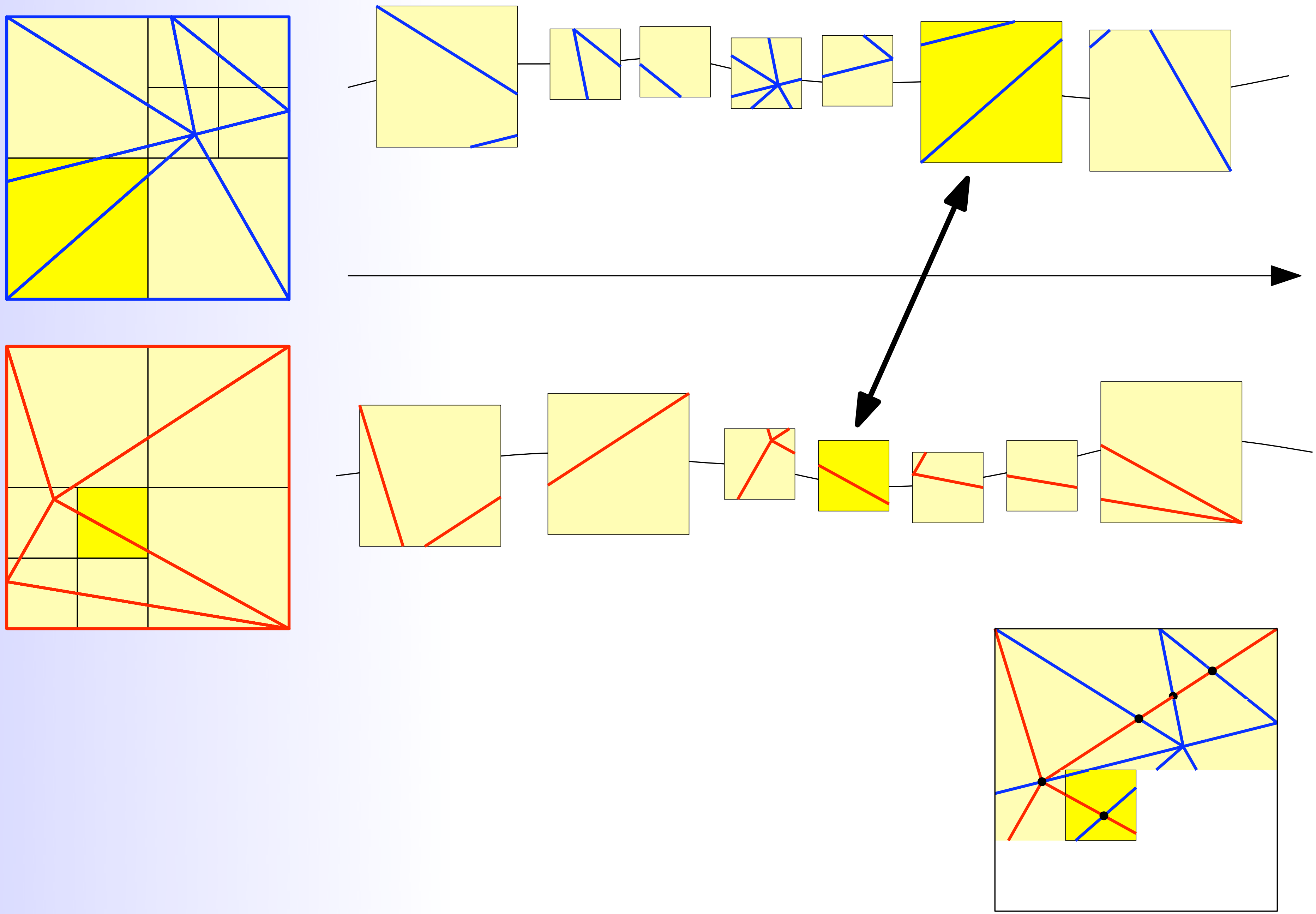
Map overlay with quadtrees in Z-order



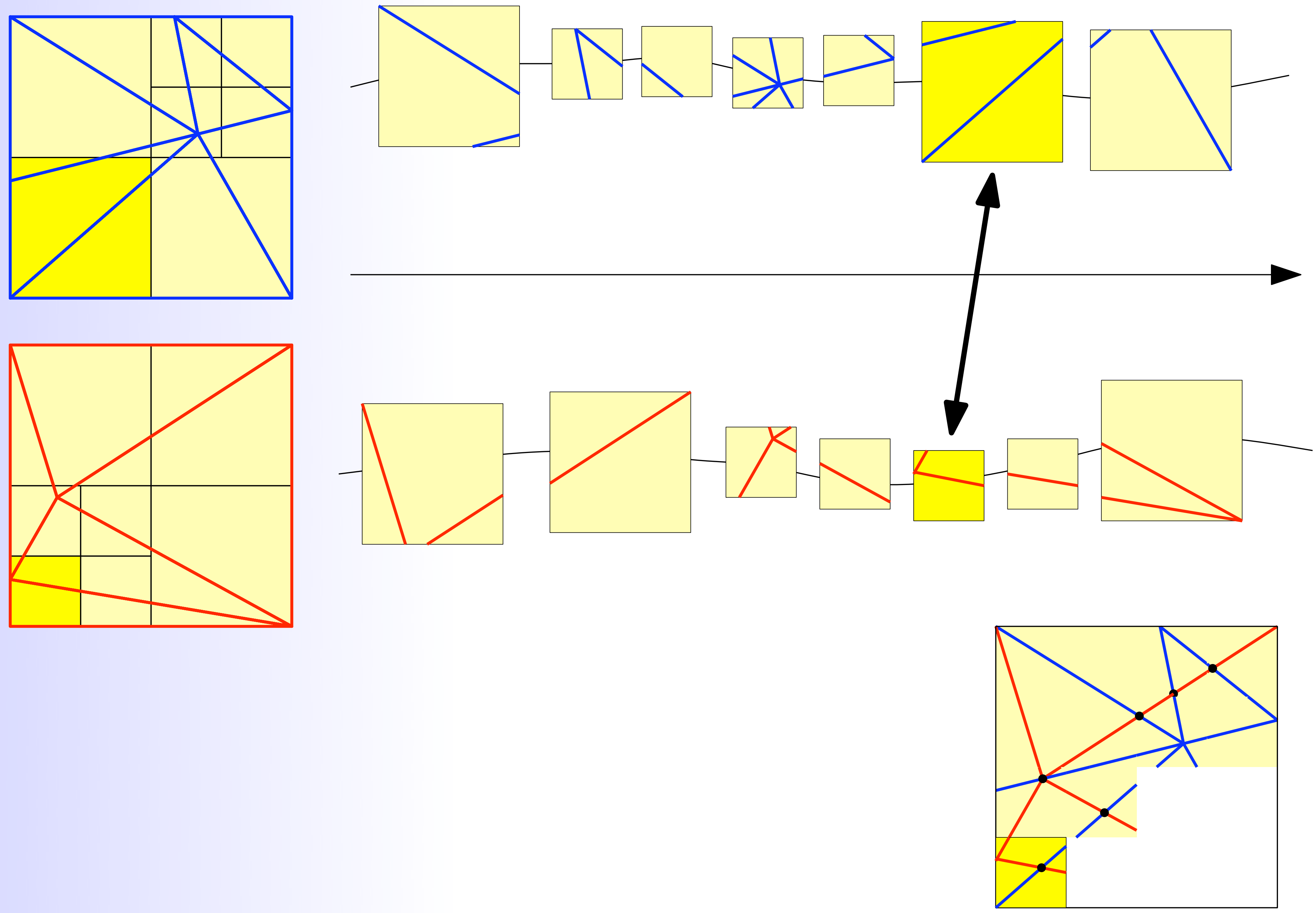
Map overlay with quadtrees in Z-order



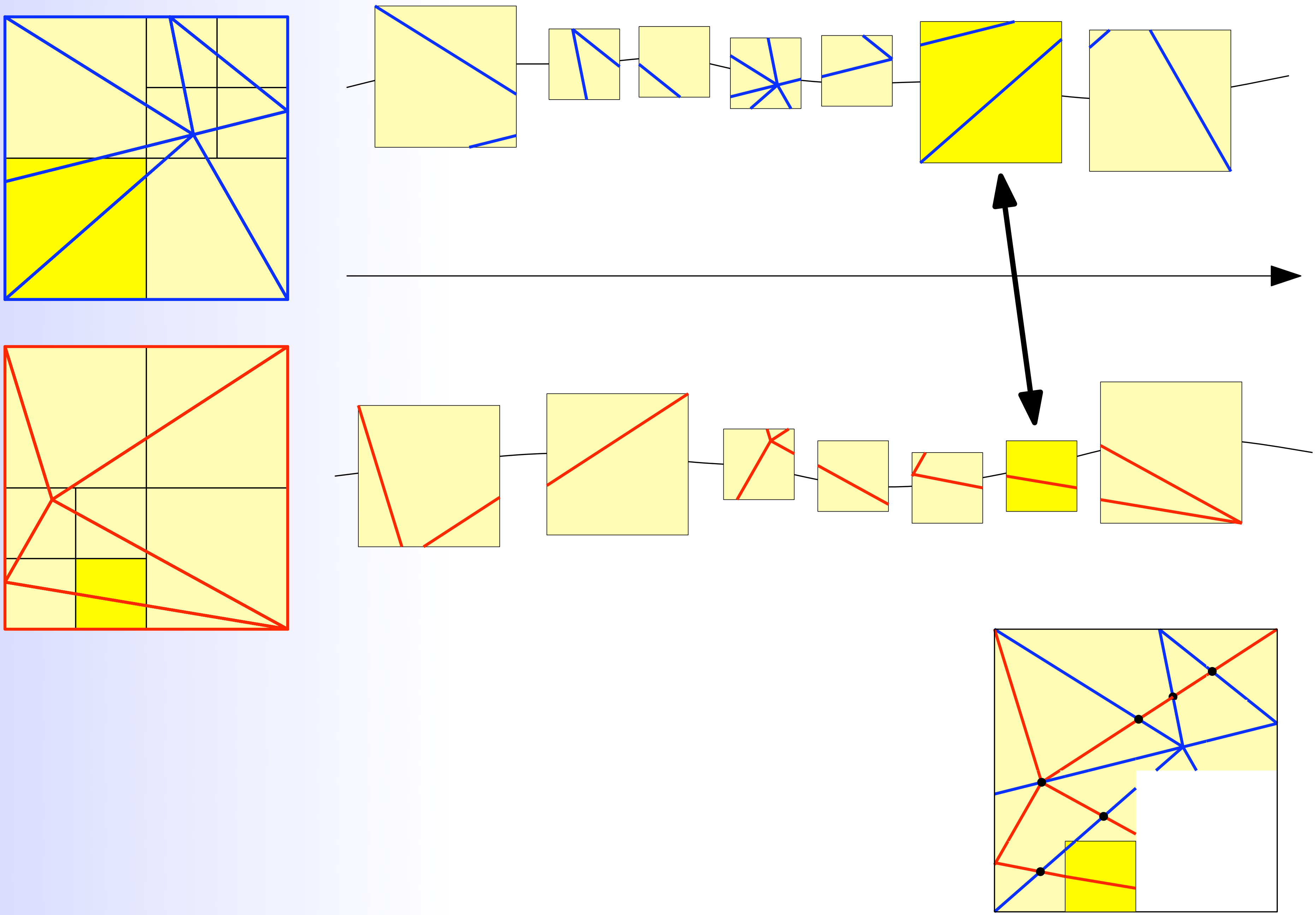
Map overlay with quadtrees in Z-order



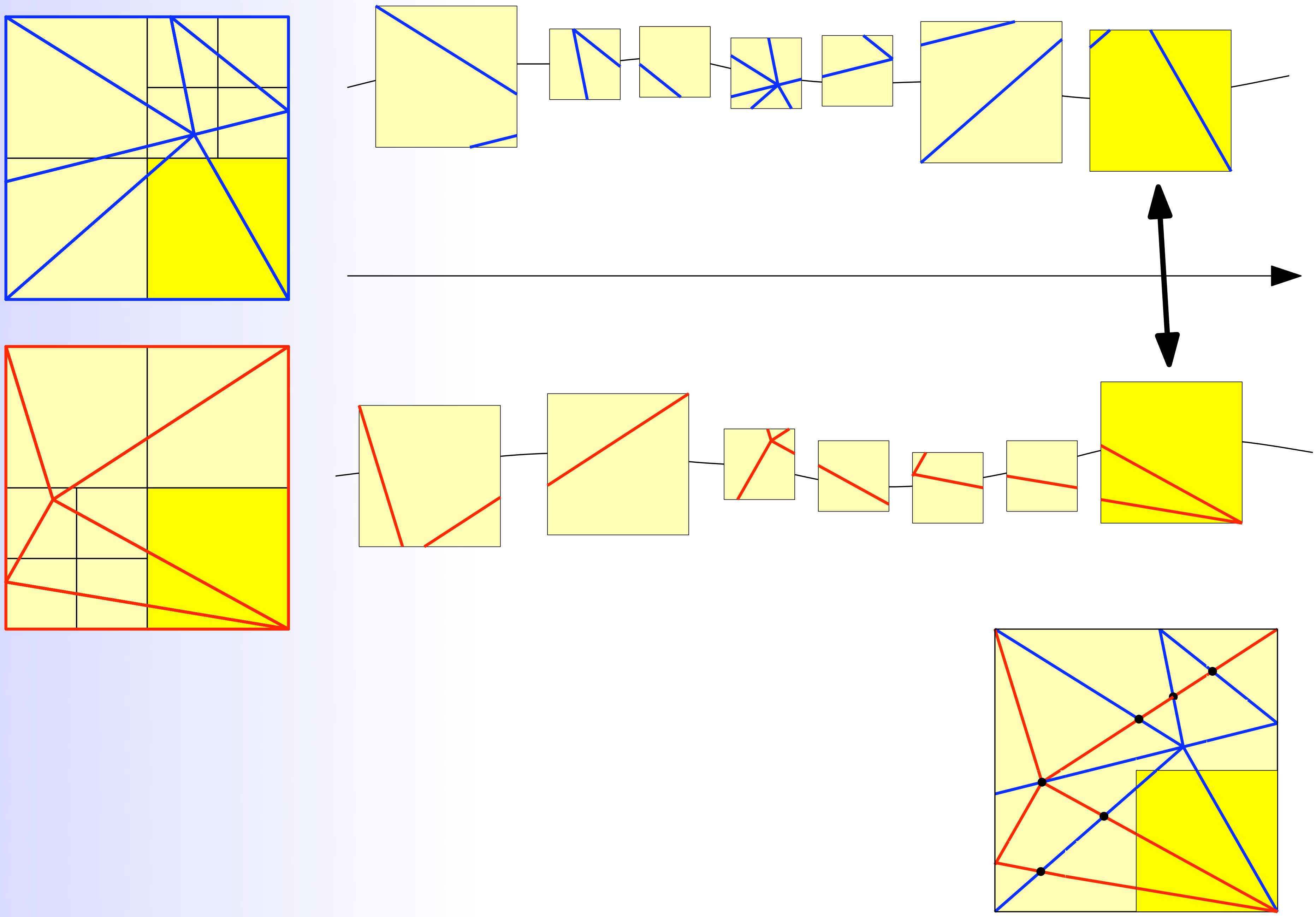
Map overlay with quadtrees in Z-order



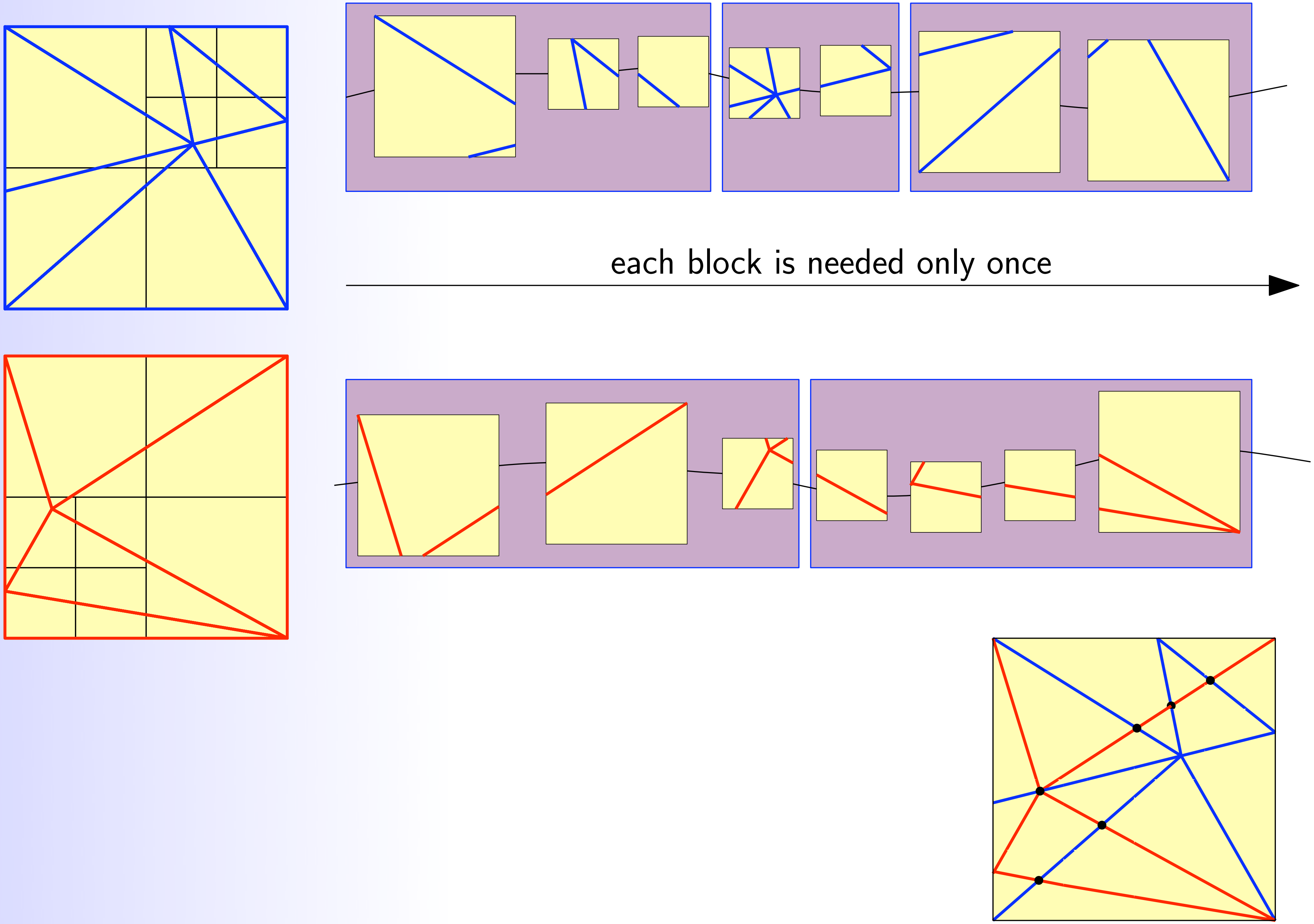
Map overlay with quadtrees in Z-order



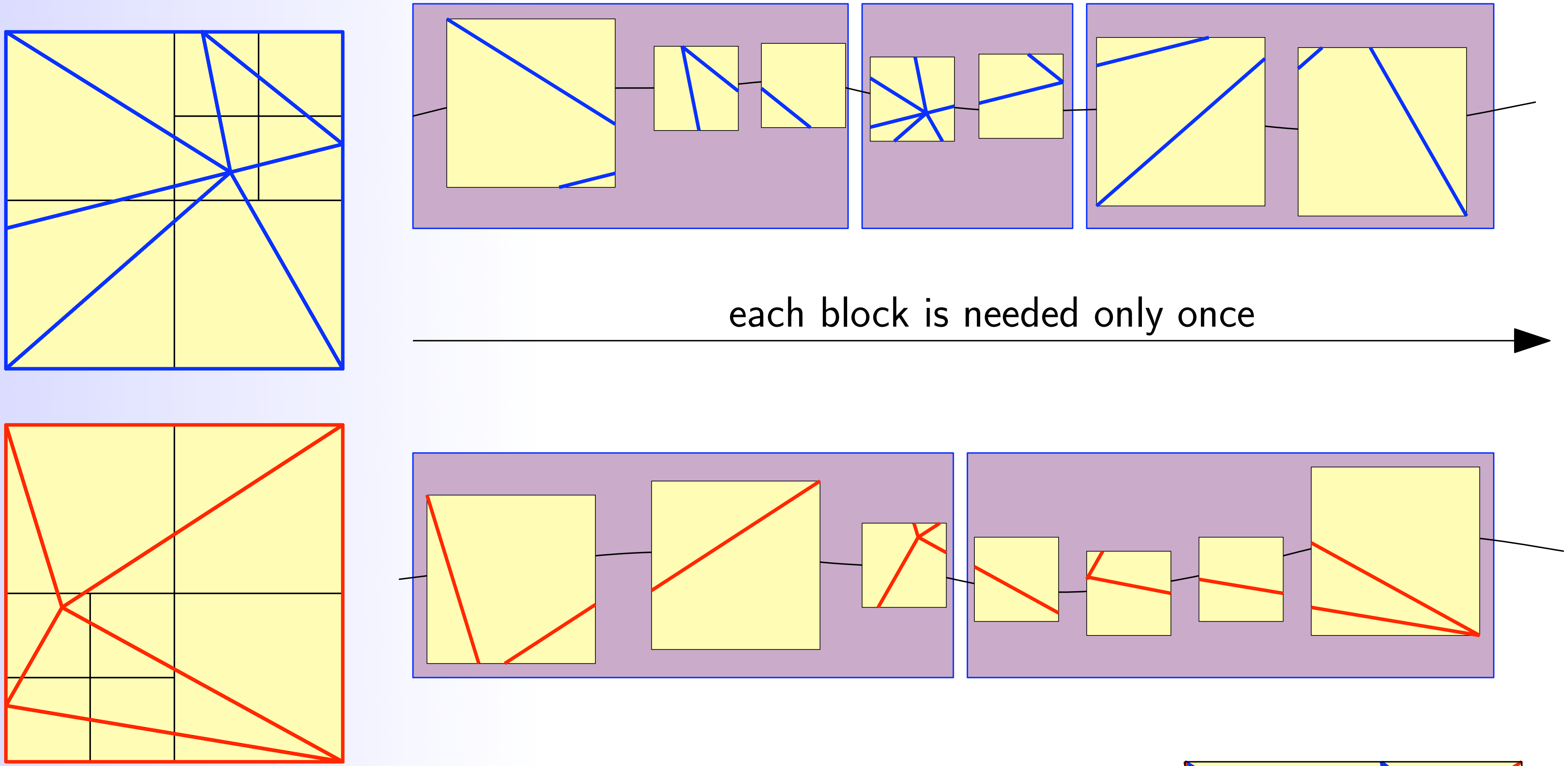
Map overlay with quadtrees in Z-order



Map overlay with quadtrees in Z-order



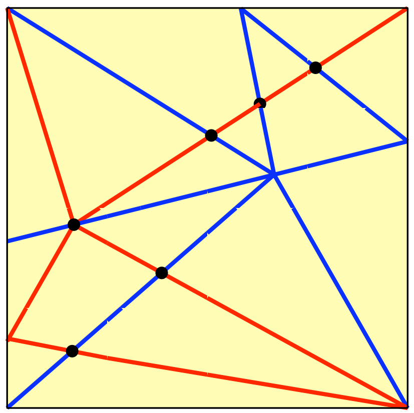
Map overlay with quadtrees in Z-order



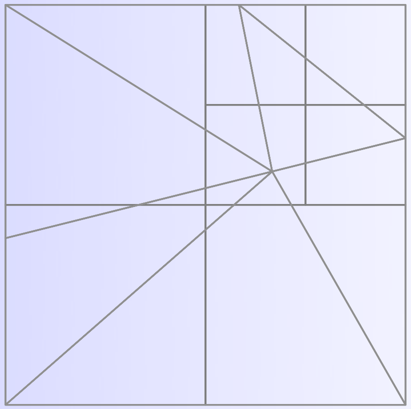
n: number of triangles; B: disk block size

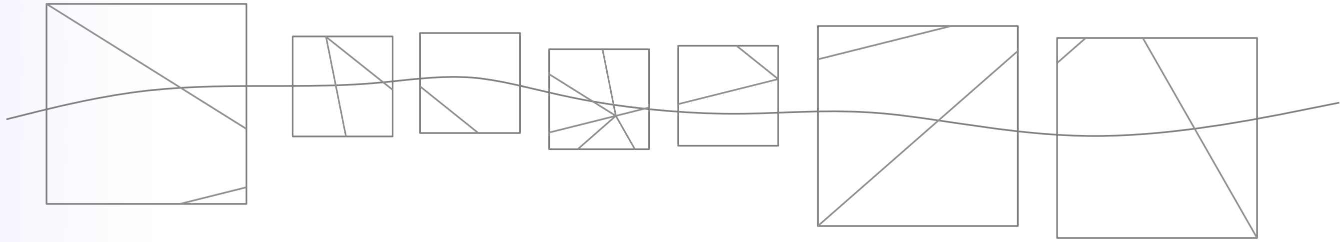
Ideally: $O(n)$ quadtree cells, $O(1)$ edges each

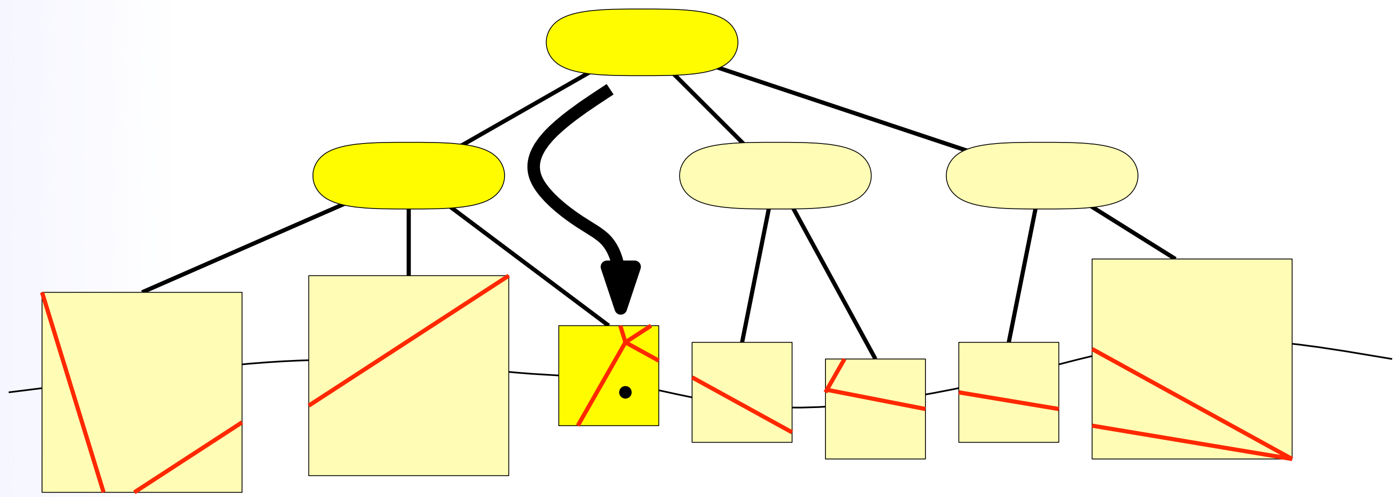
$\longrightarrow$ Overlay in $O(scan(n)) = O(n/B)$ I/O's.

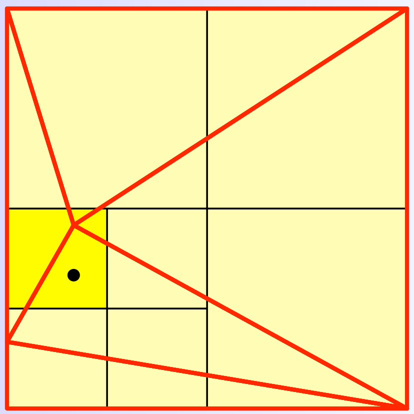


Map overlay with quadtrees in Z-order







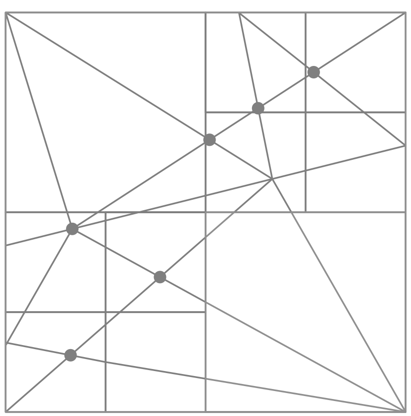


n: number of triangles; B: disk block size

Ideally: $O(n)$ quadtree cells, $O(1)$ edges each

$\rightarrow$ Overlay in $O(scan(n)) = O(n/B)$ I/O's.

$\rightarrow$ Point location with B-tree in $O(\log_B n)$ I/O's.



How to get that quadtree in Z-order (for triangulations of unit square)

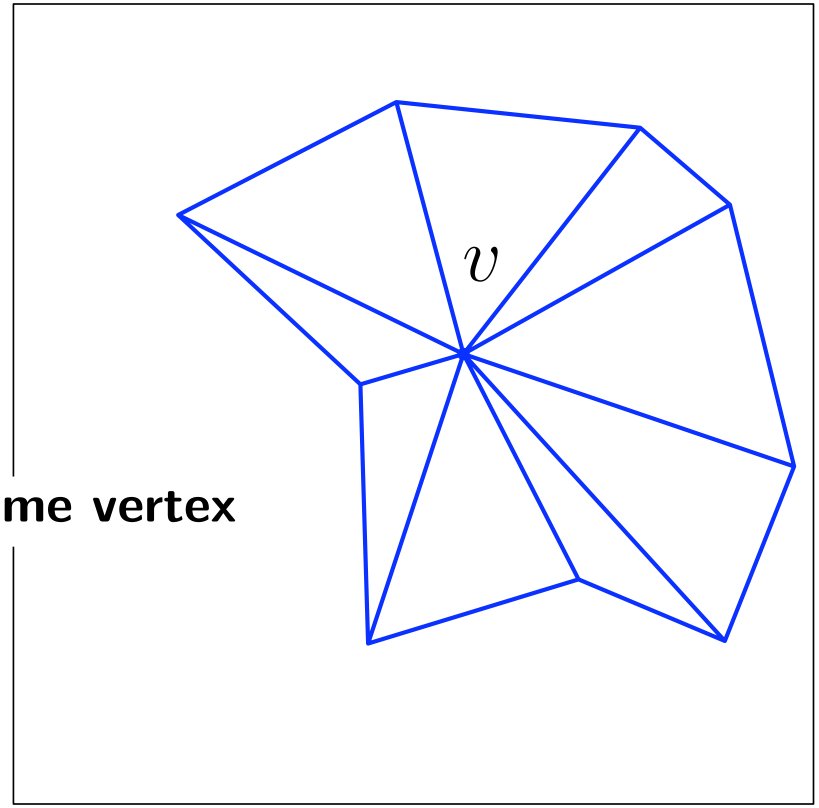
Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex

   - output each cell that is completely inside $star(v)$
2. Sort cells into Z-order (removing duplicates)



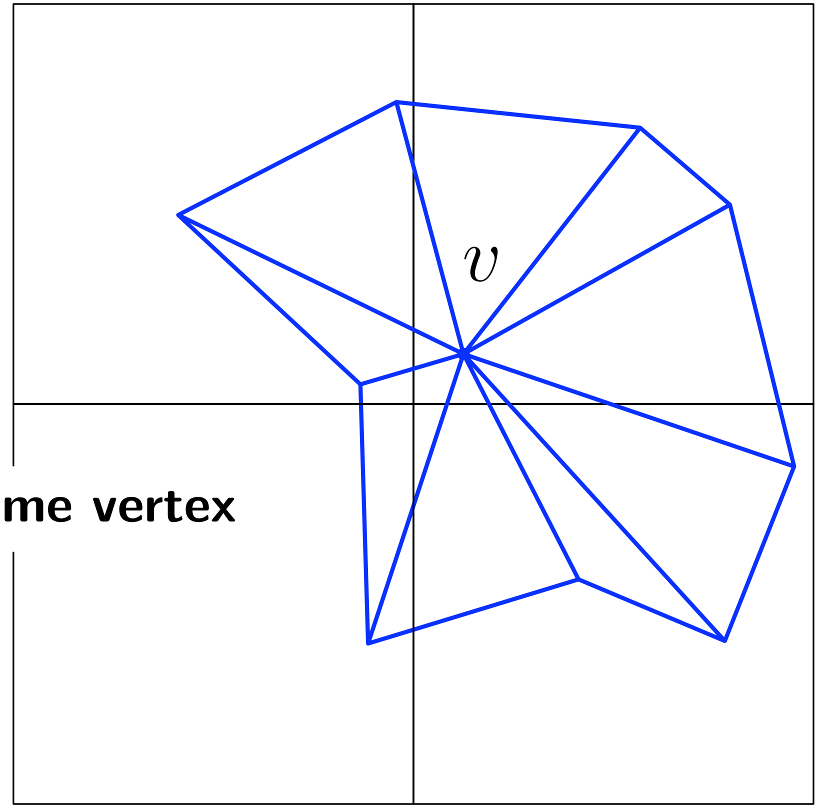
How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex



How to get that quadtree in Z-order (for triangulations of unit square)

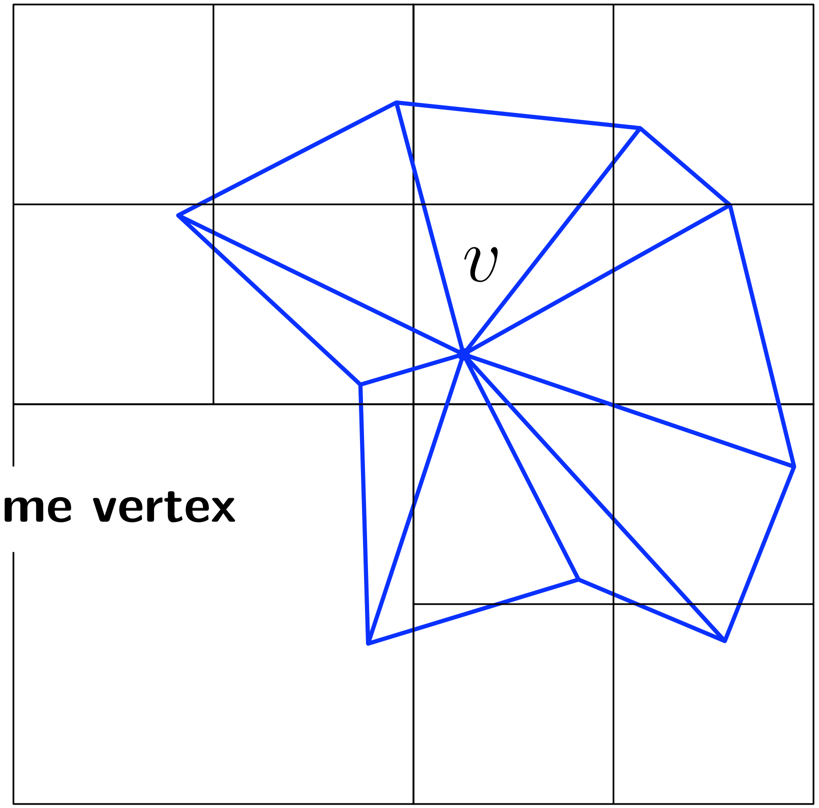
Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:

- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

Stop splitting when all edges incident to same vertex



How to get that quadtree in Z-order (for triangulations of unit square)

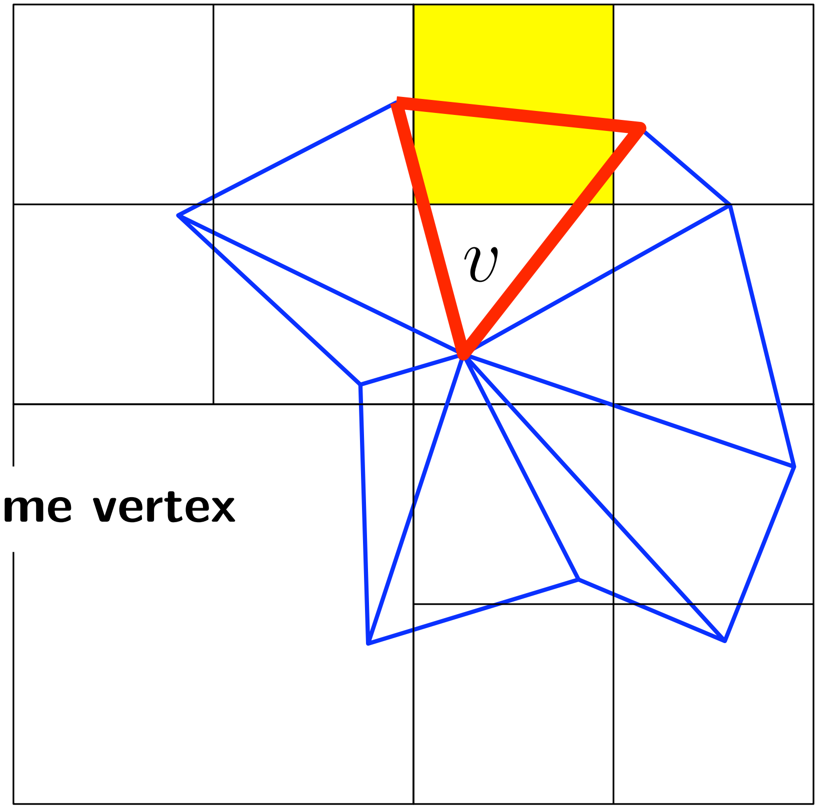
Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:

   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex



How to get that quadtree in Z-order (for triangulations of unit square)

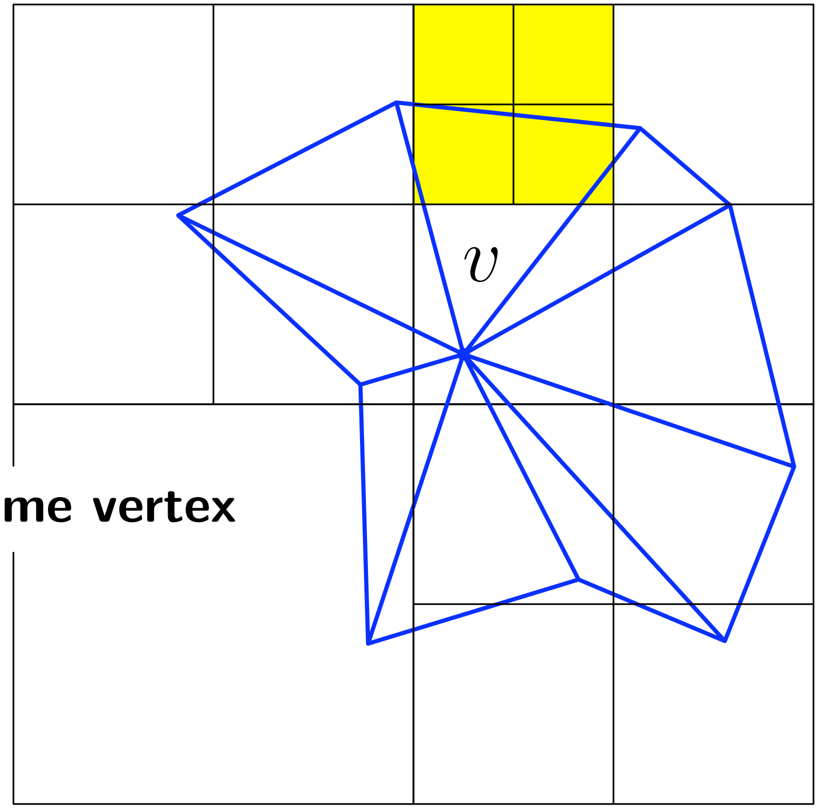
Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:

- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

Stop splitting when all edges incident to same vertex



How to get that quadtree in Z-order (for triangulations of unit square)

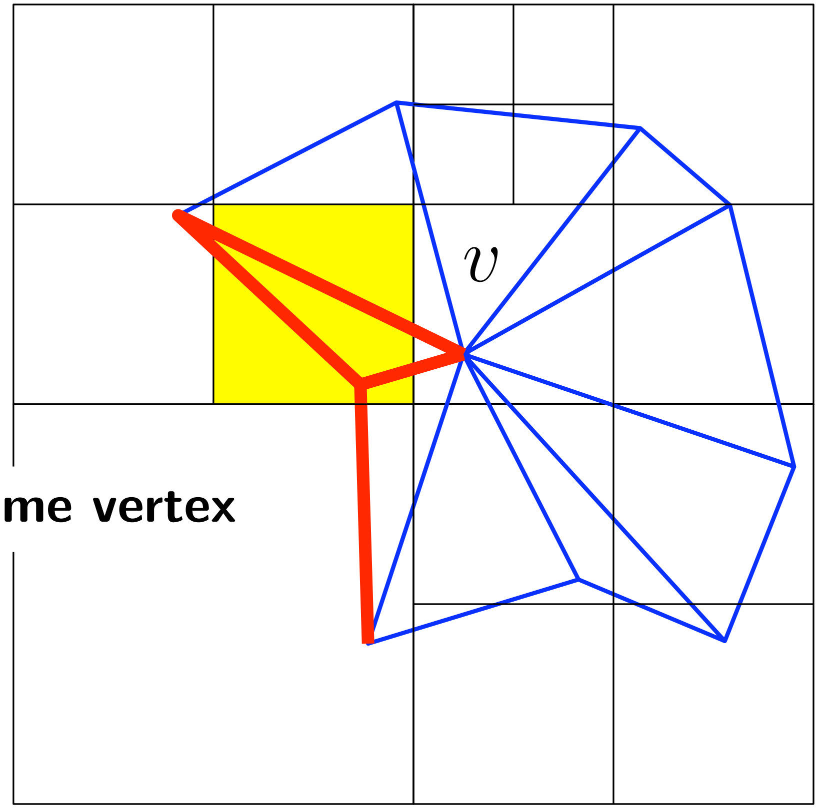
Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:

   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex



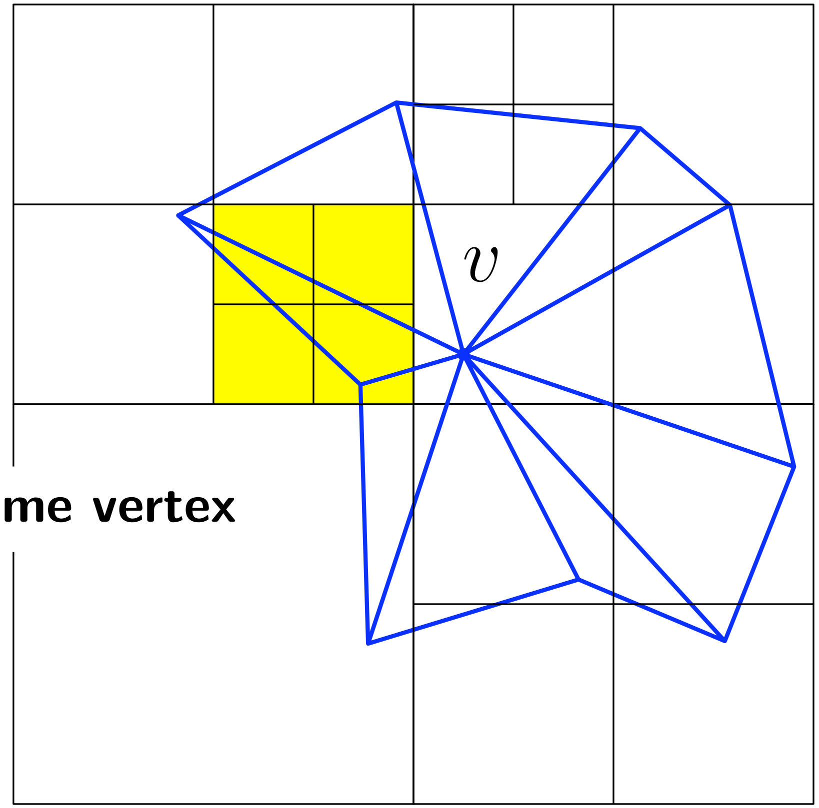
How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex



How to get that quadtree in Z-order (for triangulations of unit square)

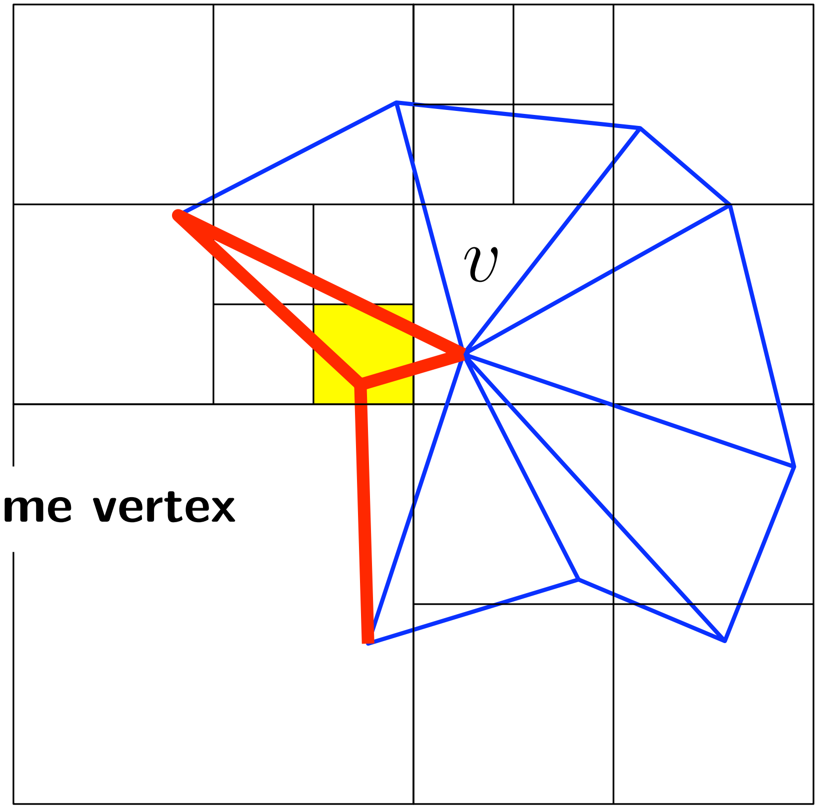
Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:

   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex



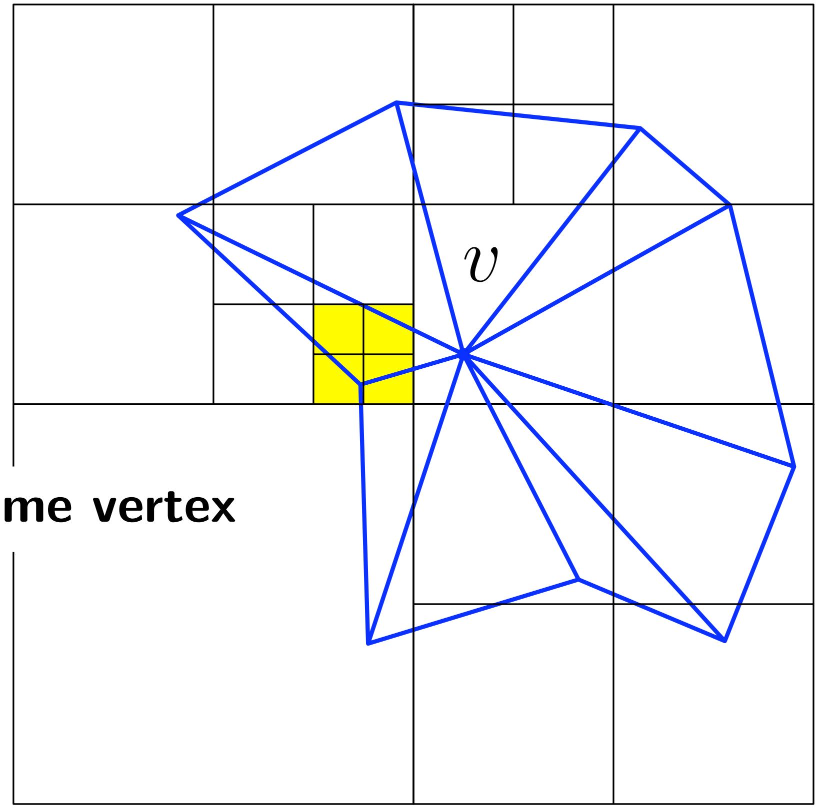
How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex



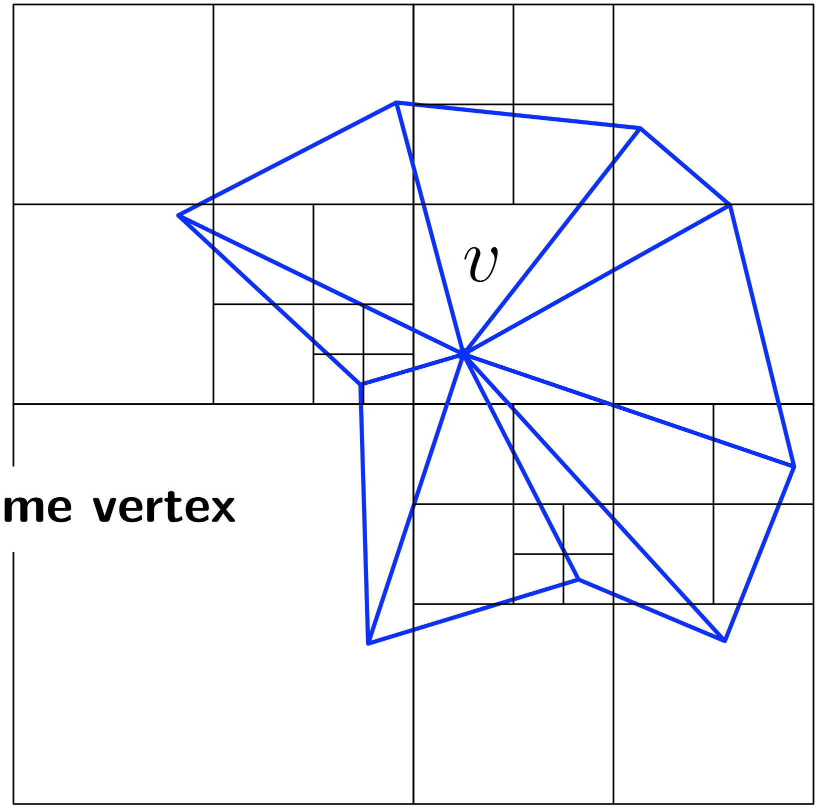
How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex



How to get that quadtree in Z-order (for triangulations of unit square)

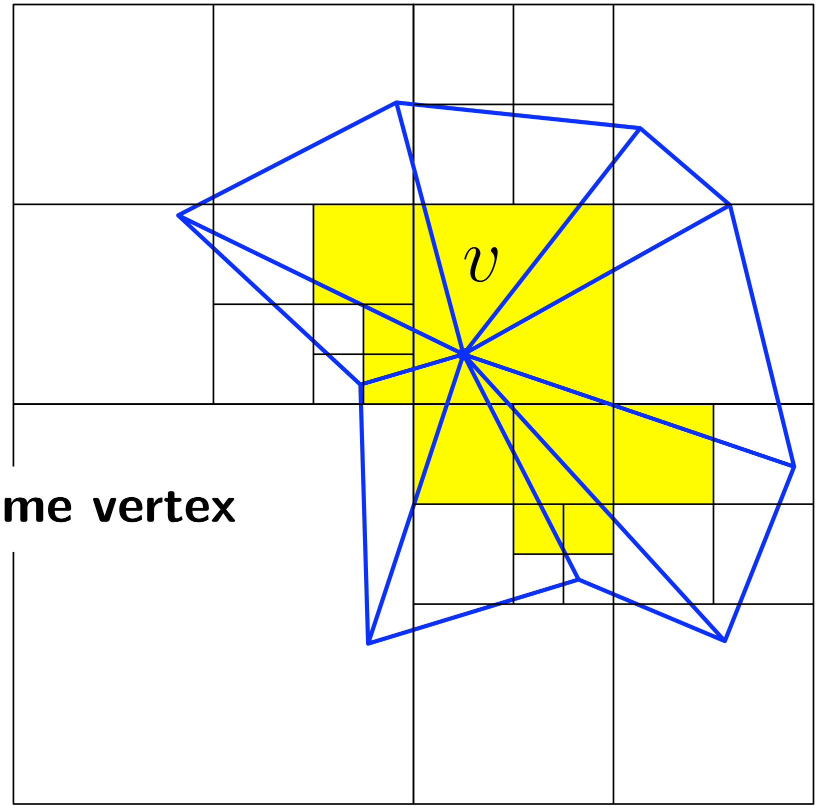
Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex

   - output each cell that is completely inside $star(v)$



How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

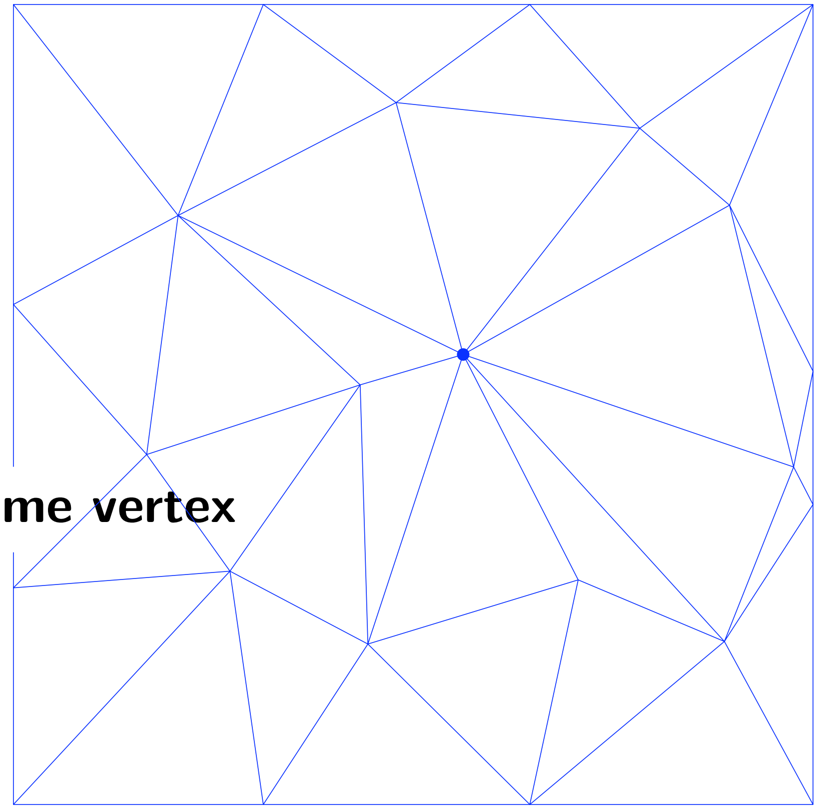
Algorithm:

1. For each vertex v:

- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$



How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

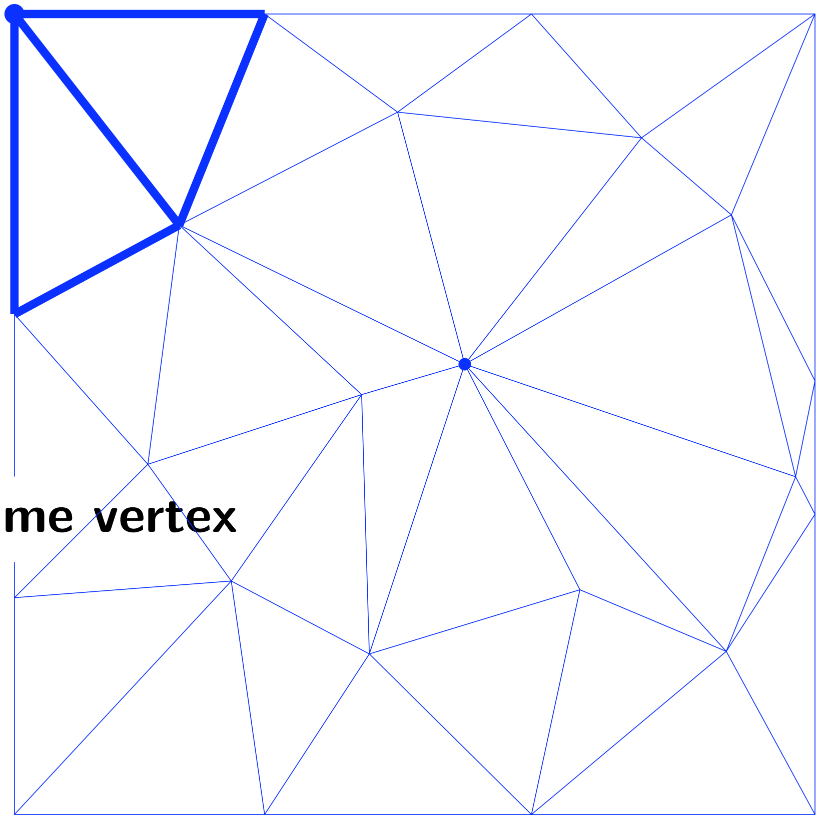
Algorithm:

1. For each vertex v:

- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$



How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

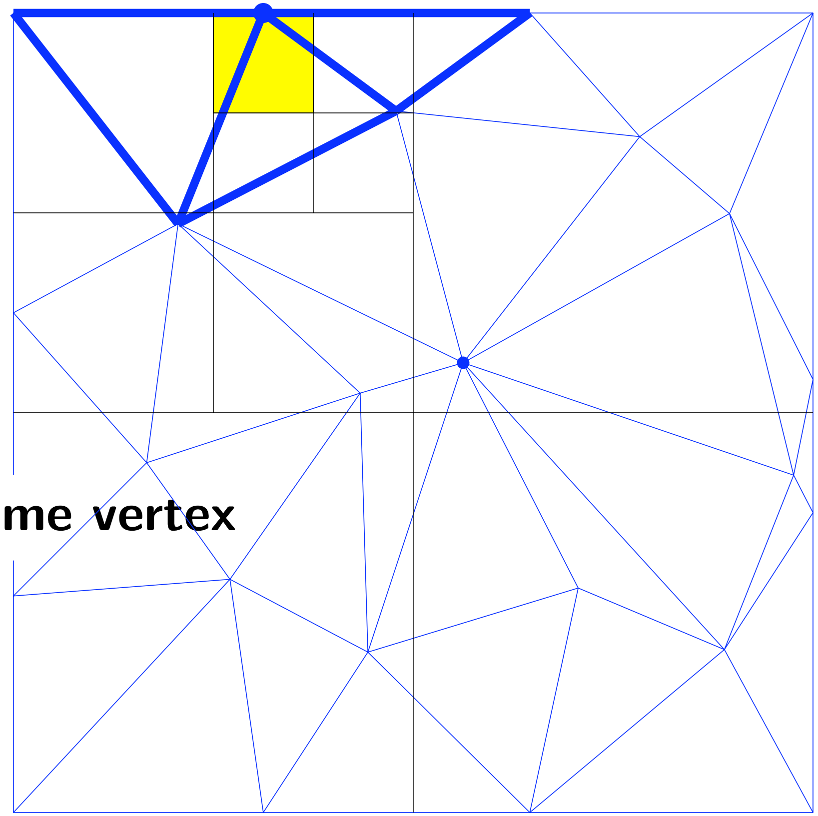
Algorithm:

1. For each vertex v:

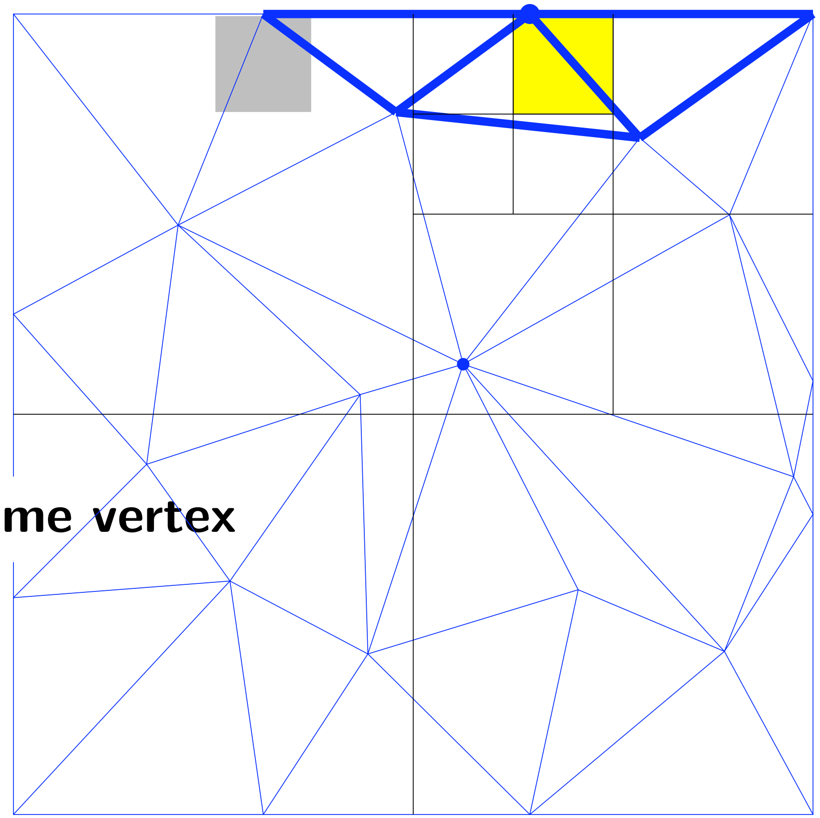
- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$



How to get that quadtree in Z-order (for triangulations of unit square)



Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex

   - output each cell that is completely inside $star(v)$

How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

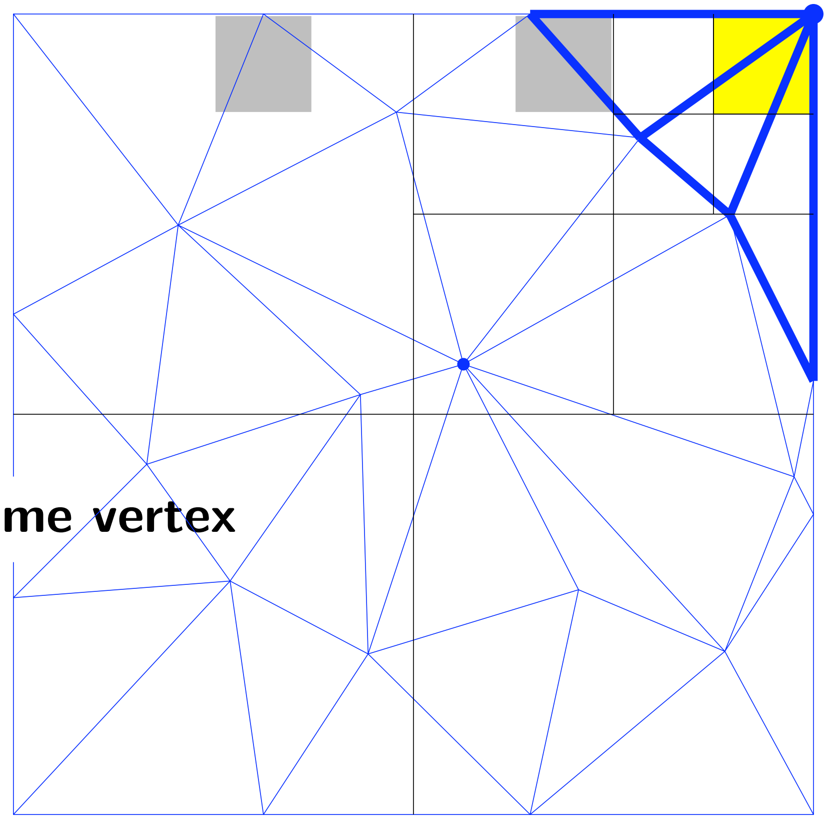
Algorithm:

1. For each vertex v:

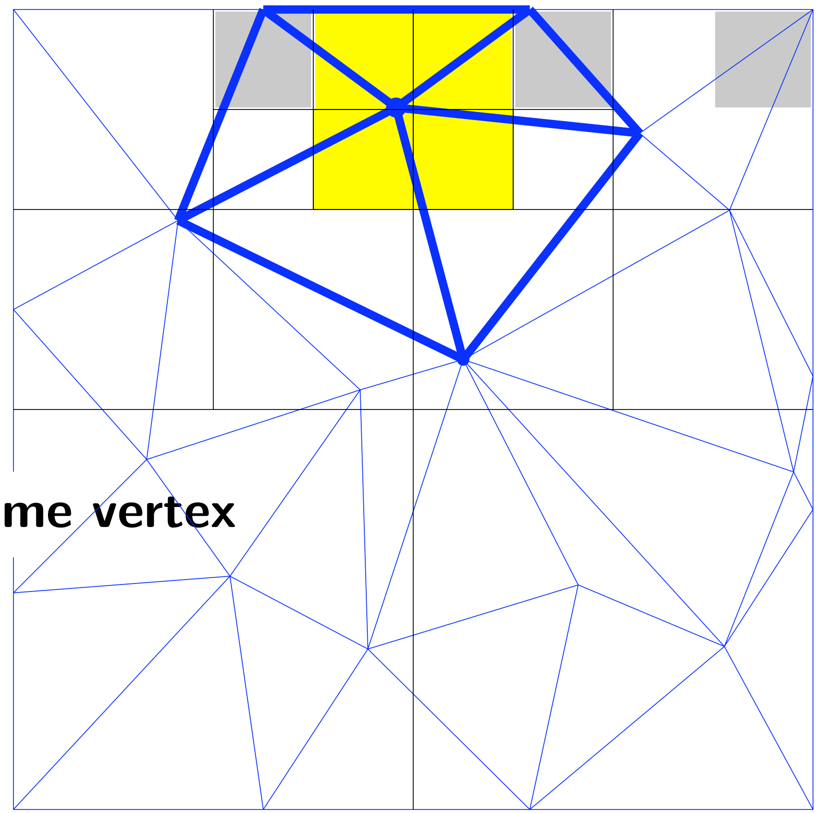
- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$



How to get that quadtree in Z-order (for triangulations of unit square)



Input: file with for each vertex its adjacency list.

Algorithm:

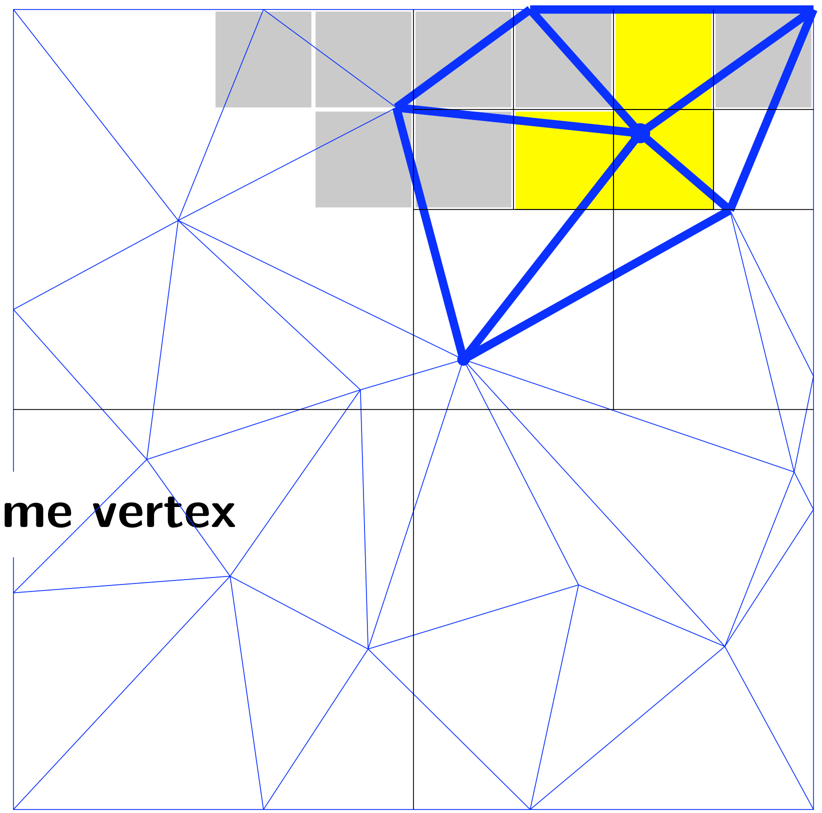
1. For each vertex v:

- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$

How to get that quadtree in Z-order (for triangulations of unit square)



Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:

- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$

How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

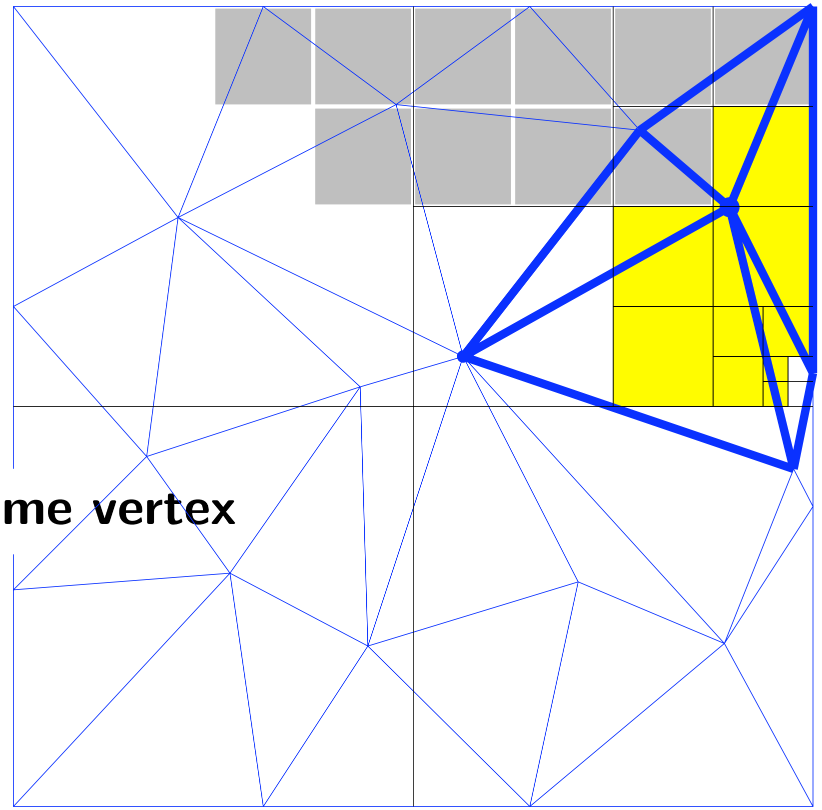
Algorithm:

1. For each vertex v:

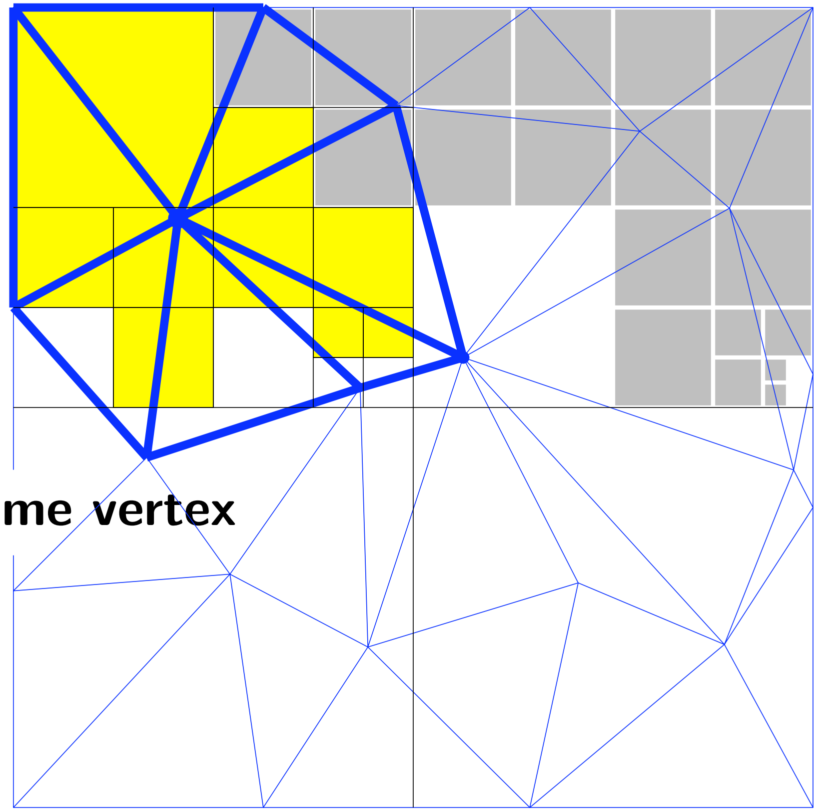
- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$



How to get that quadtree in Z-order (for triangulations of unit square)



Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:

- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$

How to get that quadtree in Z-order (for triangulations of unit square)

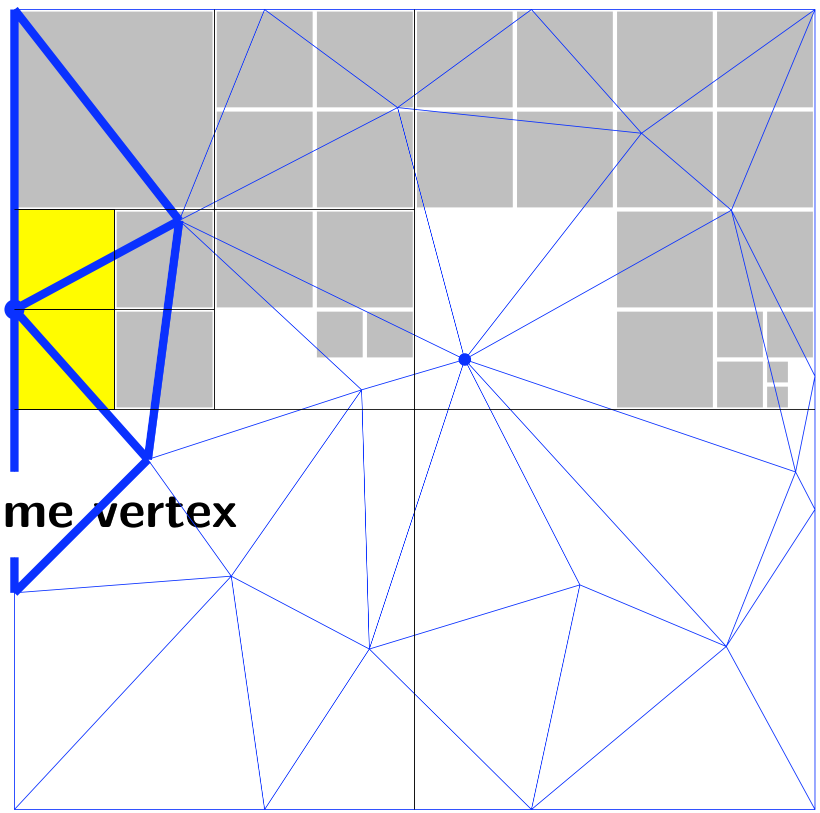
Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex

   - output each cell that is completely inside $star(v)$



How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

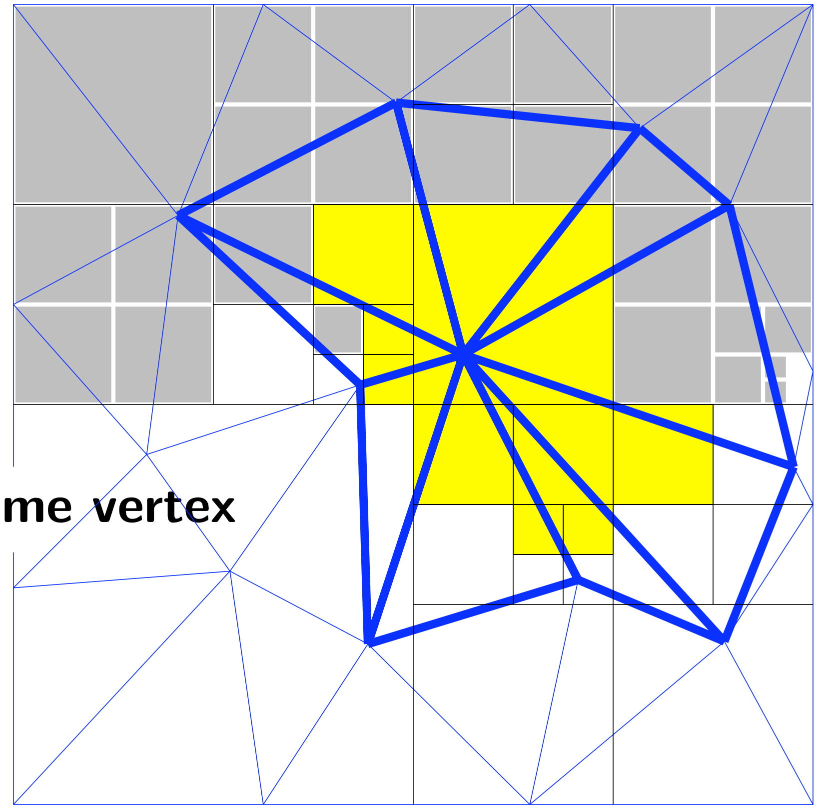
Algorithm:

1. For each vertex v:

- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$



How to get that quadtree in Z-order (for triangulations of unit square)

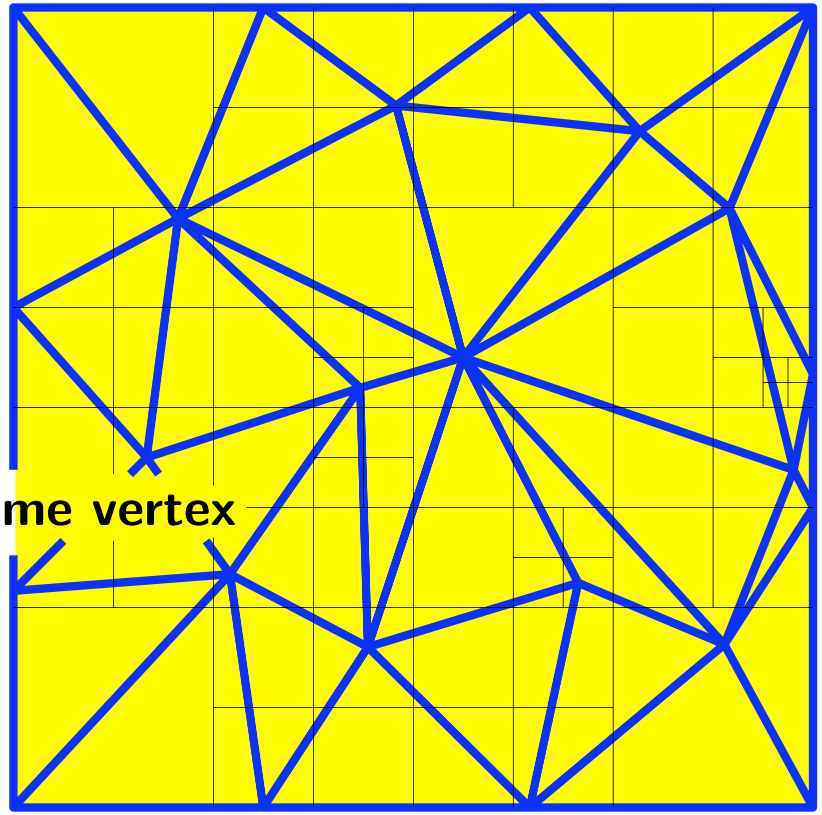
Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

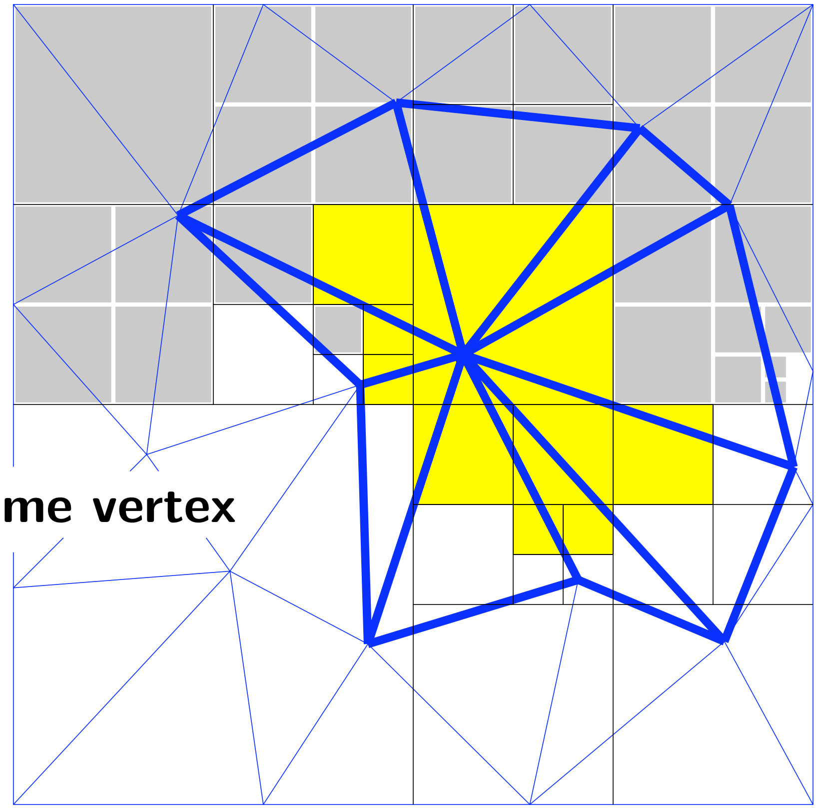
     Stop splitting when all edges incident to same vertex

   - output each cell that is completely inside $star(v)$
2. Sort cells into Z-order (removing duplicates)



How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.



Algorithm:

1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex

   - output each cell that is completely inside $star(v)$
2. Sort cells into Z-order (removing duplicates)

To prove for input of n triangles:

- together cells form subdivision of unit square;
- $O(1)$ triangles per cell;
- $O(n)$ cells in total;
- algorithm runs in $O(sort(n))$ I/O's

How to get that quadtree in Z-order (for triangulations of unit square)

Input: file with for each vertex its adjacency list.

Algorithm:

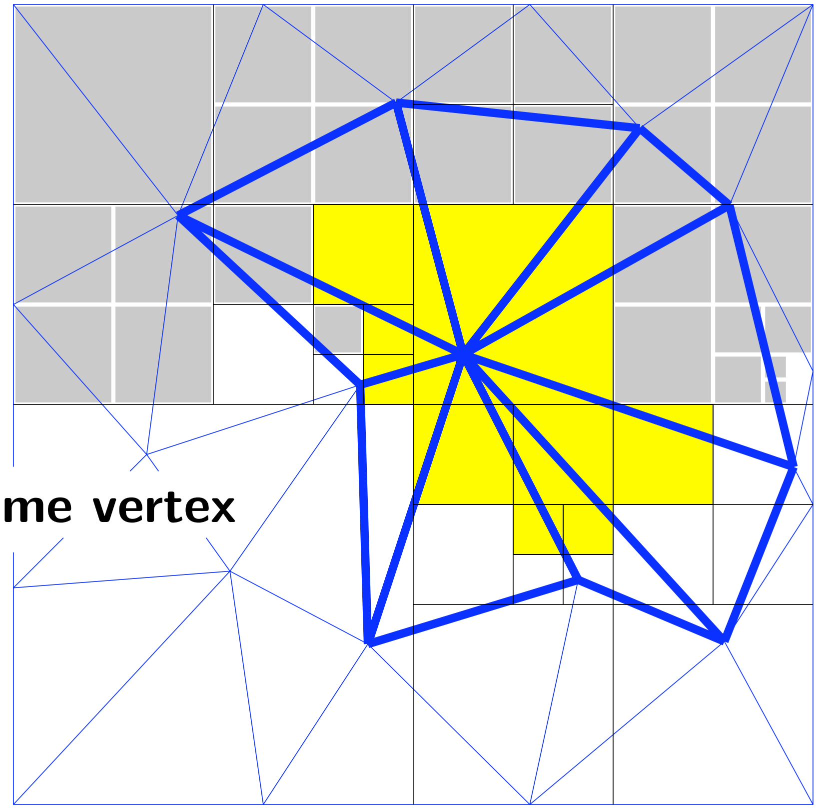
1. For each vertex v:

- load adjacency list in memory;
- build quadtree on $star(v)$ with splitting criterion:

  Stop splitting when all edges incident to same vertex

- output each cell that is completely inside $star(v)$

2. Sort cells into Z-order (removing duplicates)



To prove for input of n triangles:

- together cells form subdivision of unit square;
- $O(1)$ triangles per cell;
- $O(n)$ cells in total;
- algorithm runs in $O(sort(n))$ I/O's

Works if triangles are *fat*:
minimum angle $>$
positive constant independent of n

Quadtrees for Fat Triangulations

- Theorem: Let F be a δ-fat triangulation with n edges. We can construct, in $O(\mathrm{sort}(n/\delta^2))$ IOs a quadtree for F that stores $O(n/\delta)$ cells and $O(n/\delta^2)$ edge-cell intersections.

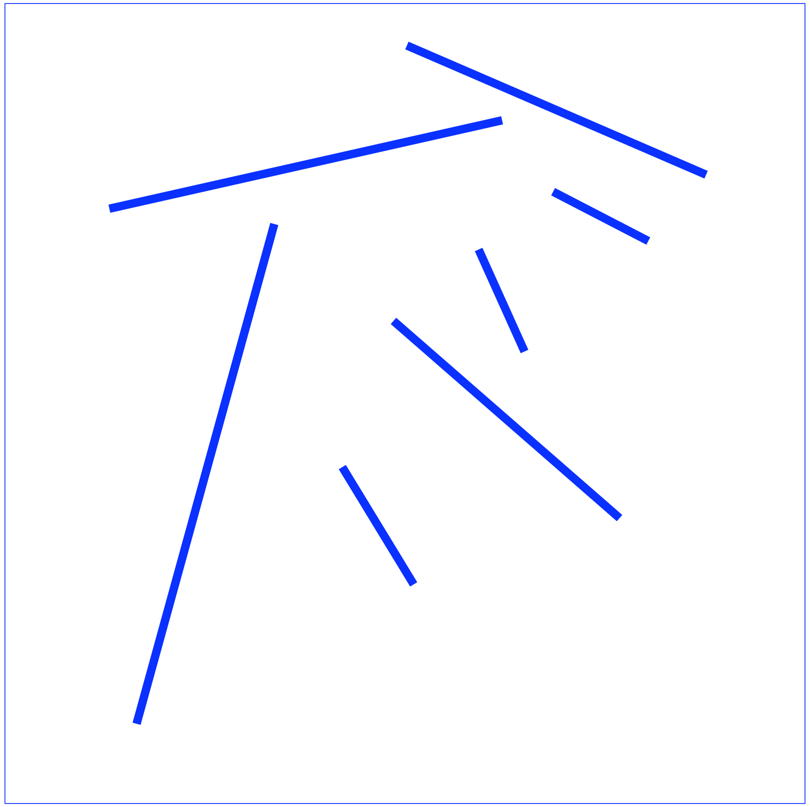
- Given two δ-fat triangulations stored as above, we can find all pairs of intersections in $O(\mathrm{scan}(n/\delta^2))$ IOs.

How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.

Algorithm:

1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order
2. For $i \leftarrow 1$ to m:
   - find smallest cell Q that contains L_i and L_{i+1};
   - output cell boundaries of Q and its subquadrants
3. Sort cell boundaries in Z-order (removing duplicates)
4. Put line segments in cells

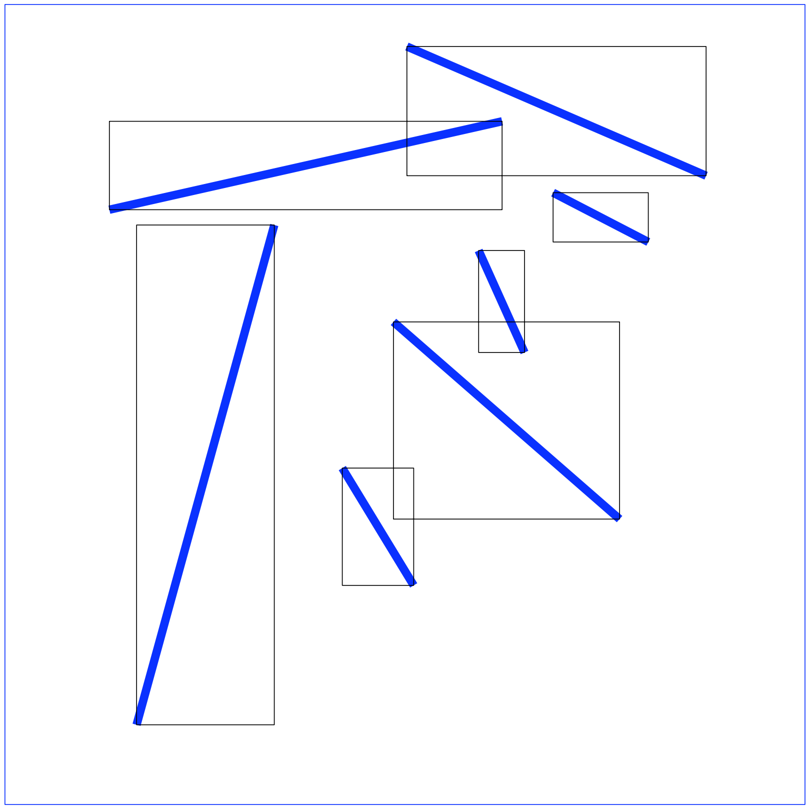


How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.

Algorithm:

1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order

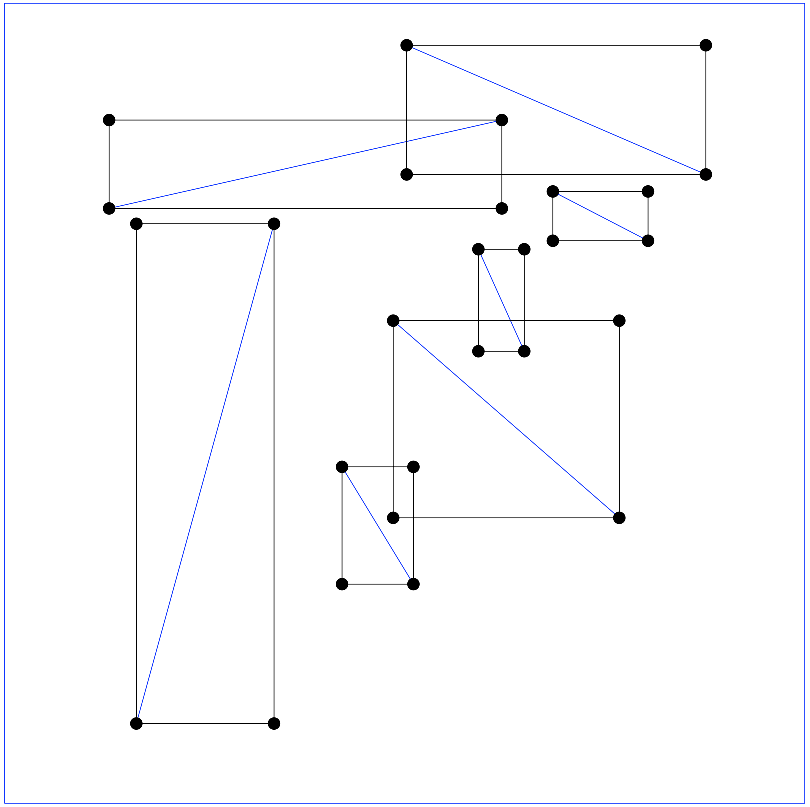


How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.

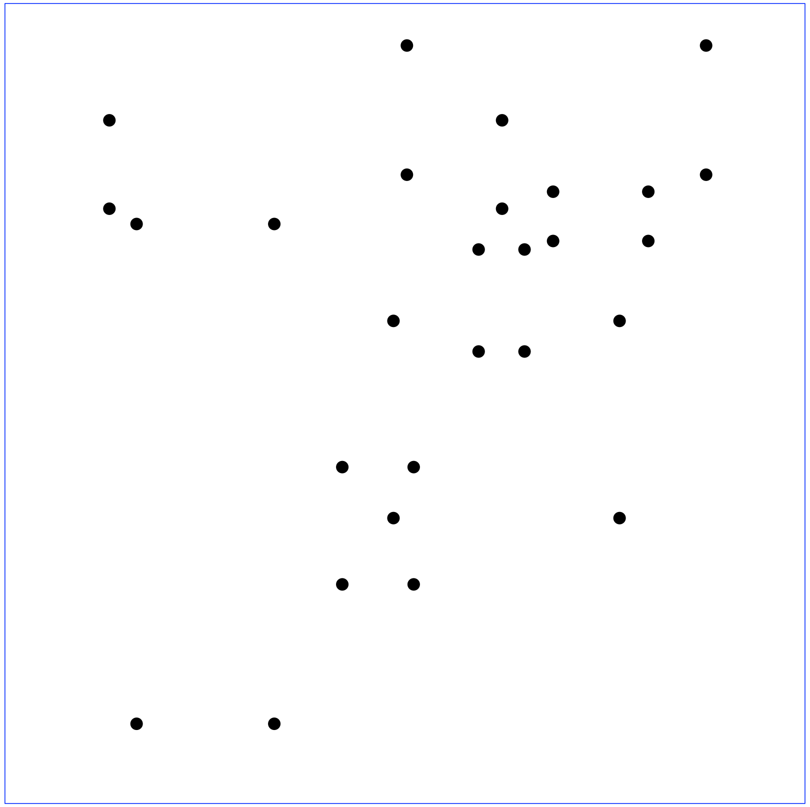
Algorithm:

1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order



How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.



Algorithm:

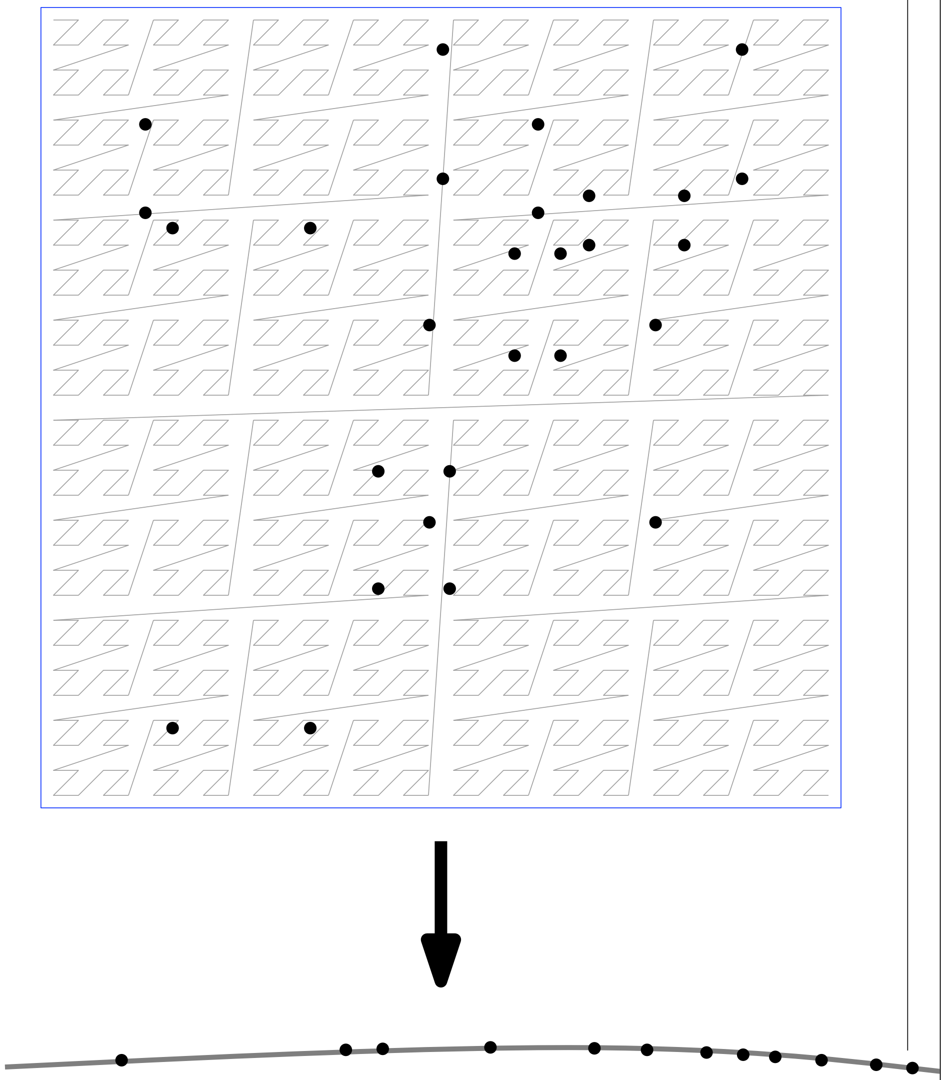
1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order

How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.

Algorithm:

1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order

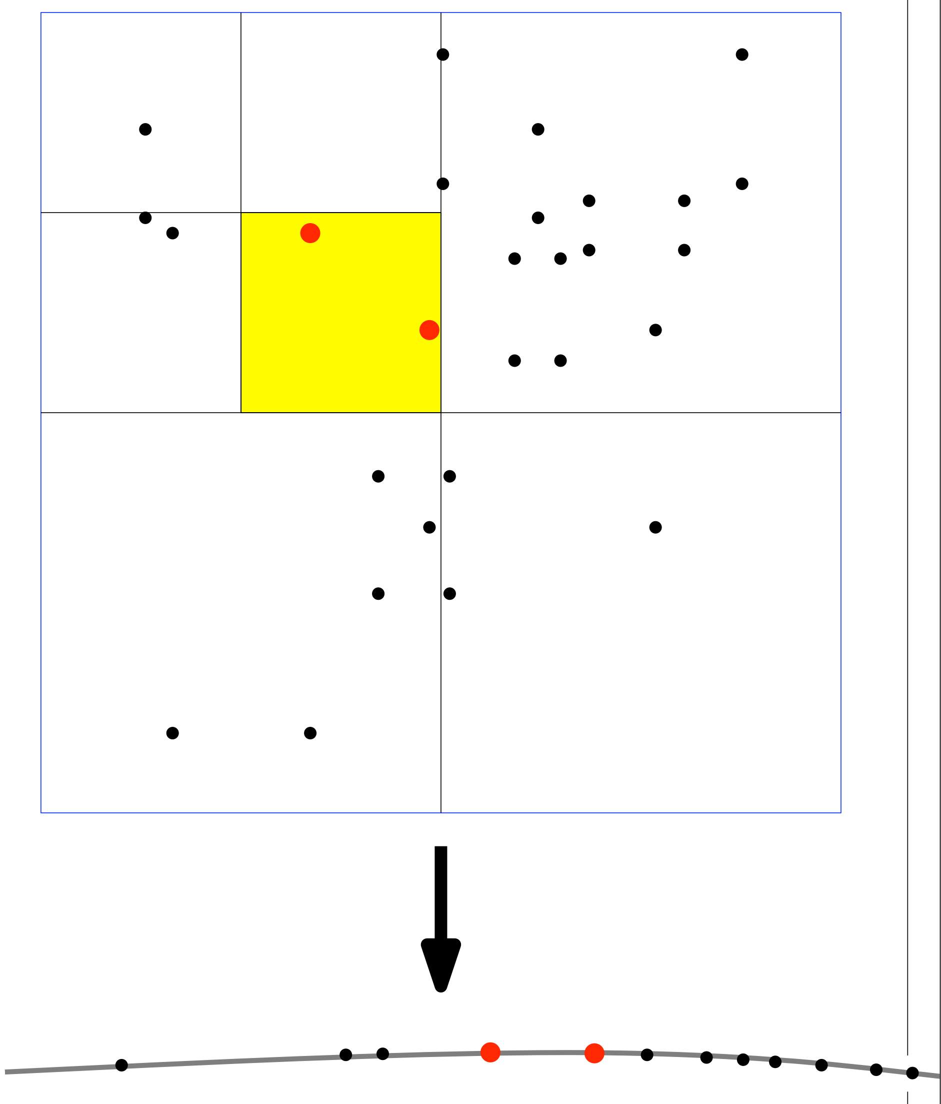


How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.

Algorithm:

1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order
2. For $i \leftarrow 1$ to m:
   - find smallest cell Q that contains L_i and L_{i+1};

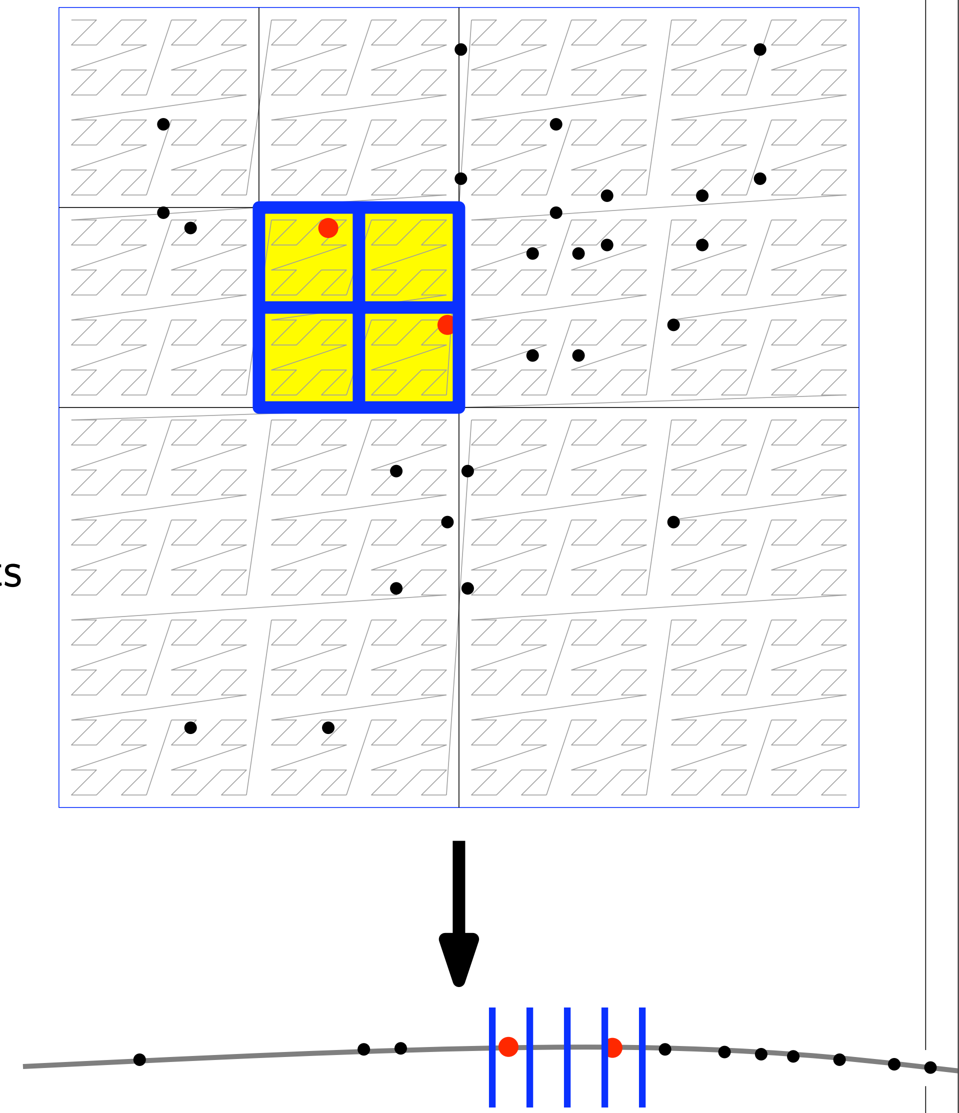


How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.

Algorithm:

1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order
2. For $i \leftarrow 1$ to m:
   - find smallest cell Q that contains L_i and L_{i+1};
   - output cell boundaries of Q and its subquadrants

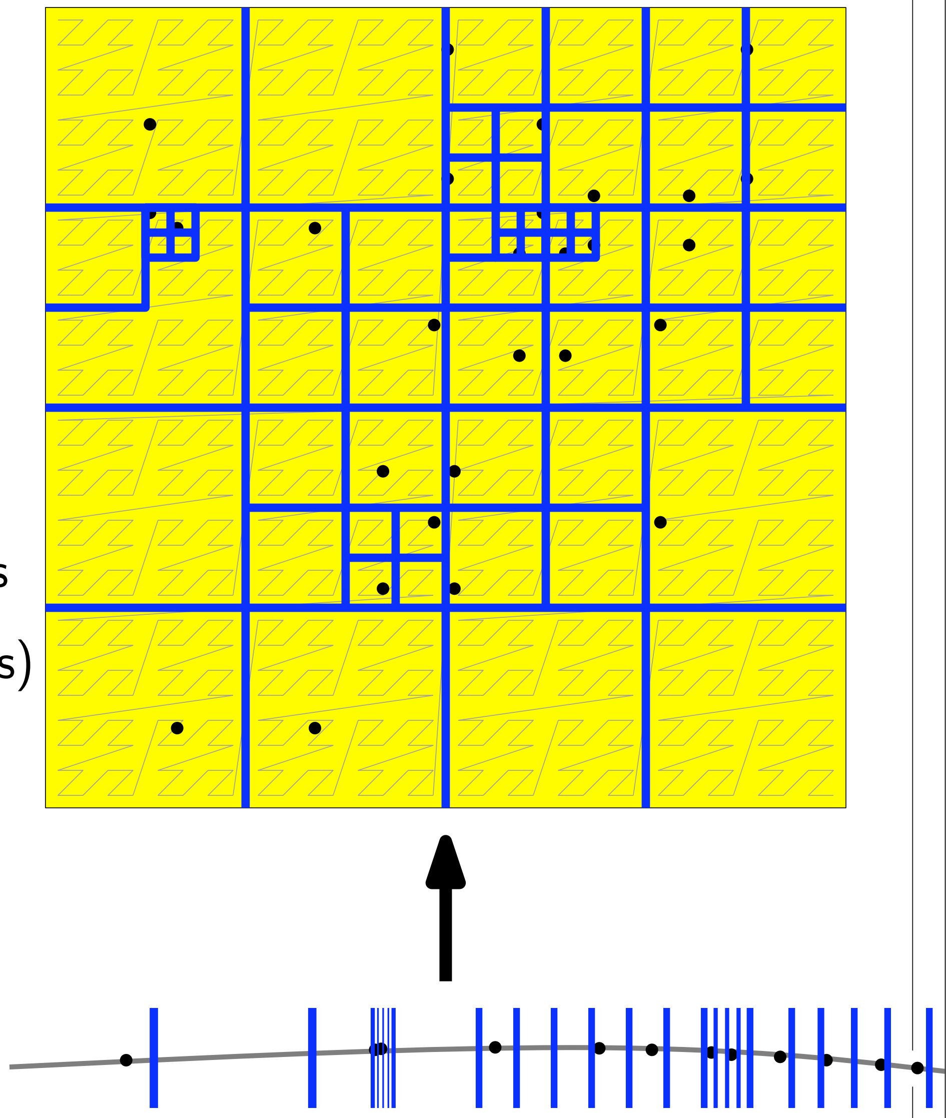


How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.

Algorithm:

1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order
2. For $i \leftarrow 1$ to m:
   - find smallest cell Q that contains L_i and L_{i+1};
   - output cell boundaries of Q and its subquadrants
3. Sort cell boundaries in Z-order (removing duplicates)

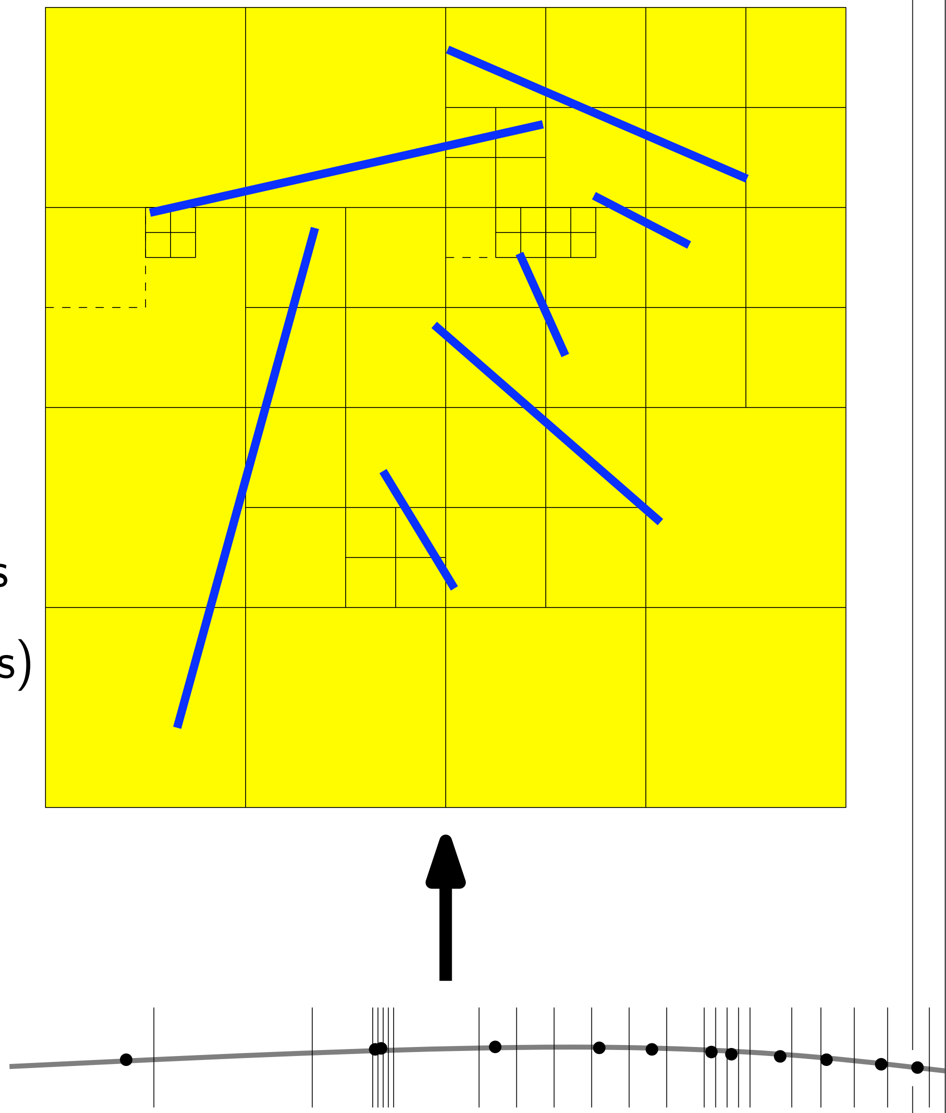


How to get that quadtree in Z-order (for line segments in unit square)

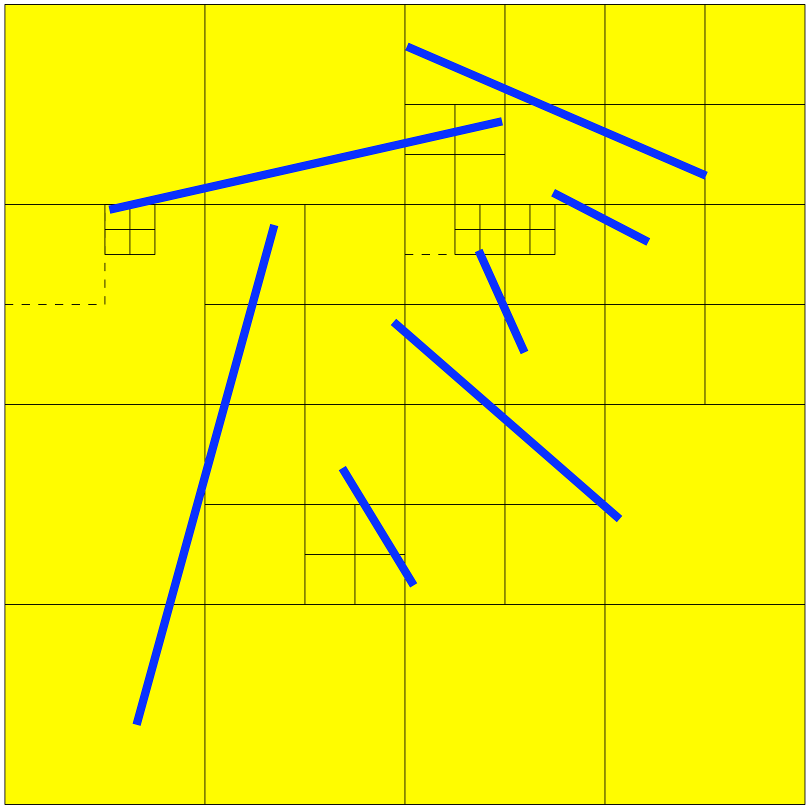
Input: file with for each line segment its endpoints.

Algorithm:

1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order
2. For $i \leftarrow 1$ to m:
   - find smallest cell Q that contains L_i and L_{i+1};
   - output cell boundaries of Q and its subquadrants
3. Sort cell boundaries in Z-order (removing duplicates)
4. Put line segments in cells



How to get that quadtree in Z-order (for line segments in unit square)



Input: file with for each line segment its endpoints.

Algorithm:

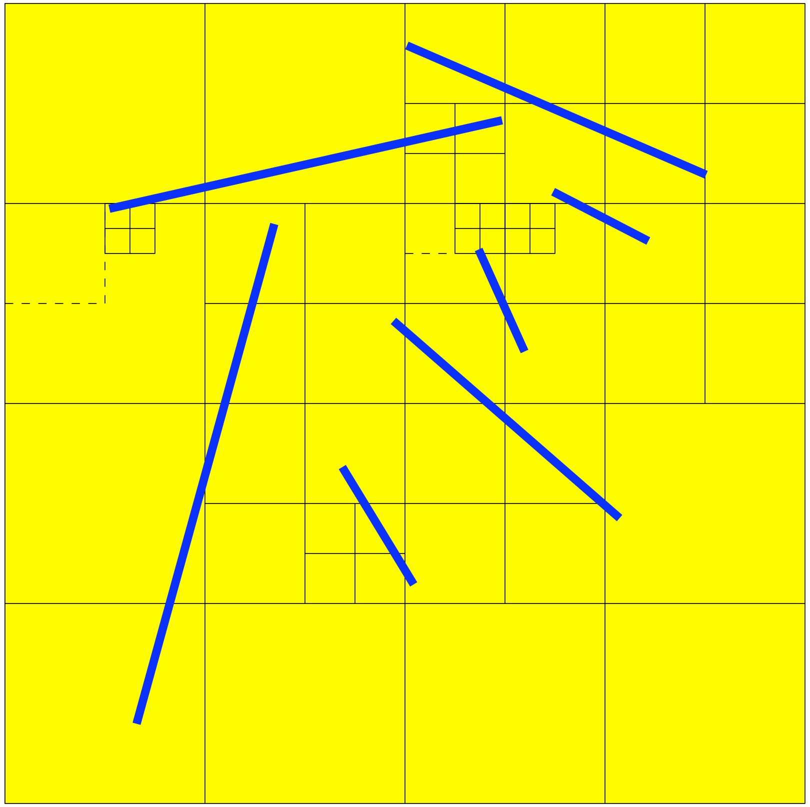
1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order
2. For $i \leftarrow 1$ to m:
   - find smallest cell Q that contains L_i and L_{i+1};
   - output cell boundaries of Q and its subquadrants
3. Sort cell boundaries in Z-order (removing duplicates)
4. Put line segments in cells

To prove for input of n line segments:

- together cell boundaries form quadtree subdivision of unit square;
- $O(1)$ line segments per cell;
- $O(n)$ cells in total;
- algorithm runs in $O(sort(n))$ I/O's

How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.



Algorithm:

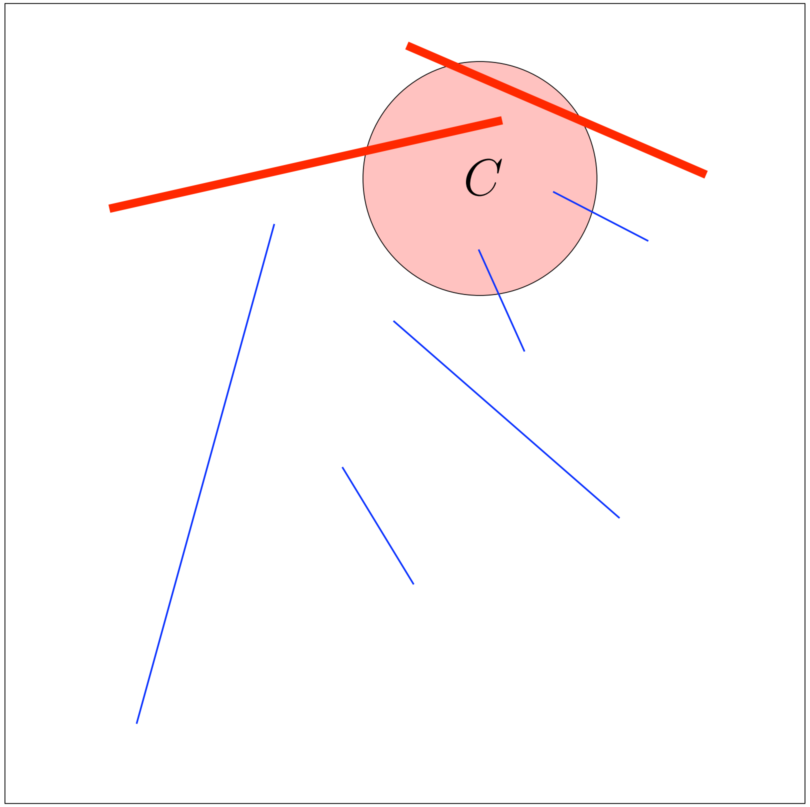
1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order
2. For $i \leftarrow 1$ to m:
   - find smallest cell Q that contains L_i and L_{i+1};
   - output cell boundaries of Q and its subquadrants
3. Sort cell boundaries in Z-order (removing duplicates)
4. Put line segments in cells

To prove for input of n line segments:

- together cell boundaries form (compressed) quadtree subdivision of unit square;
- $O(1)$ line segments per cell;
- $O(n)$ cells in total;
- algorithm runs in $O(sort(n))$ I/O's

How to get that quadtree in Z-order (for line segments in unit square)

Input: file with for each line segment its endpoints.



Algorithm:

1. Sort bounding box vertices of line segments into list $L = \{L_1, ..., L_m\}$ in Z-order
2. For $i \leftarrow 1$ to m:
   - find smallest cell Q that contains L_i and L_{i+1};
   - output cell boundaries of Q and its subquadrants
3. Sort cell boundaries in Z-order (removing duplicates)
4. Put line segments in cells

To prove for input of n line segments:

- together cell boundaries form (compressed) quadtree subdivision of unit square;
- $O(1)$ line segments per cell;
- $O(n)$ cells in total;
- algorithm runs in $O(sort(n))$ I/O's

Works if line segments have *low density*:
for every circle C of diam d,
#line segments longer than d that intersect C
is at most a constant independent of n

Quadtrees for Low-Density Subdivisions

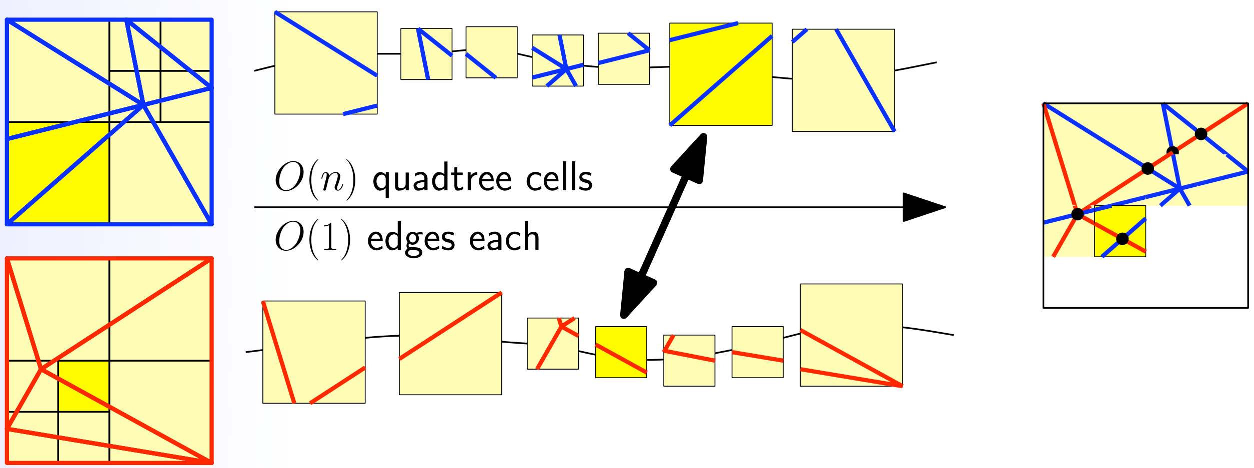
- Theorem: Let F be a subdivision of the unit square with n edges and density λ.

  A quadtree constructed on the bounding-box vertices of the edges with the following stopping rule:

  Stop splitting when the cell contains at most one vertex.

- has O(n) cells.
- each cell is intersected by $O(\lambda)$ edges.
- the total number of intersections is $O(n\lambda)$
- can be constructed in $O(\mathrm{sort}(n\lambda))$ IOs.
- all pairs of intersections can be found in $O(\mathrm{scan}(n\lambda))$

I/O-Efficient Indices for Fat Triangulations and Low-Density Subdivisions



n = input size; M = main memory size; B = disk block size; $scan(n) < sort(n) << n$

For low-density triangulations / sets of line segments*, there is a data structure that supports:

- map overlay in $O(scan(n))$ I/O's;
- range queries in $O(\frac{1}{\varepsilon}(\log_B n) + scan(k_\varepsilon))$ I/O's.
- point location in $O(\log_B n)$ I/O's;
- (triangulations only) updates in $O(\log_B n)$ I/O's;

The data structures are built with $O(sort(n))$ I/Os.

*) for any circle C, number of intersecting segments bigger than $\mathrm{diam}(C)$ is at most a constant

Discussion

- d-fat triangulations ← Much simpler
  - $O(n/d)$ cells
  - each cell intersects $O(1/d)$ edges
  - total $O(n/d^2)$ edge-cell intersections ← $O(n/d)$?
  - construction: $O(sort(n/d^2))$ IOs

- set of edges of density λ ← Better dependency on parameters
  - $O(n)$ cells
  - each cell intersects $O(\lambda)$ edges
  - total $O(\lambda n)$ edge-cell intersections
  - construction: $O(sort(\lambda n))$ IOs

- A d-fat triangulation has density $O(1/d)$
  - can use both approaches
  - More efficient?

Thank you