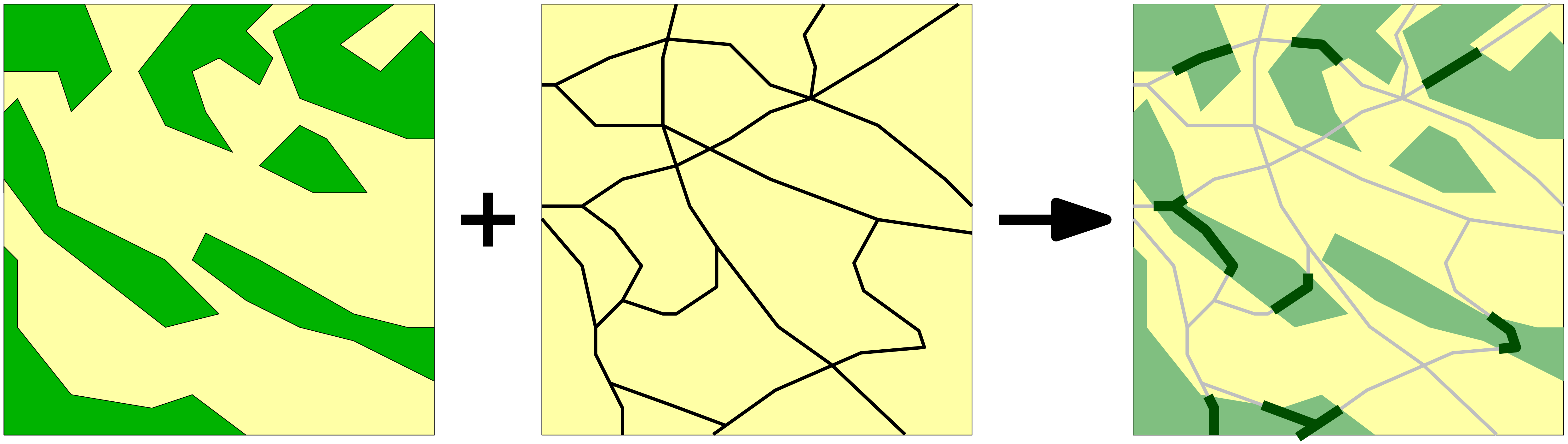
I/O-Efficient Map Overlay and Point Location on Low-Density Planar Maps

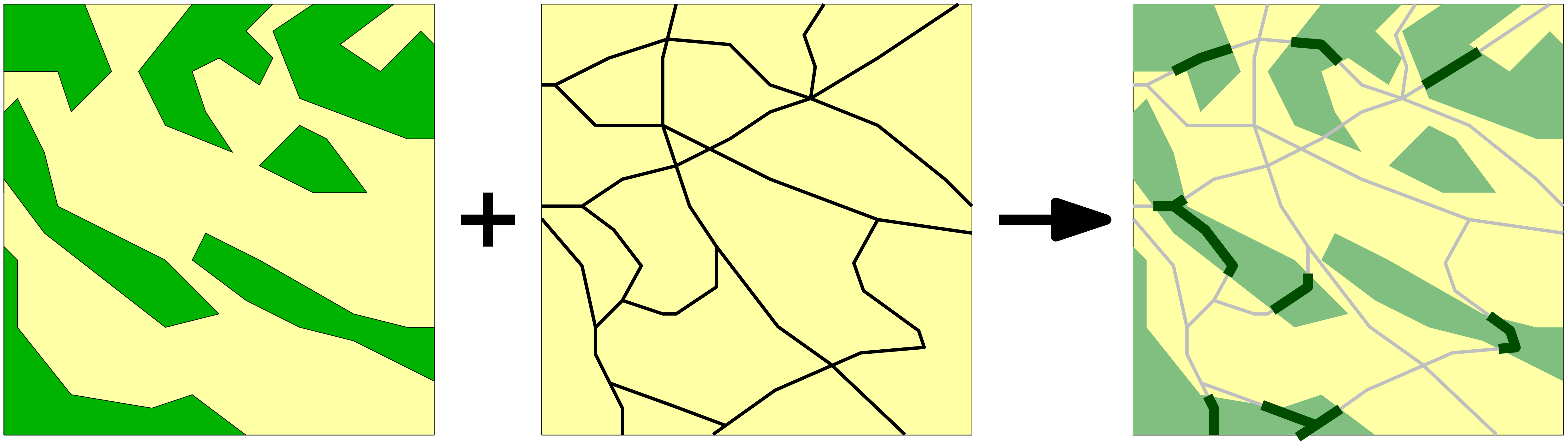
Mark de Berg   Herman Haverkort   Shripad Thite   Laura Toma



I/O-Efficient Map Overlay and Point Location on Low-Density Planar Maps

Mark de Berg   Herman Haverkort   Shripad Thite   Laura Toma

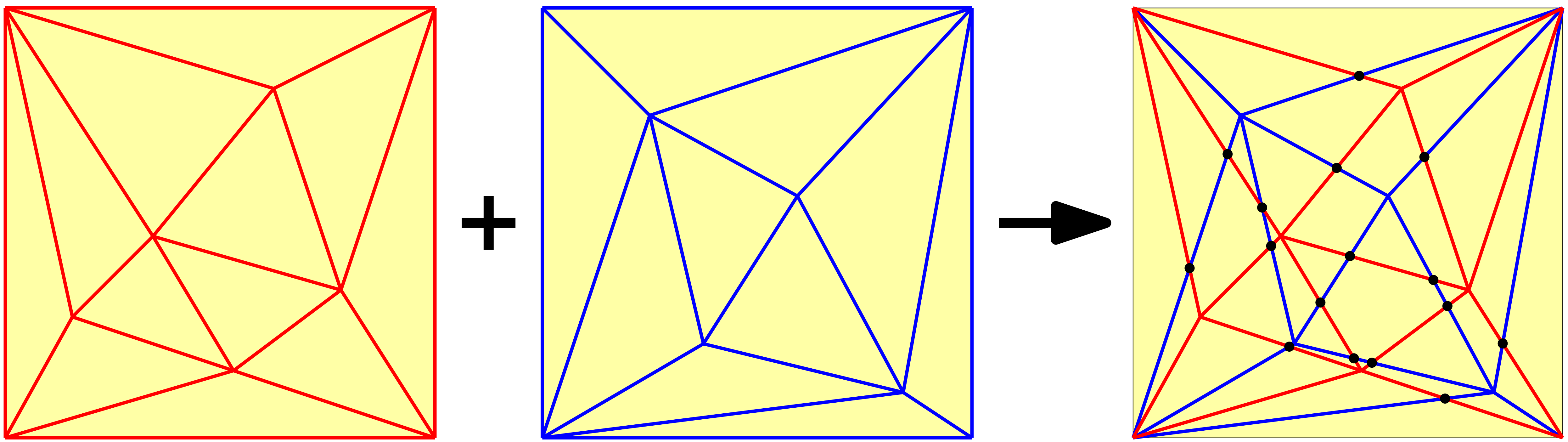
Maps: planar subdivisions, sets of (non-intersecting) line segments, ...



I/O-Efficient Map Overlay and Point Location on Low-Density Planar Maps

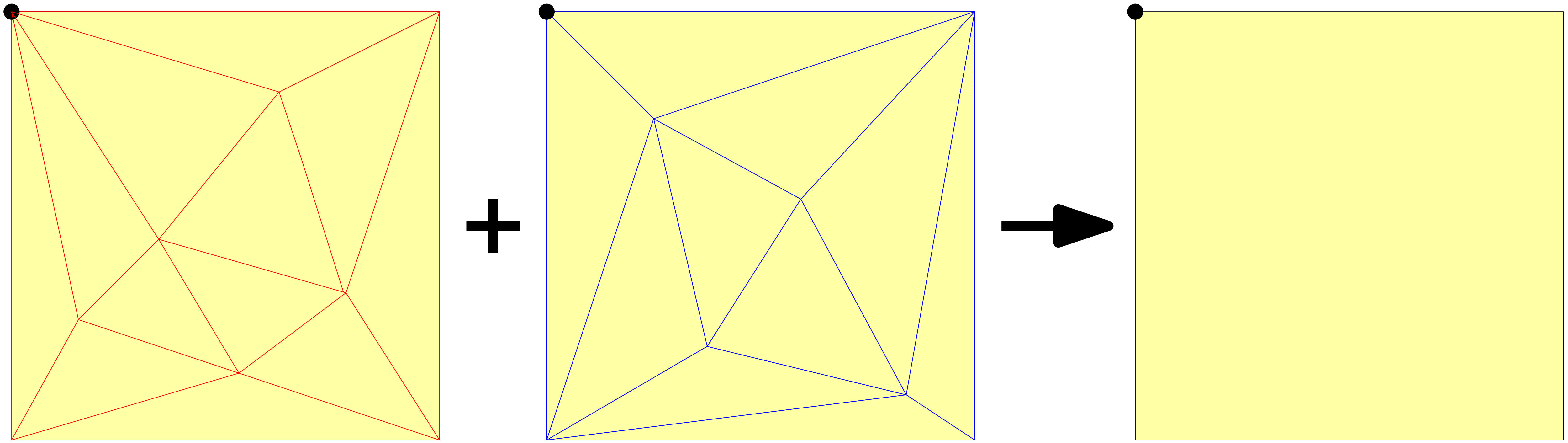
Mark de Berg   Herman Haverkort   Shripad Thite   Laura Toma

Maps: ..., triangulations



Overlaying triangulations CPU-efficiently

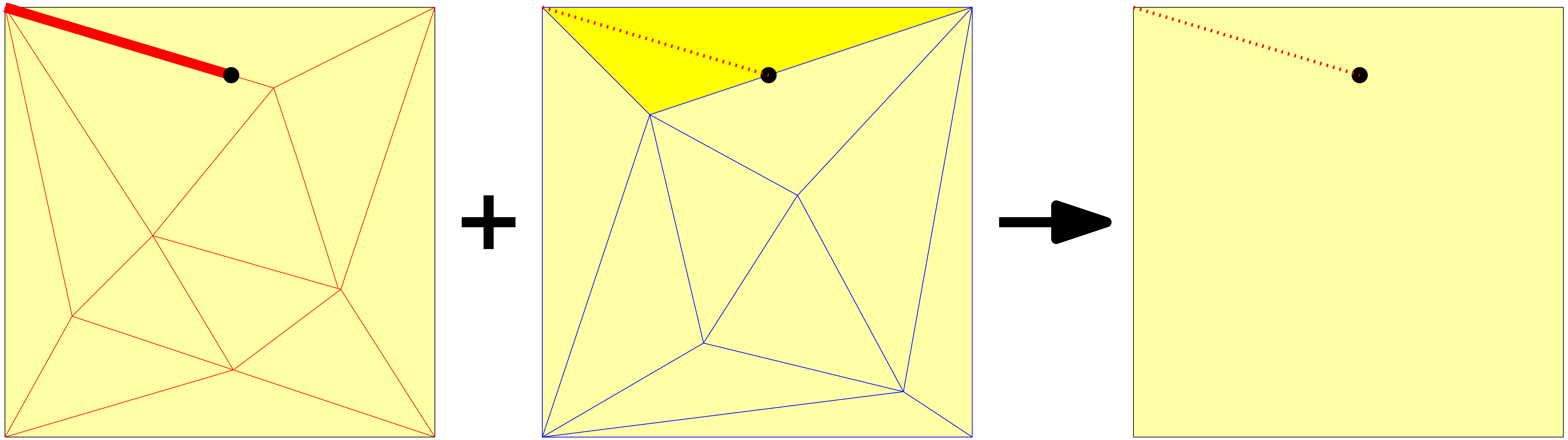
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

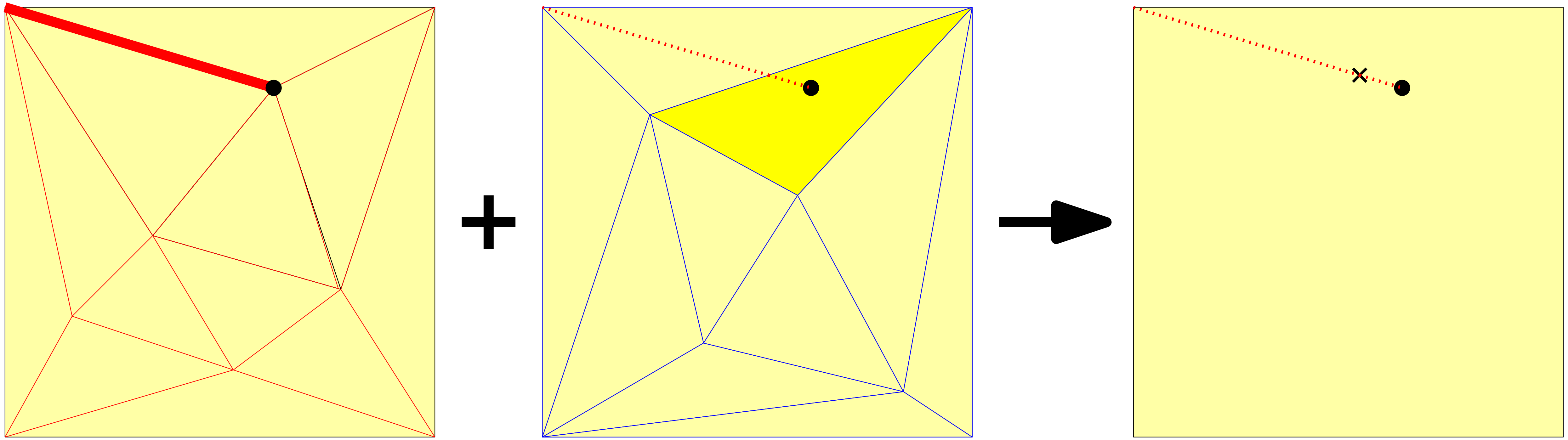
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

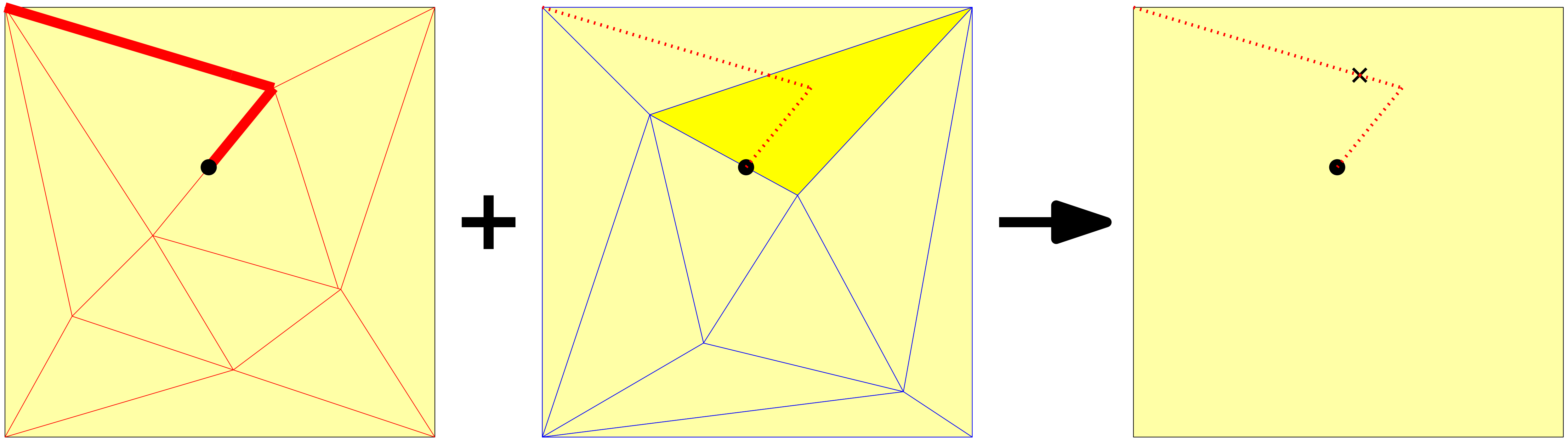
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

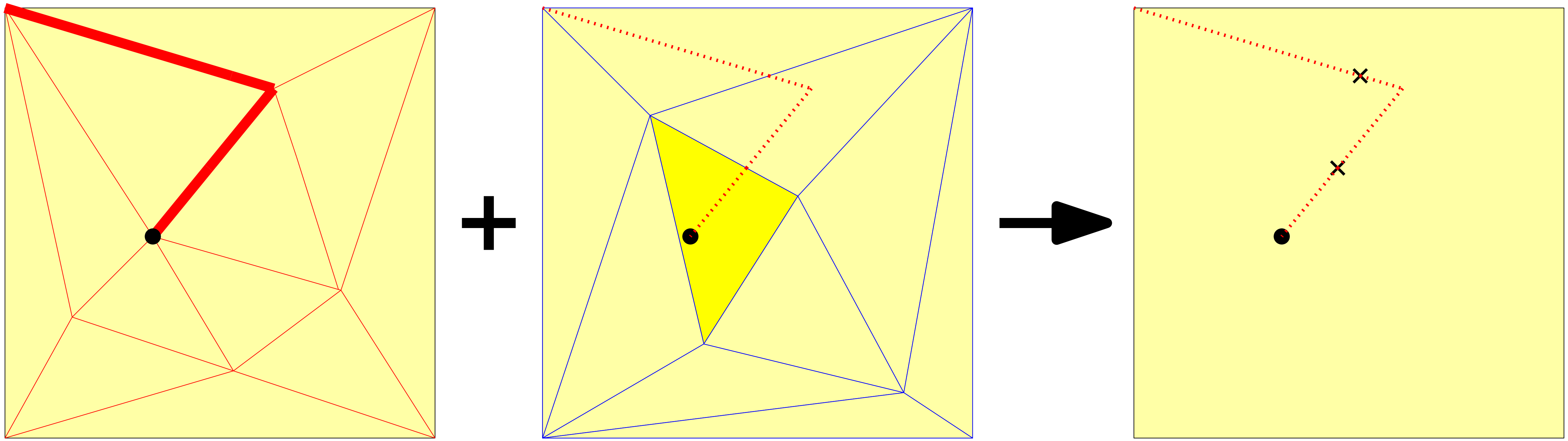
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

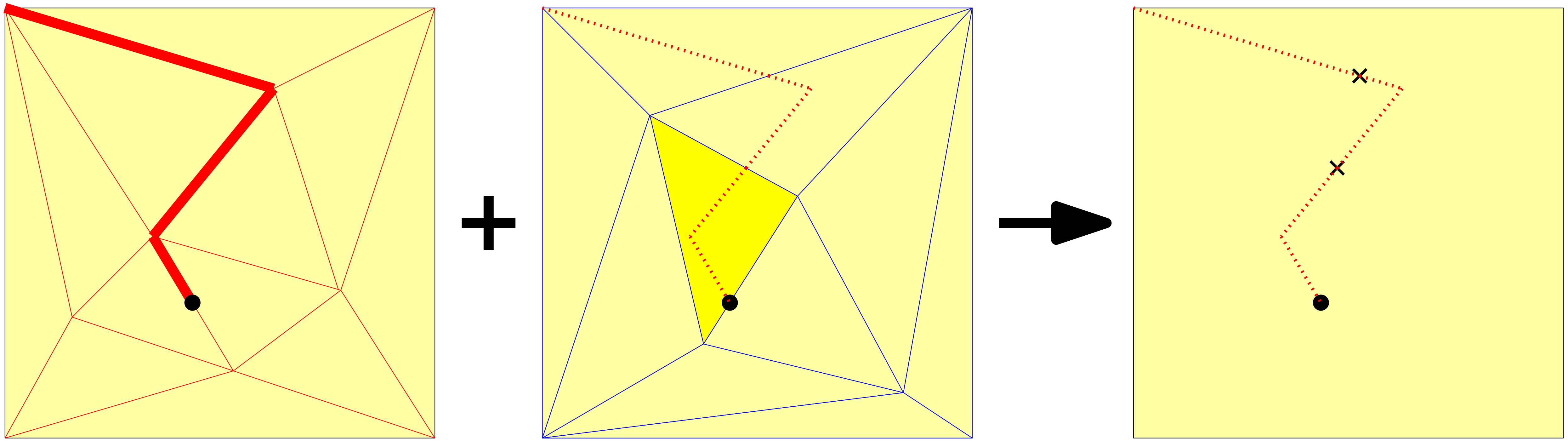
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

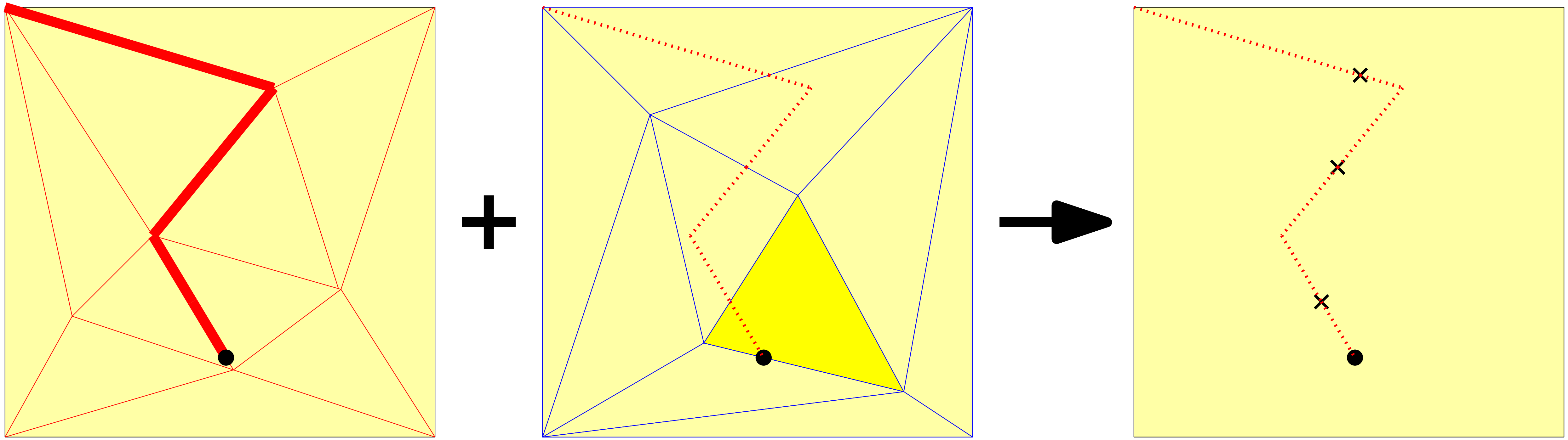
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

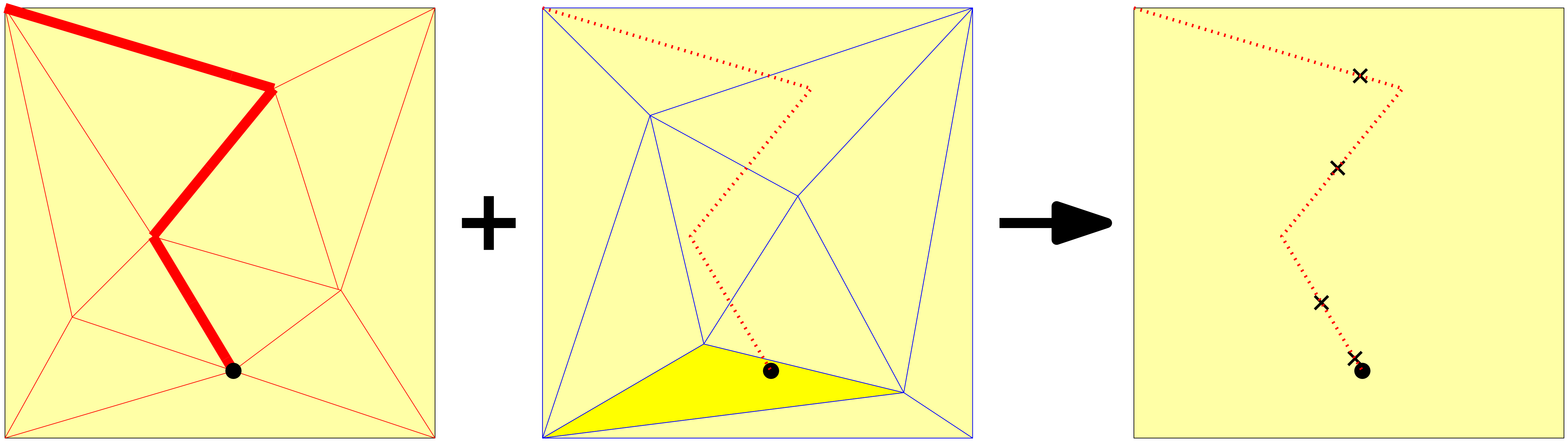
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

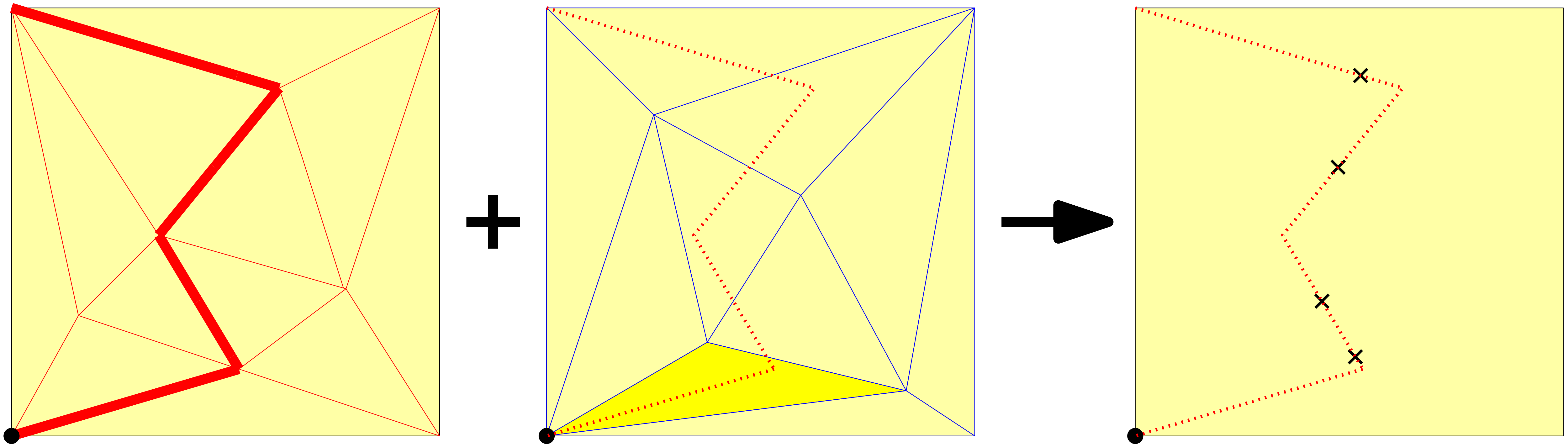
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

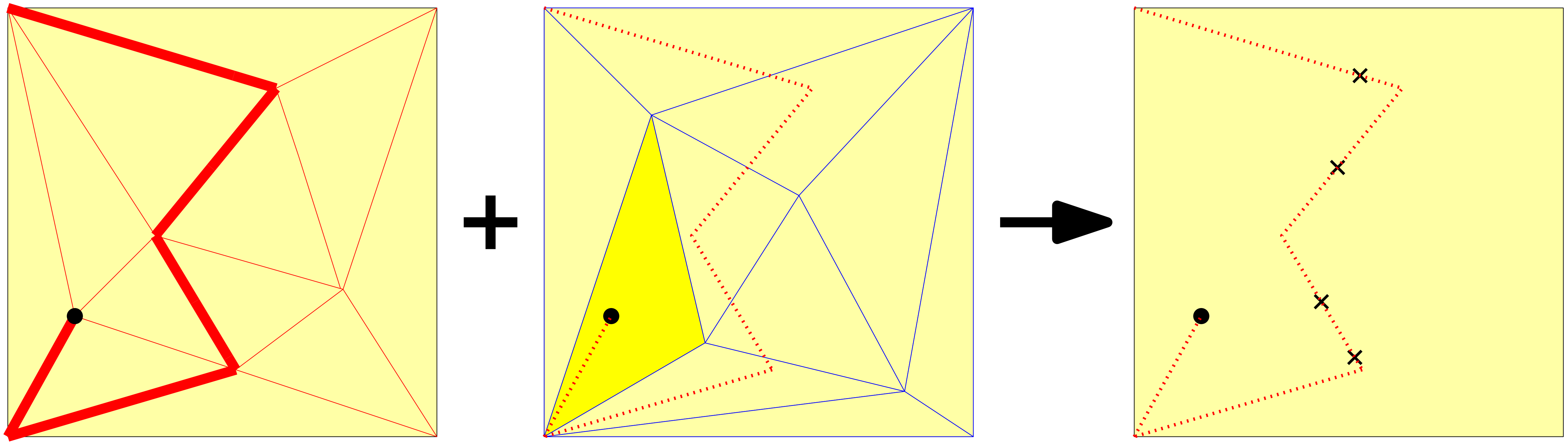
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

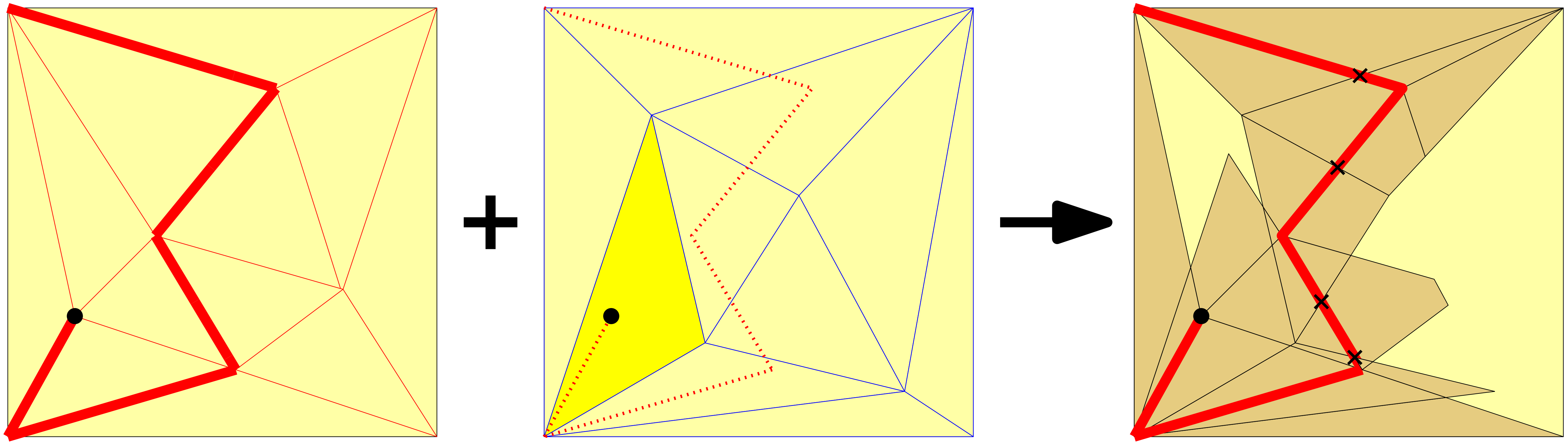
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

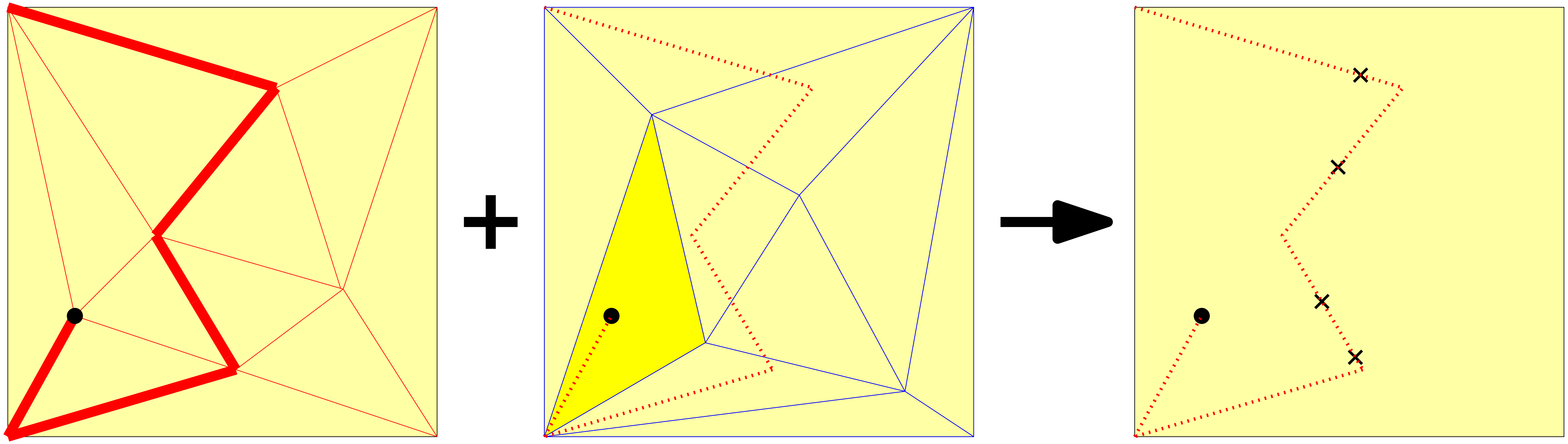
Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other

Overlaying triangulations CPU-efficiently

Maps: ..., triangulations



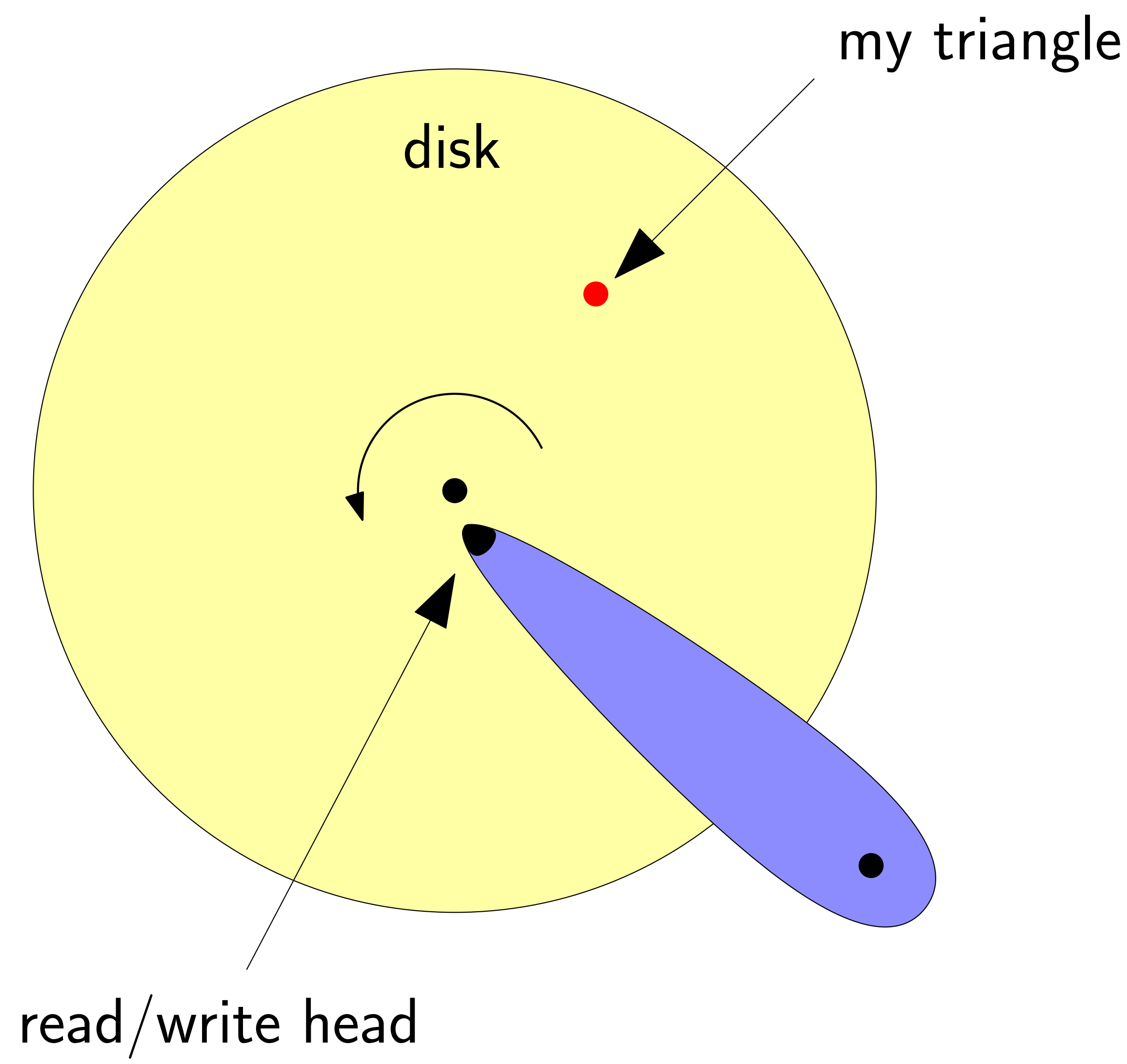
DFS in one triangulation, traverse triangles in the other:

- $\Theta(1)$ operations per edge
- $\Theta(1)$ operations per crossing

Total: $\Theta(n + k)$ CPU-operations (for n triangles, k intersections)

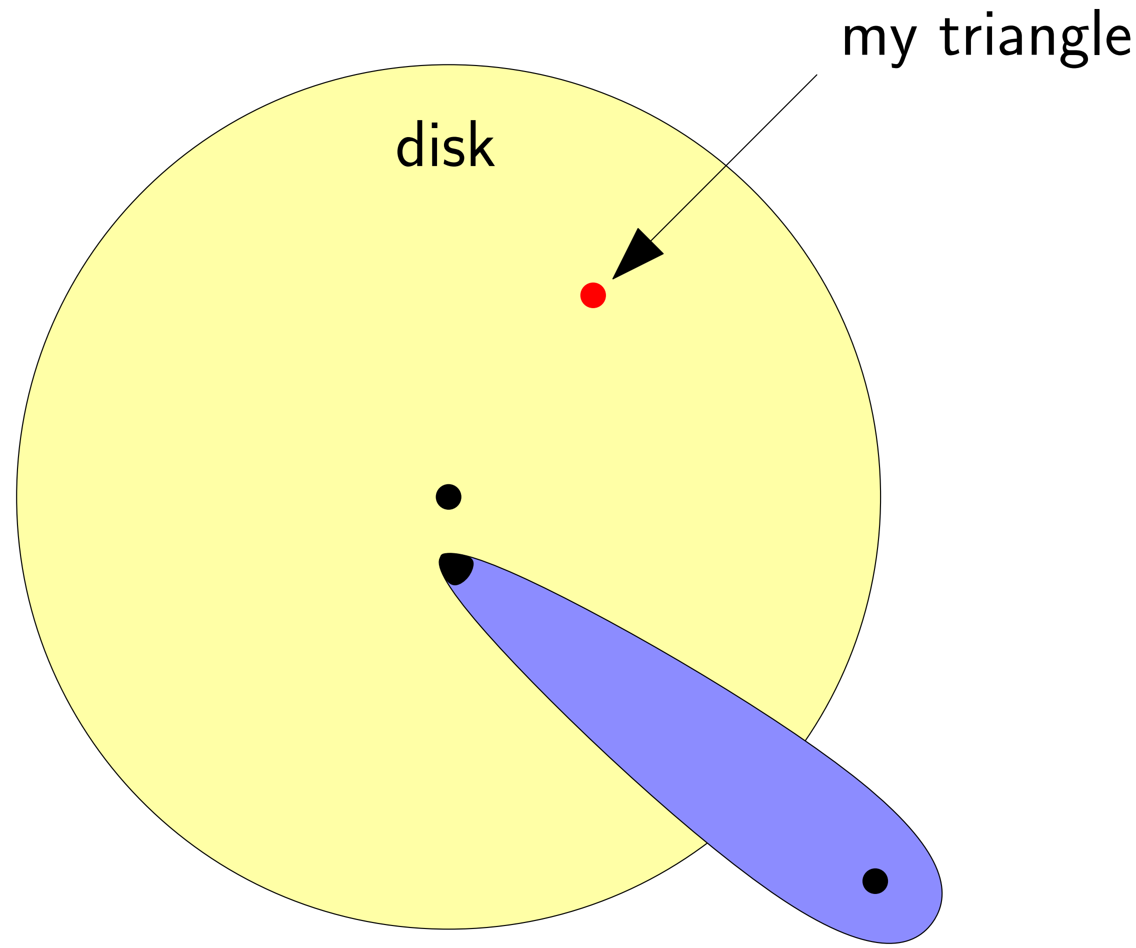
Using external memory

main memory
of size M
(too small for
all data)



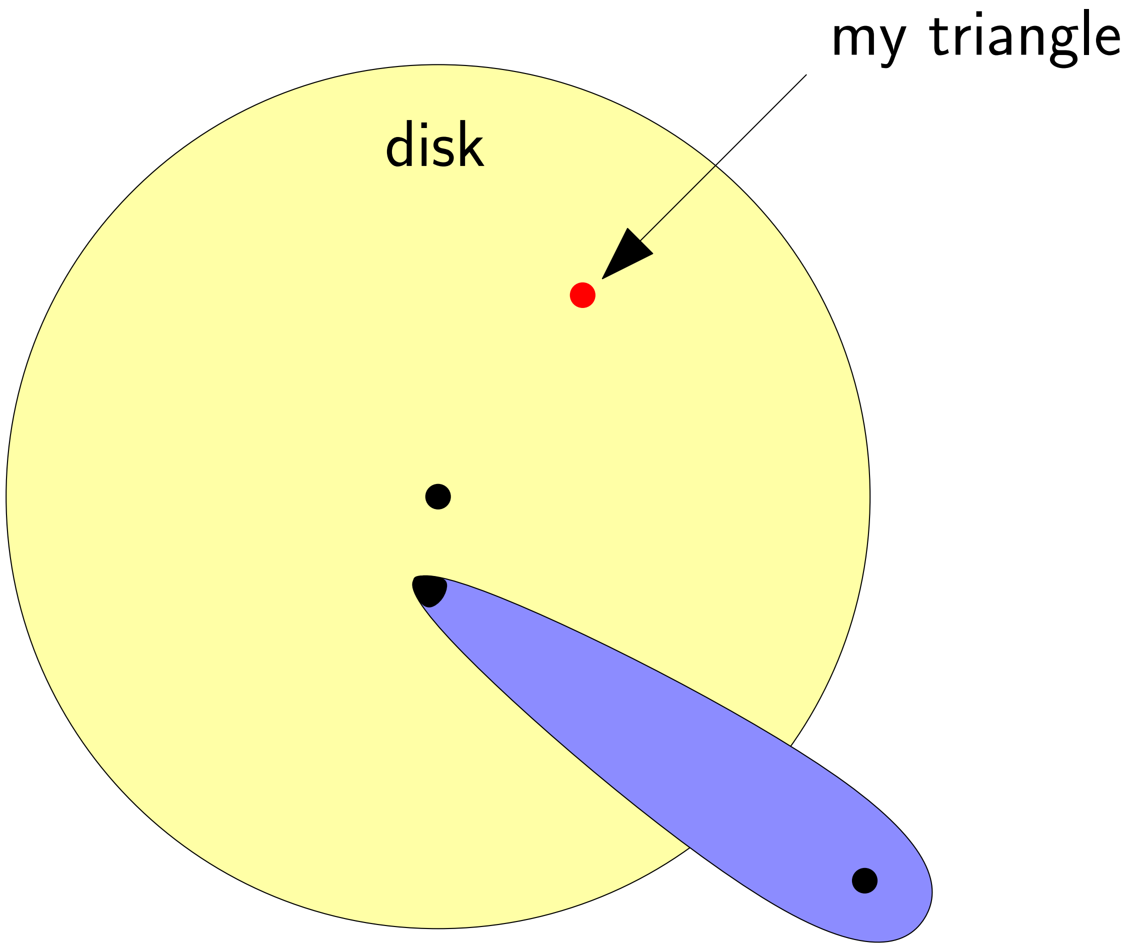
Using external memory

main memory
of size M
(too small for
all data)



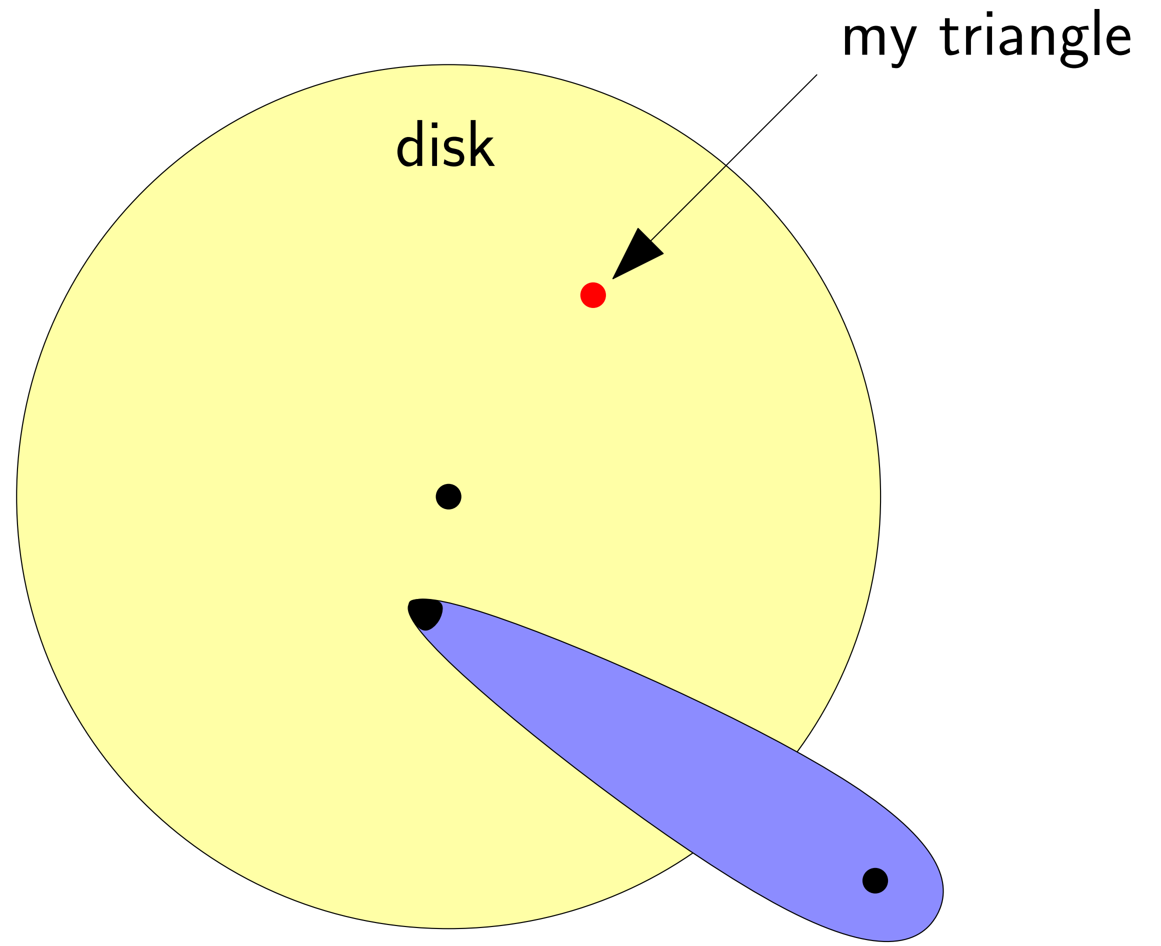
Using external memory

main memory
of size M
(too small for
all data)

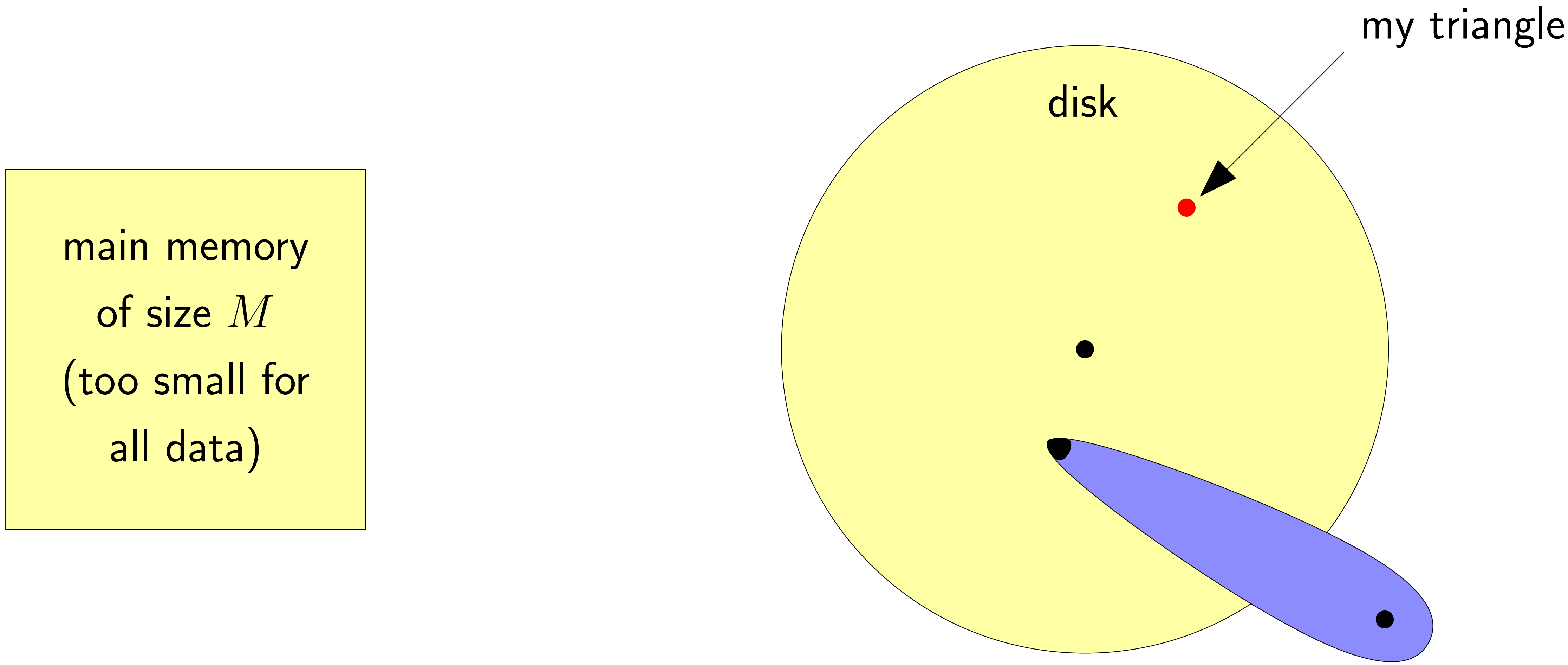


Using external memory

main memory of size M (too small for all data)

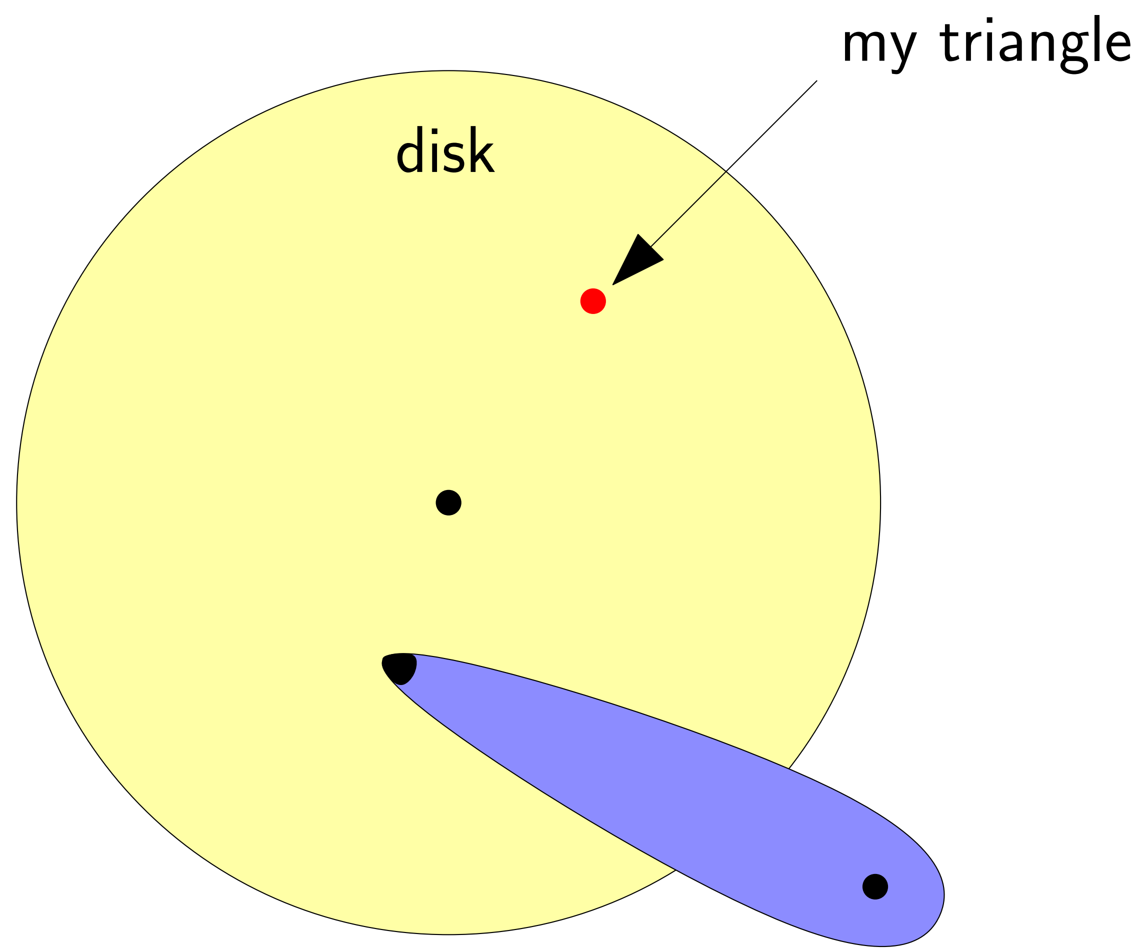


Using external memory

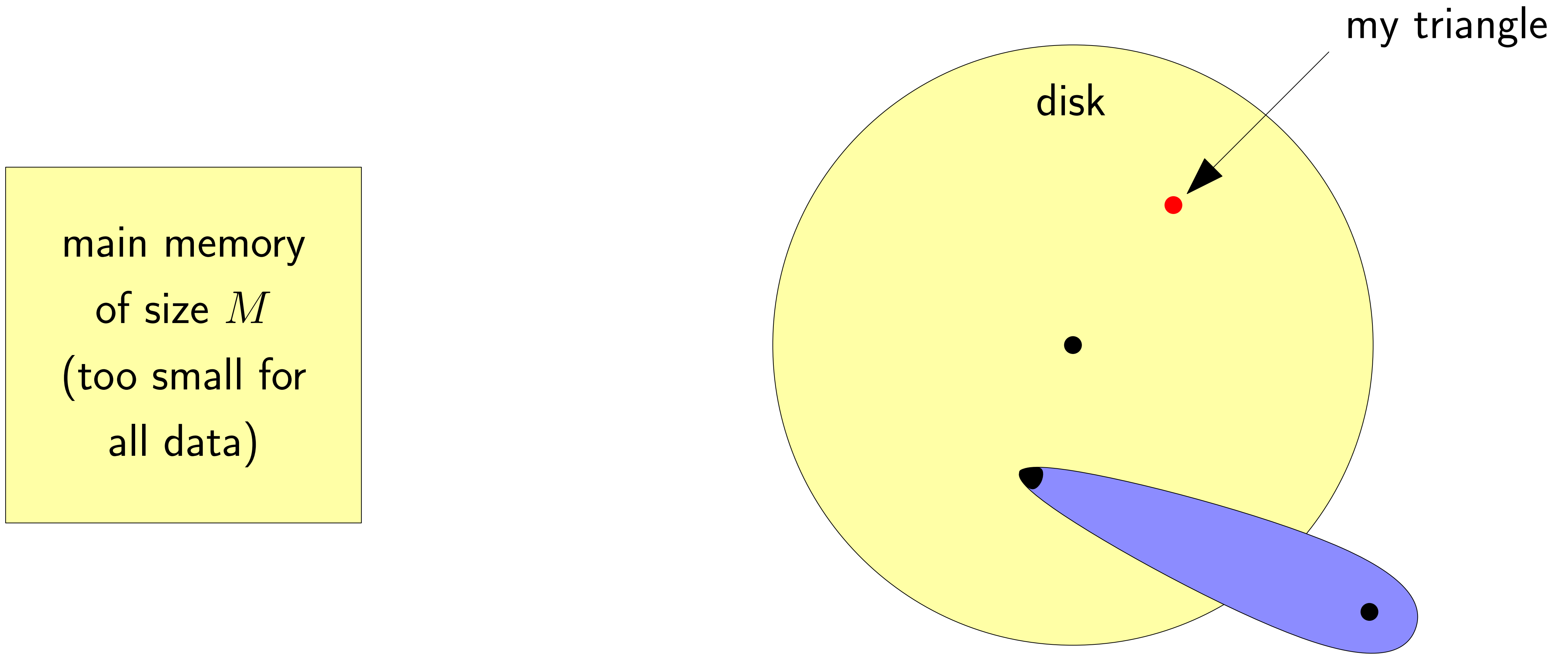


Using external memory

main memory
of size M
(too small for
all data)

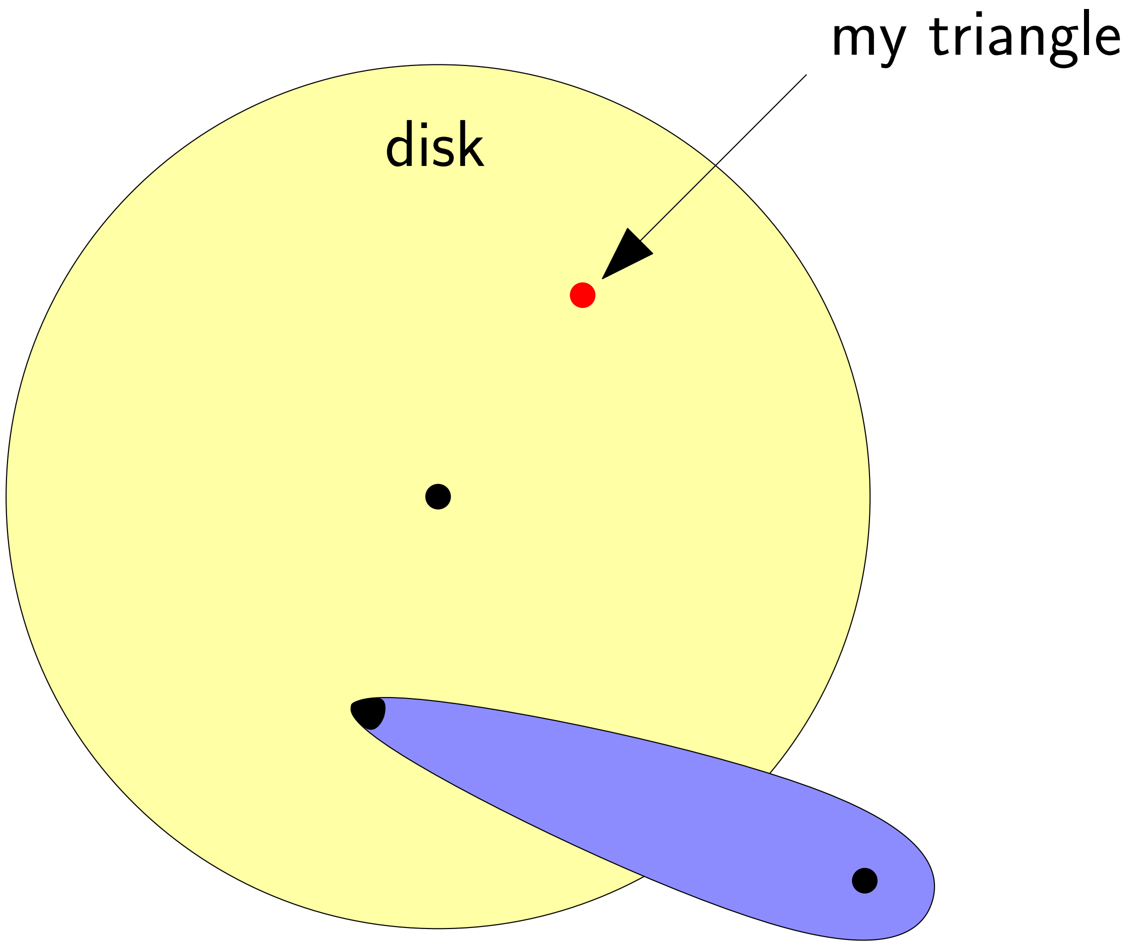


Using external memory



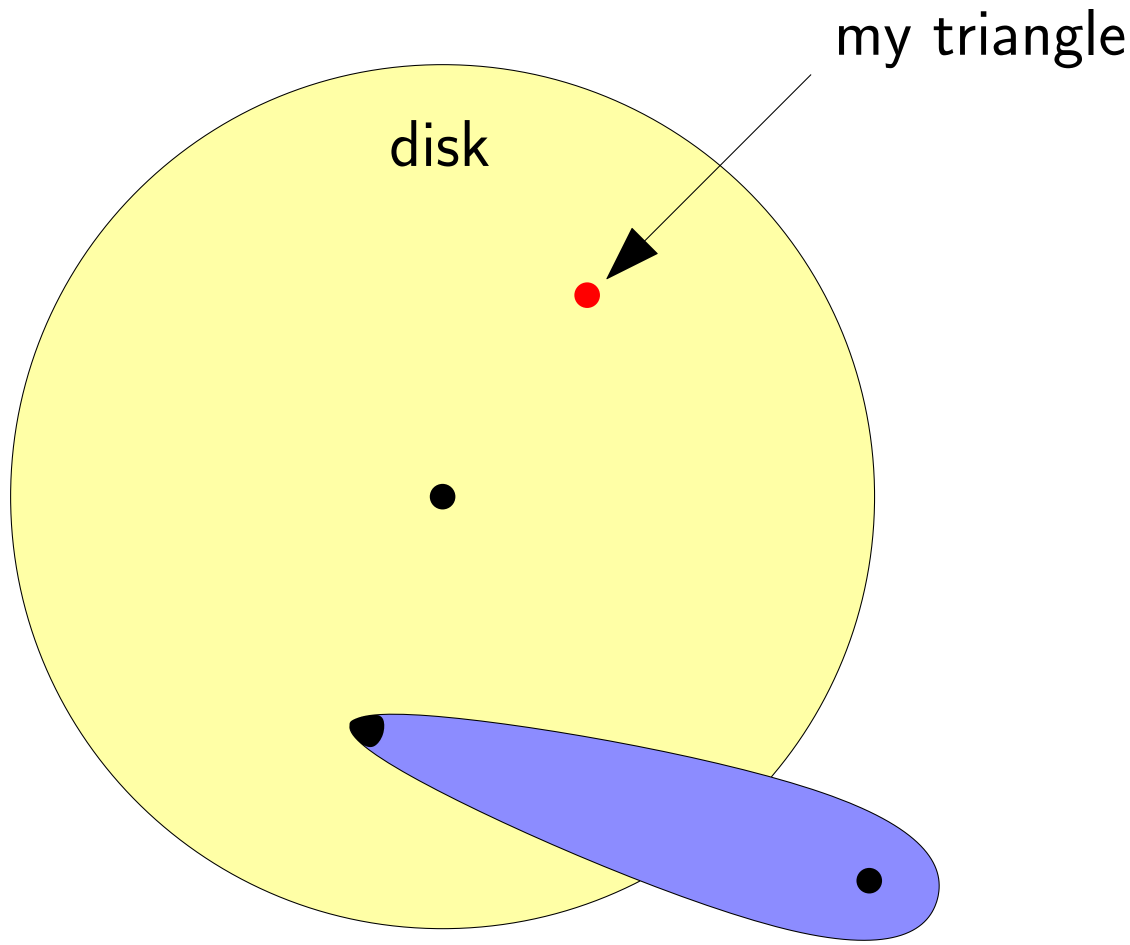
Using external memory

main memory
of size M
(too small for
all data)



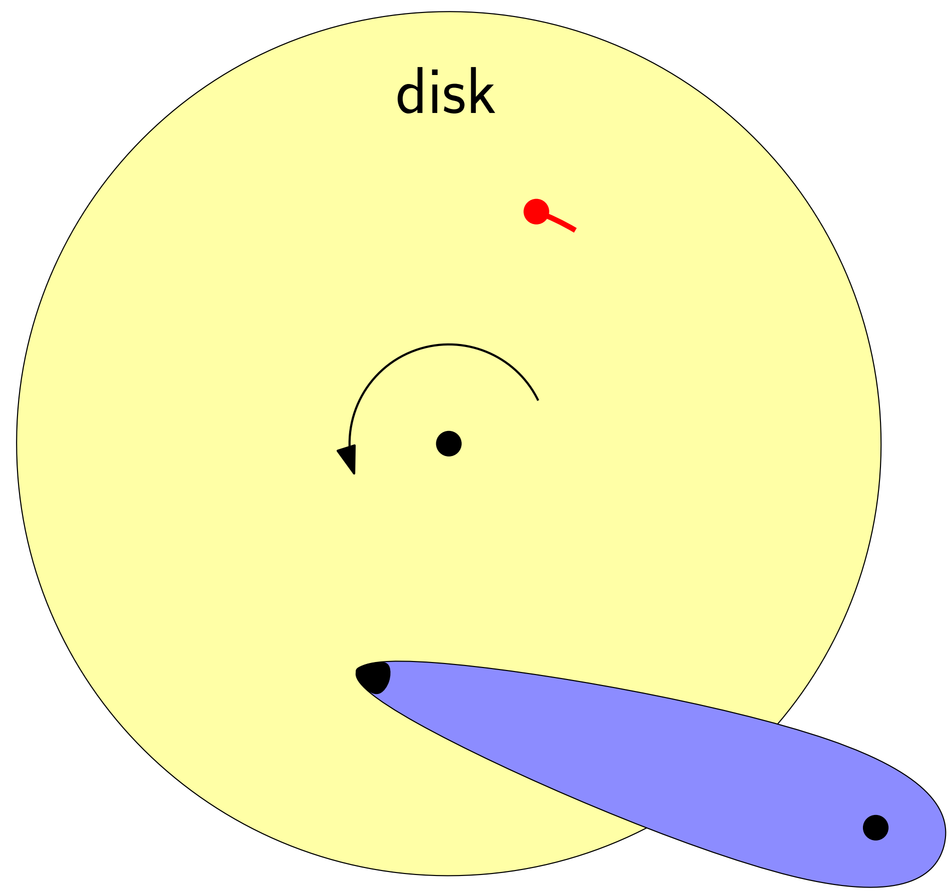
Using external memory

main memory of size M (too small for all data)



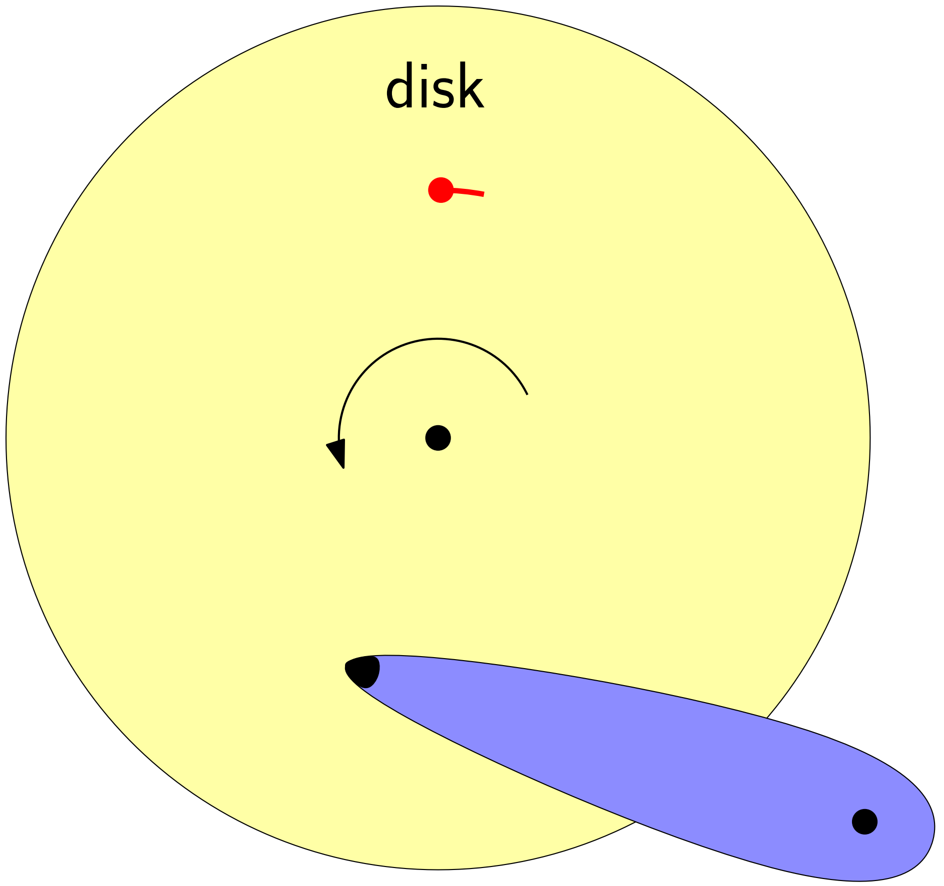
Using external memory

main memory of size M (too small for all data)



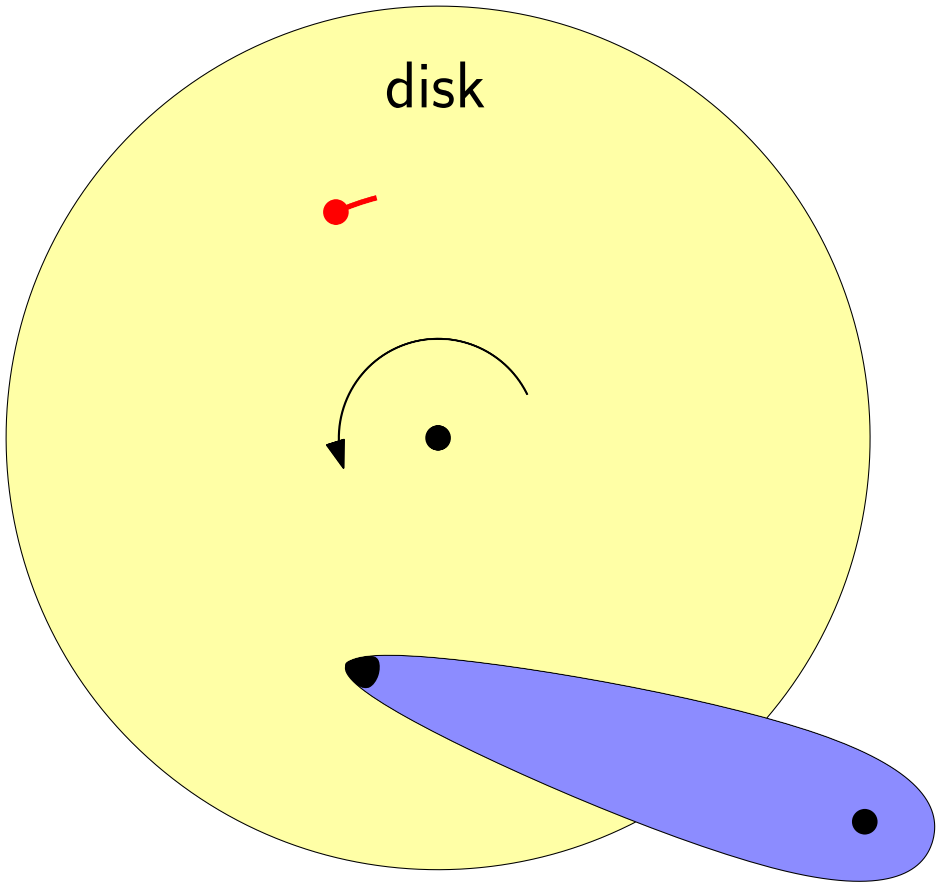
Using external memory

main memory
of size M
(too small for
all data)



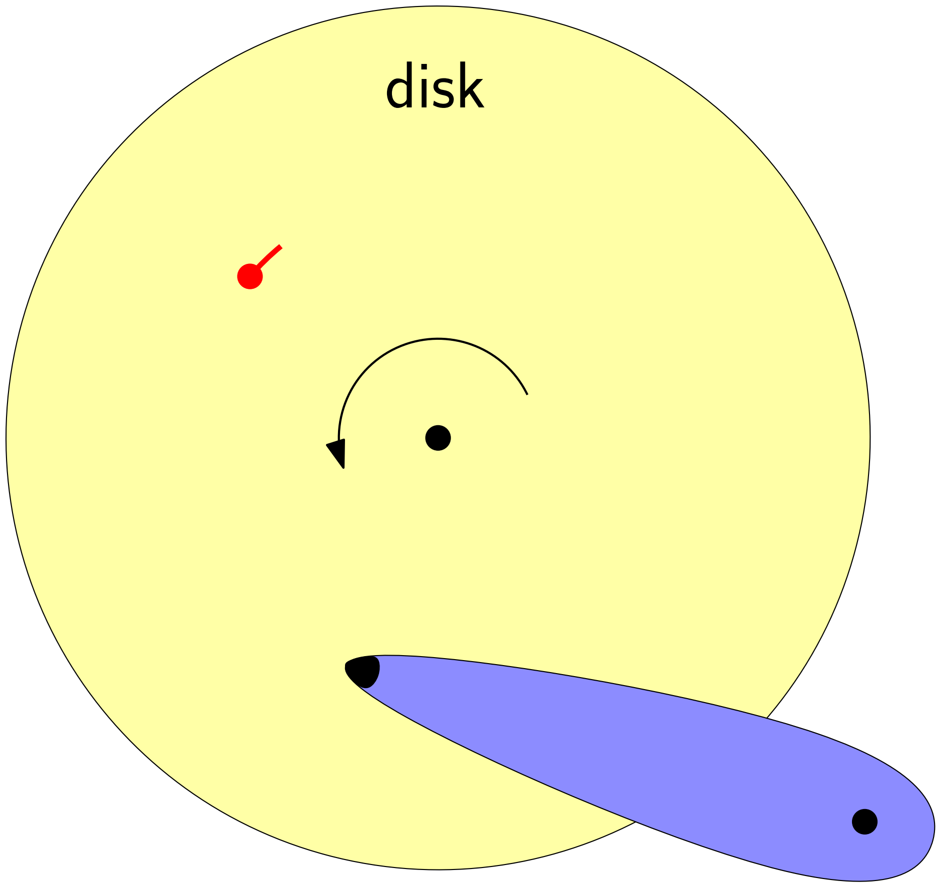
Using external memory

main memory
of size M
(too small for
all data)



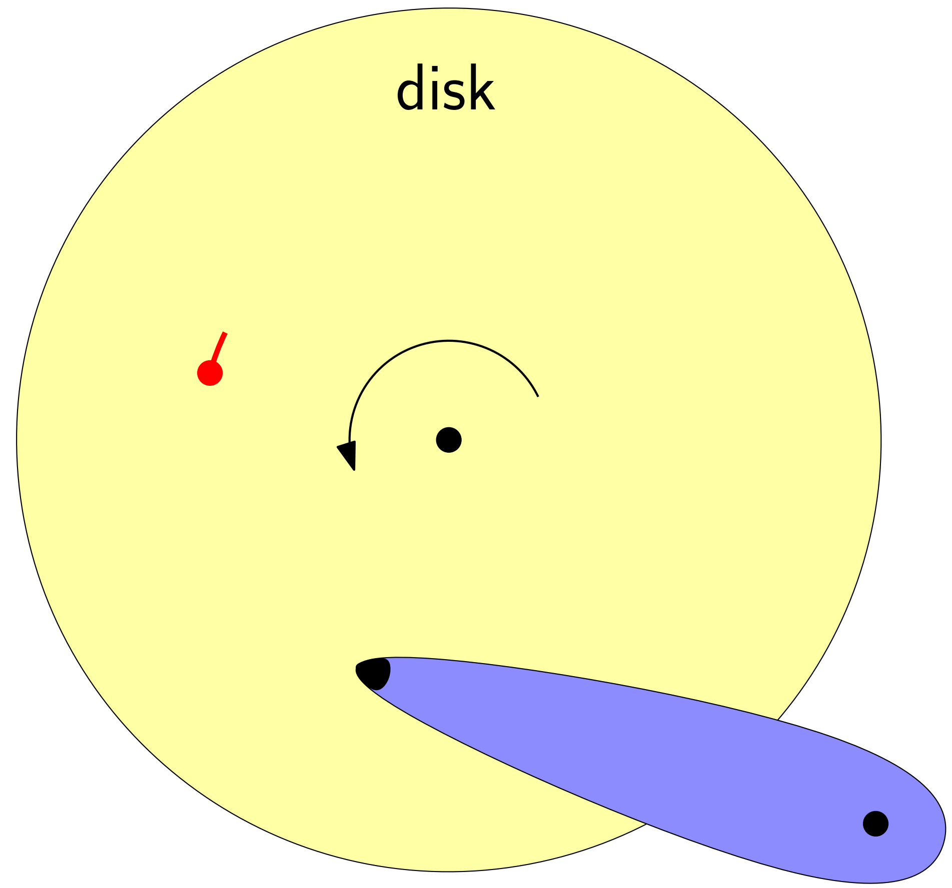
Using external memory

main memory
of size M
(too small for
all data)



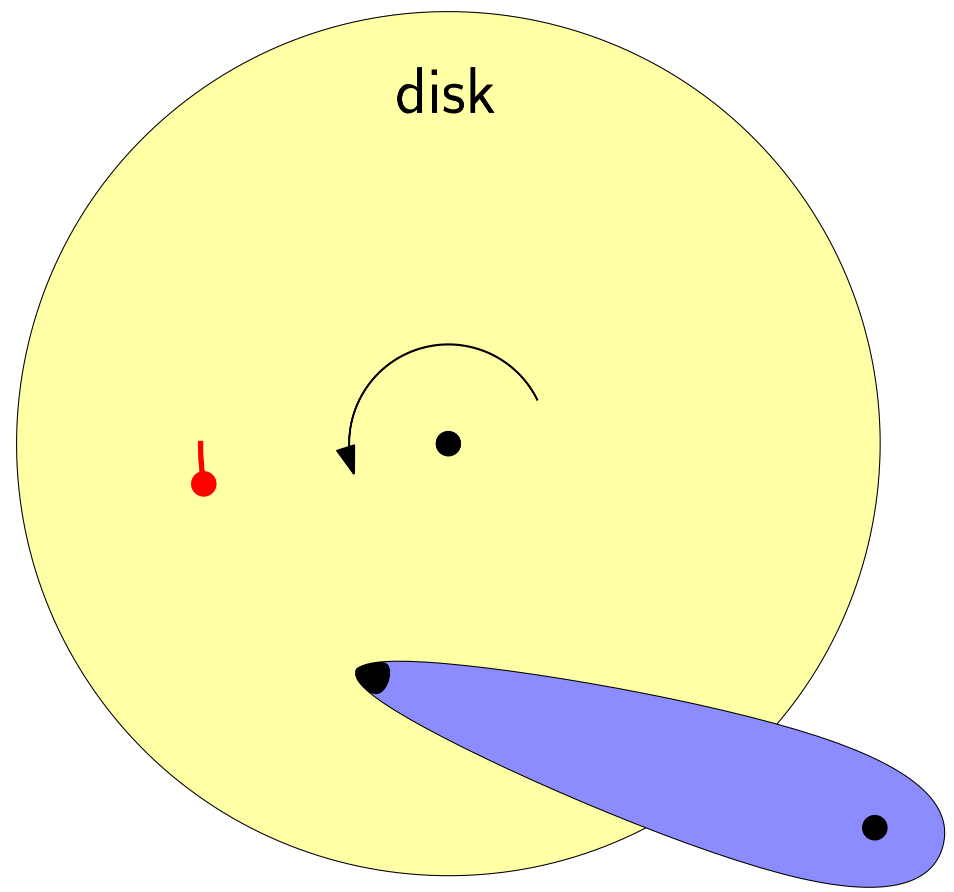
Using external memory

main memory
of size M
(too small for
all data)

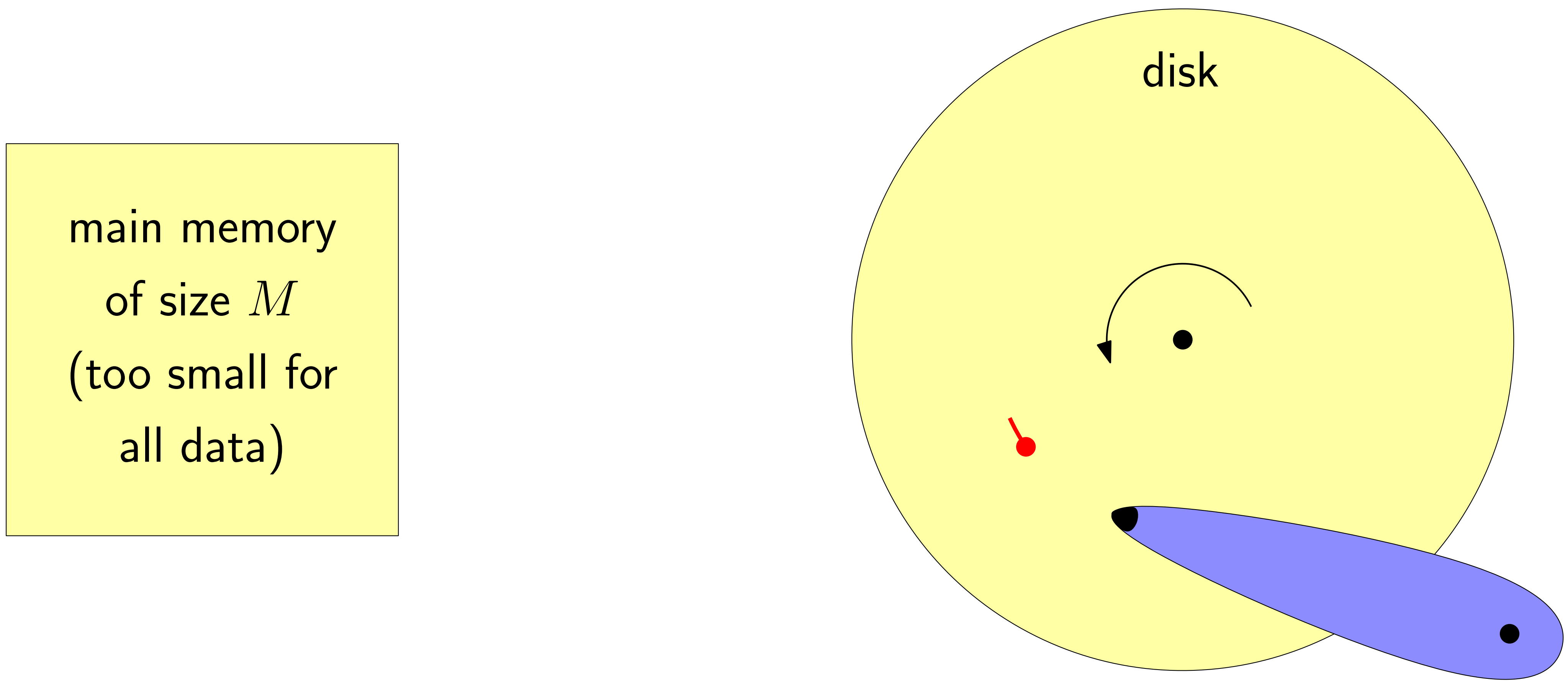


Using external memory

main memory
of size M
(too small for
all data)

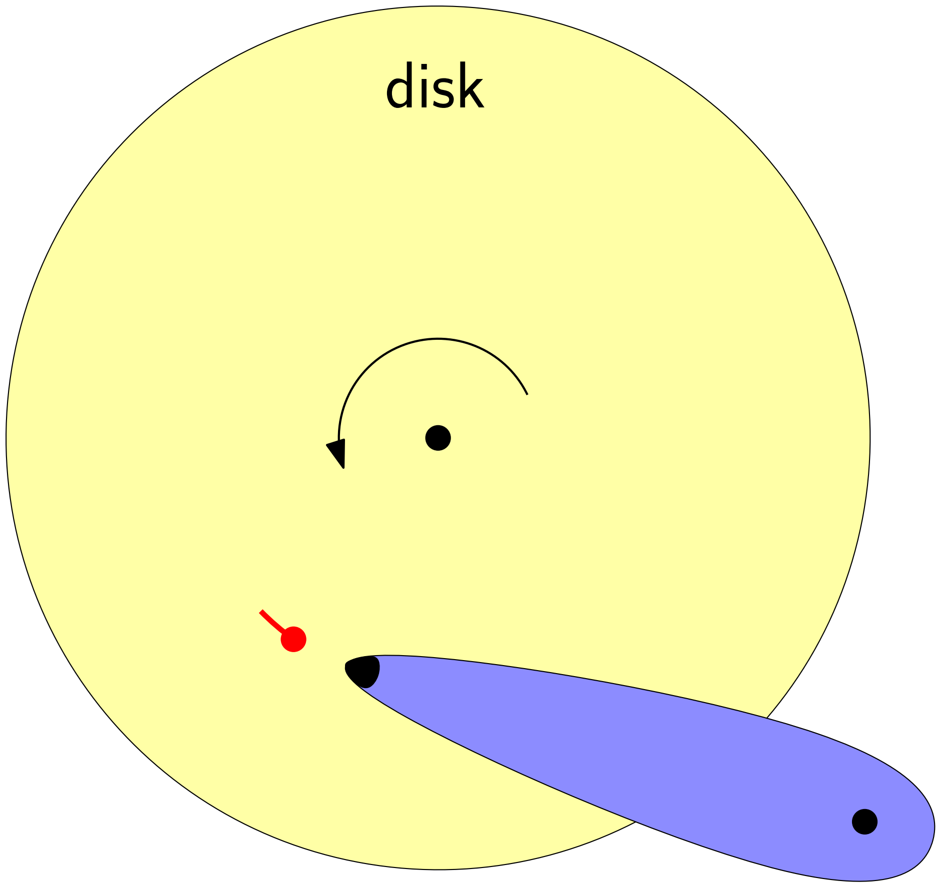


Using external memory

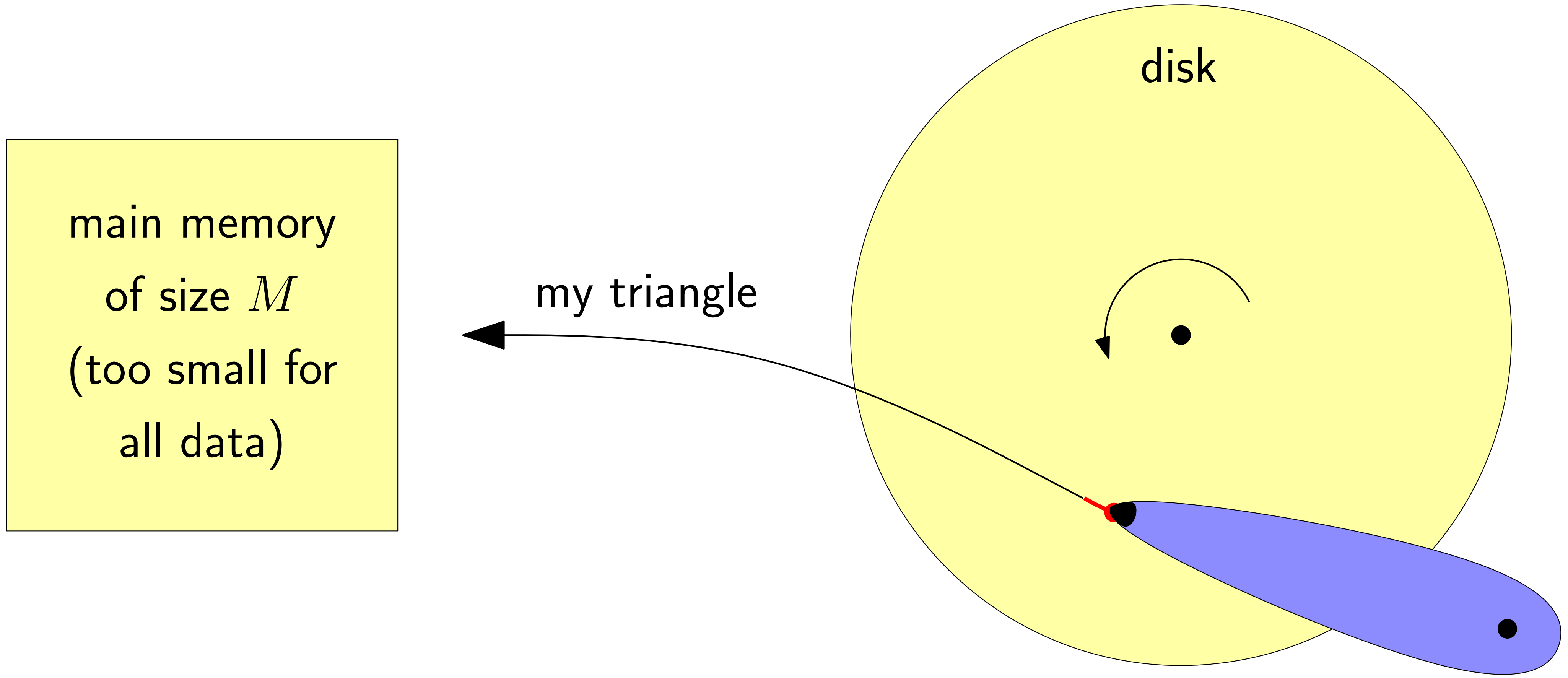


Using external memory

main memory
of size M
(too small for
all data)

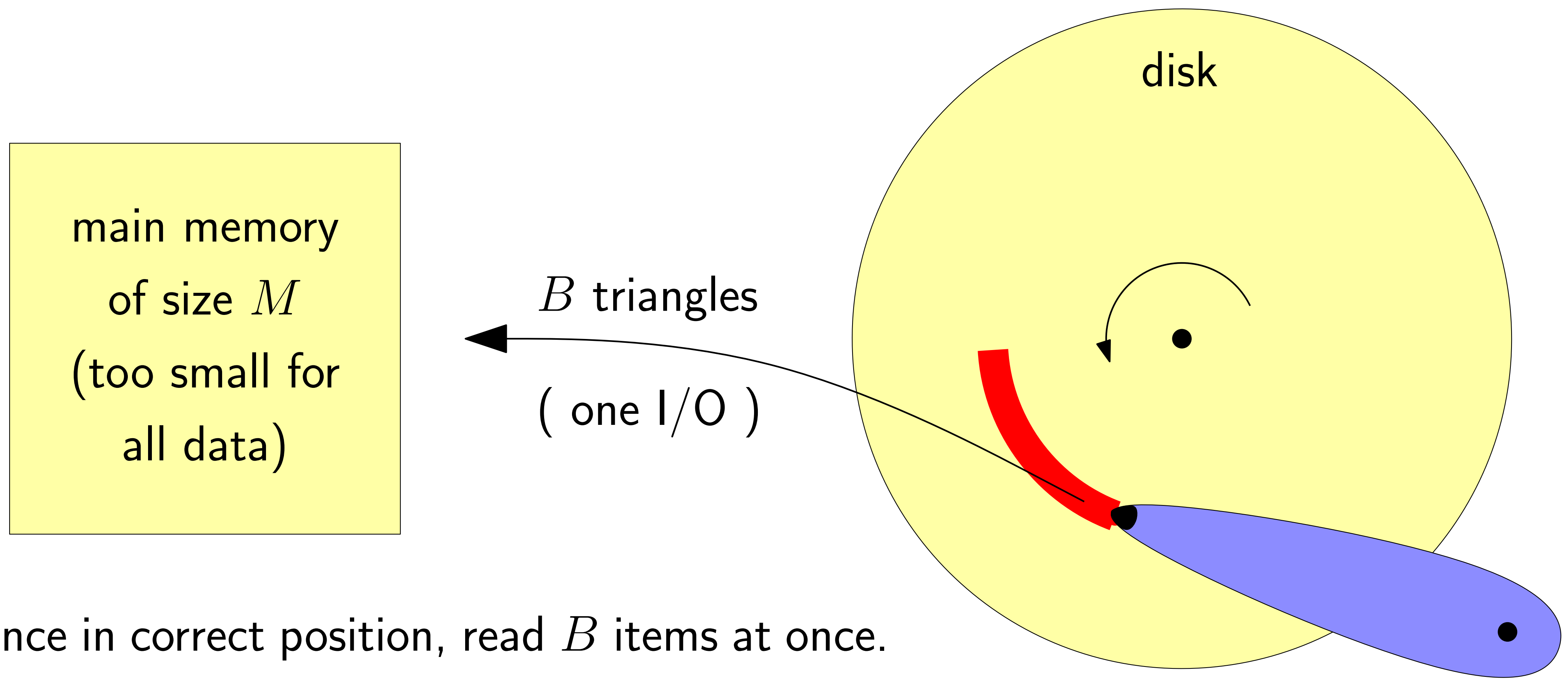


Waiting for one triangle takes $\approx 1\,000\,000$ CPU cycles



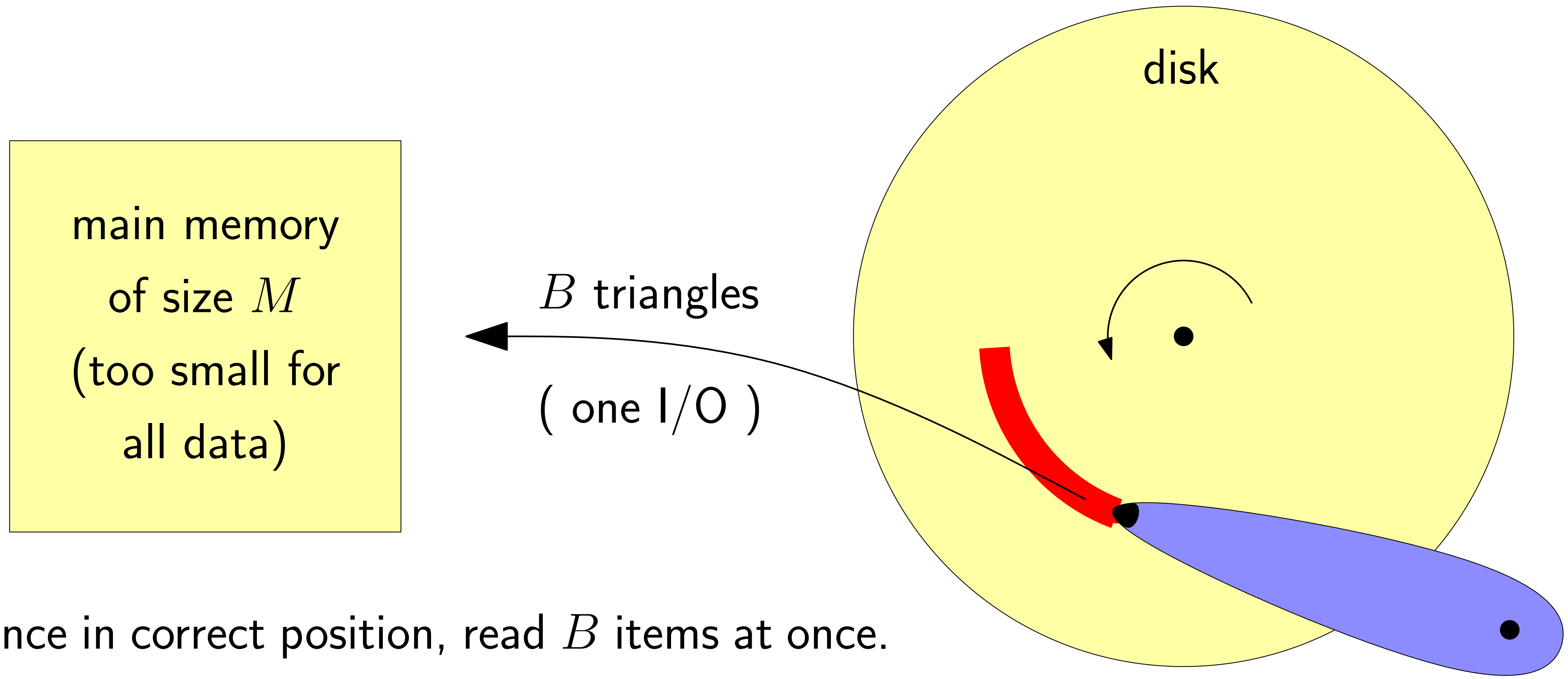
Using external memory

Waiting for one triangle takes $\approx 1\,000\,000$ CPU cycles



Solution: once in correct position, read B items at once.
(hope you can keep them in memory until you need them)

Waiting for one triangle takes $\approx 1\,000\,000$ CPU cycles



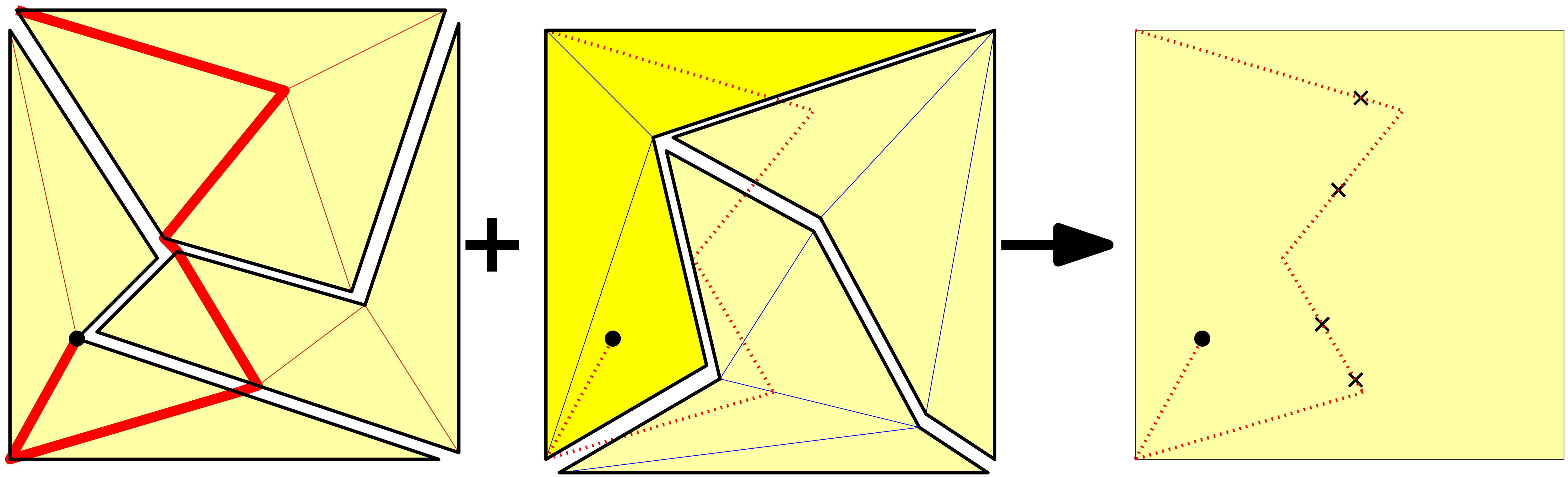
Solution: once in correct position, read B items at once.
(hope you can keep them in memory until you need them)

Analysing algorithms that work on data on disk: number of I/O's dominate.

$$scan(n) = \frac{n}{B} \quad < \quad sort(n) = \frac{n}{B}\log_{M/B}\frac{n}{B} \quad << \quad n \quad \text{I/O's}$$

Overlaying triangulations on disk?

Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other:

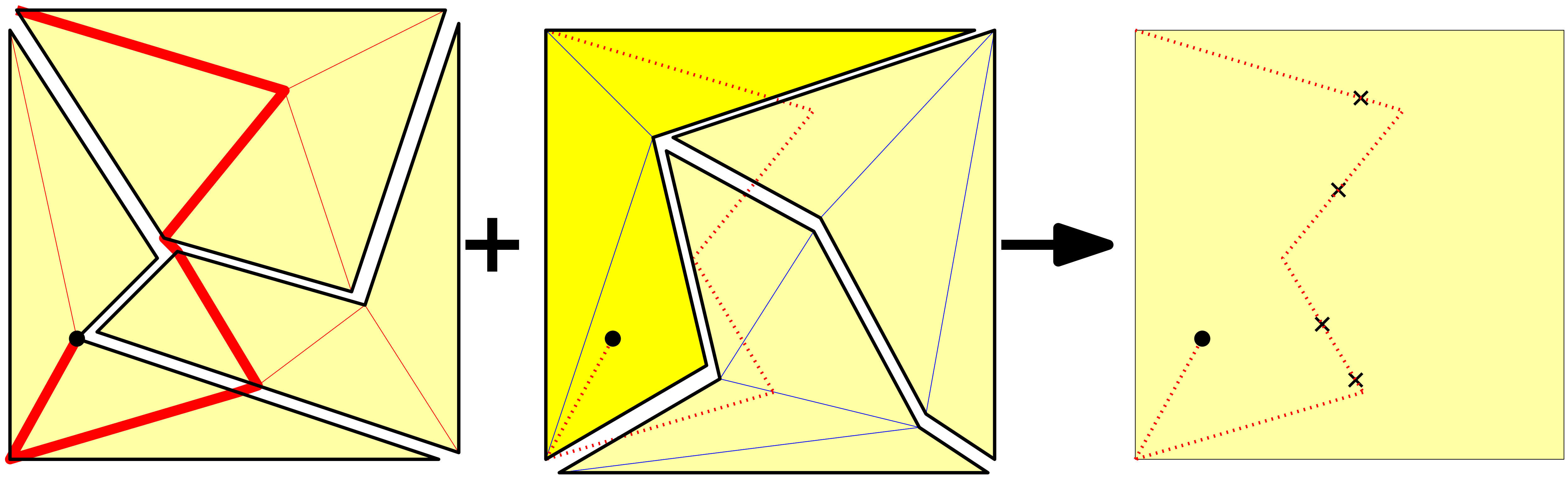
- $\Theta(1)$ operations per edge
- $\Theta(1)$ operations per crossing

Total: $\Theta(n+k)$ CPU-operations (for n triangles, k intersections)

On disk, data arranged in blocks.

Overlaying triangulations on disk?

Maps: ..., triangulations



DFS in one triangulation, traverse triangles in the other:

- $\Theta(1)$ operations per edge
- $\Theta(1)$ operations per crossing

Total: $\Theta(n+k)$ CPU-operations (for n triangles, k intersections)

On disk, data arranged in blocks. 1 I/O $\approx$ 1,000,000 CPU-ops. $\Theta(n+k)$ I/O's?

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:

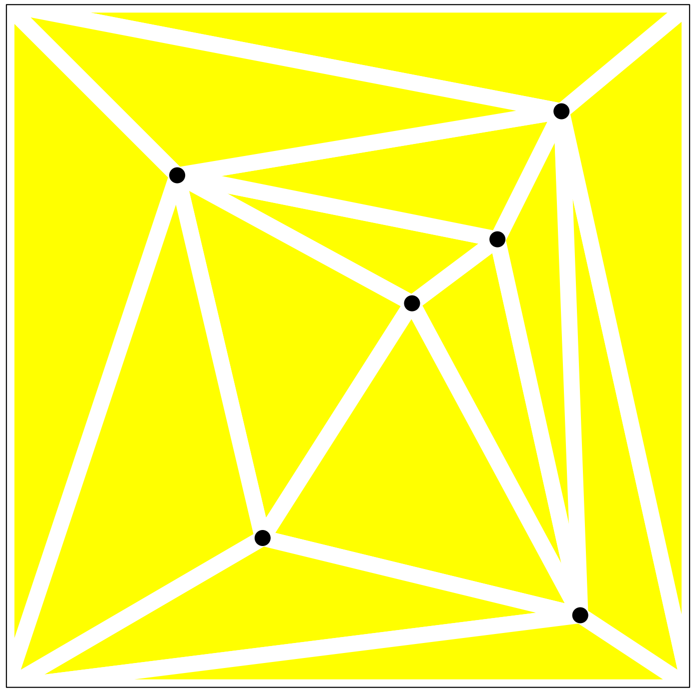
- map overlay in $O(scan(n))$ I/O's;
- point location in $O(\log_B n)$ I/O's;
- for triangulations: basic updates in $O(\log_B n)$ I/O's.

Condition: input must be *fat* triangulation (all angles $>$ positive constant), or a *low-density* set of segments (for any $\square$, number of intersecting segments bigger than $\square$ is $O(1)$)

Ingredients: quadtrees ...

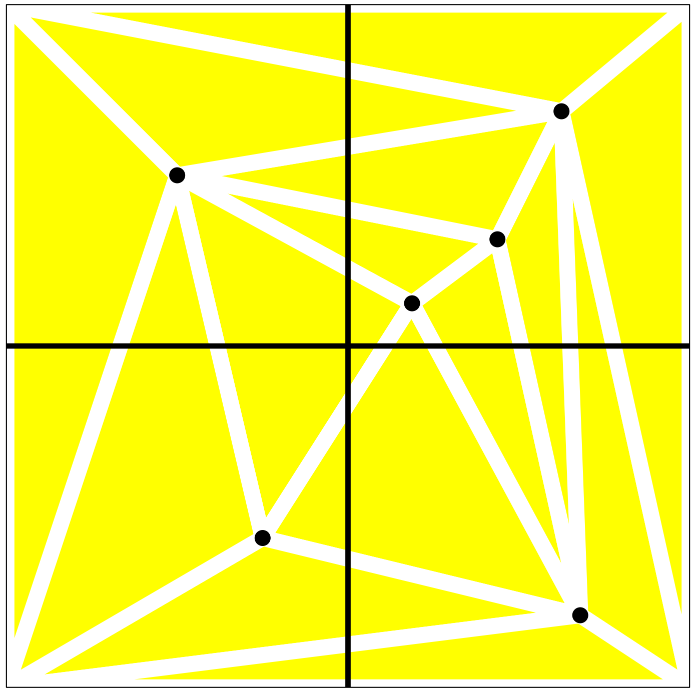
Quadtree: divide unit square into quadrants, refine until amount of data per cell is small.

(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.

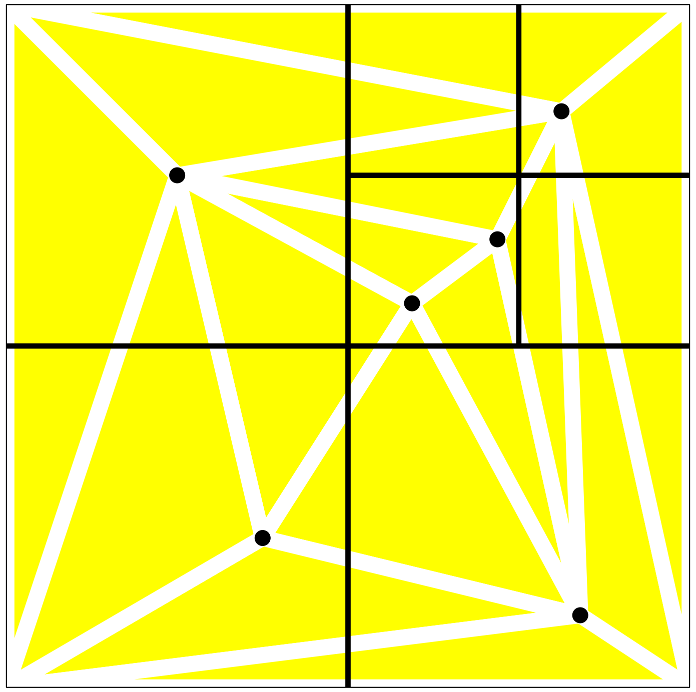
(for example: until every cell has at most one vertex)



Ingredients: quadtrees ...

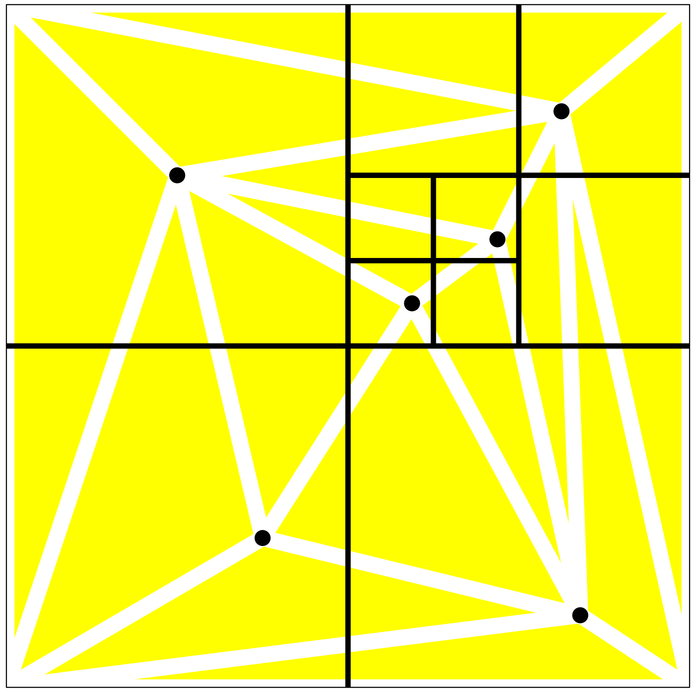
Quadtree: divide unit square into quadrants, refine until amount of data per cell is small.

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



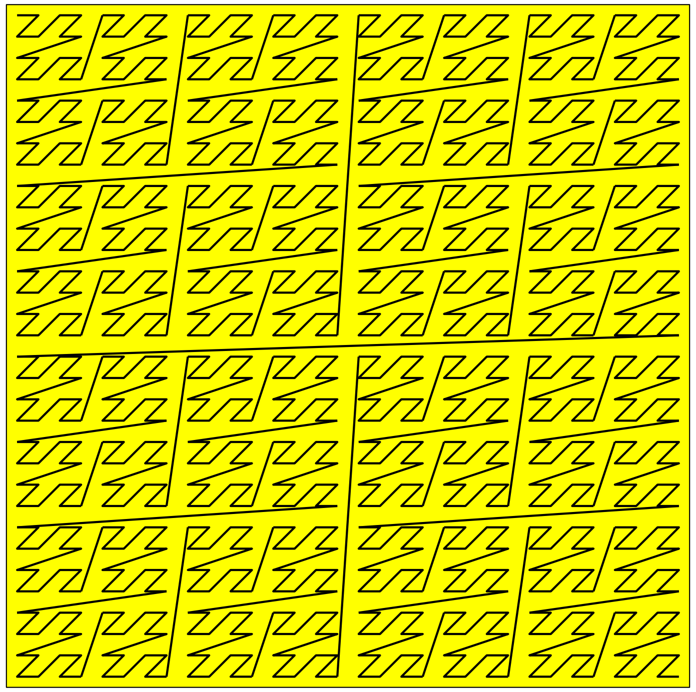
Quadtree: divide unit square into quadrants, refine until amount of data per cell is small.

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

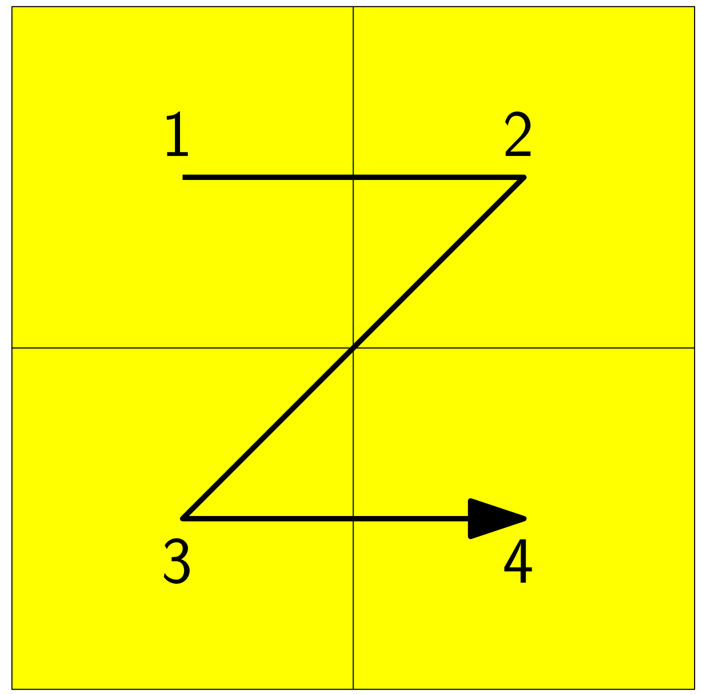


Ingredients: ... and Z-order

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

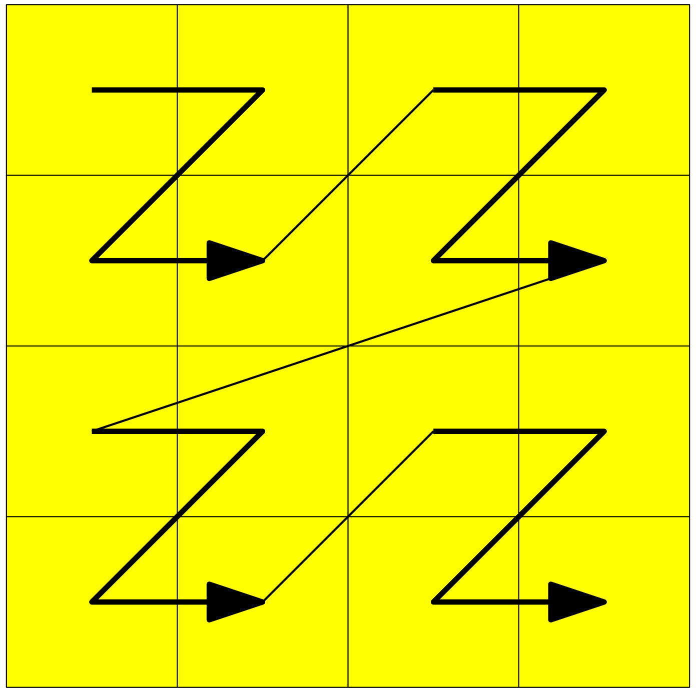


Z-order space-filling curve: visit quadrants recursively in order NW, NE, SW, SE

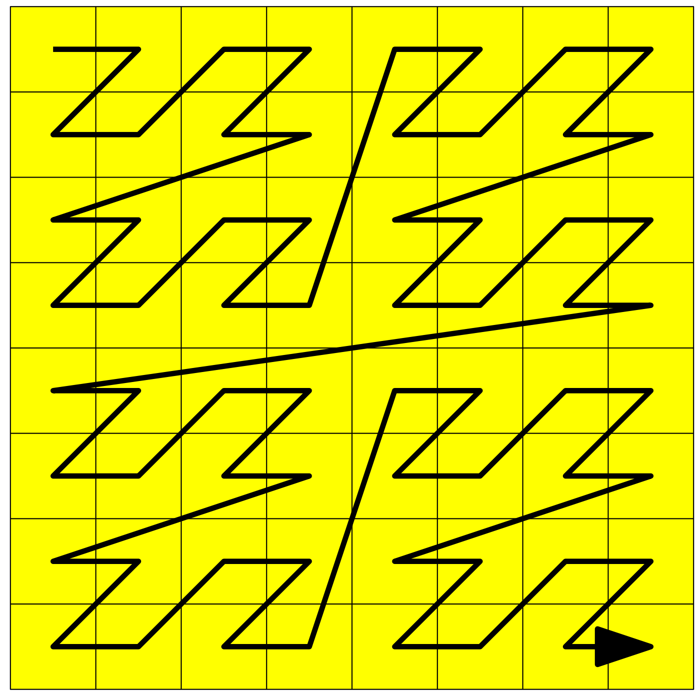


Ingredients: ... and Z-order

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

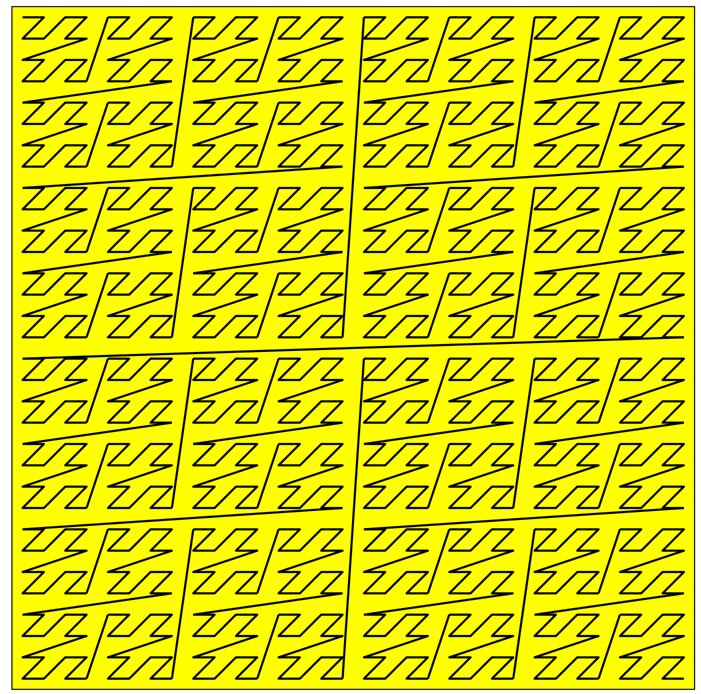


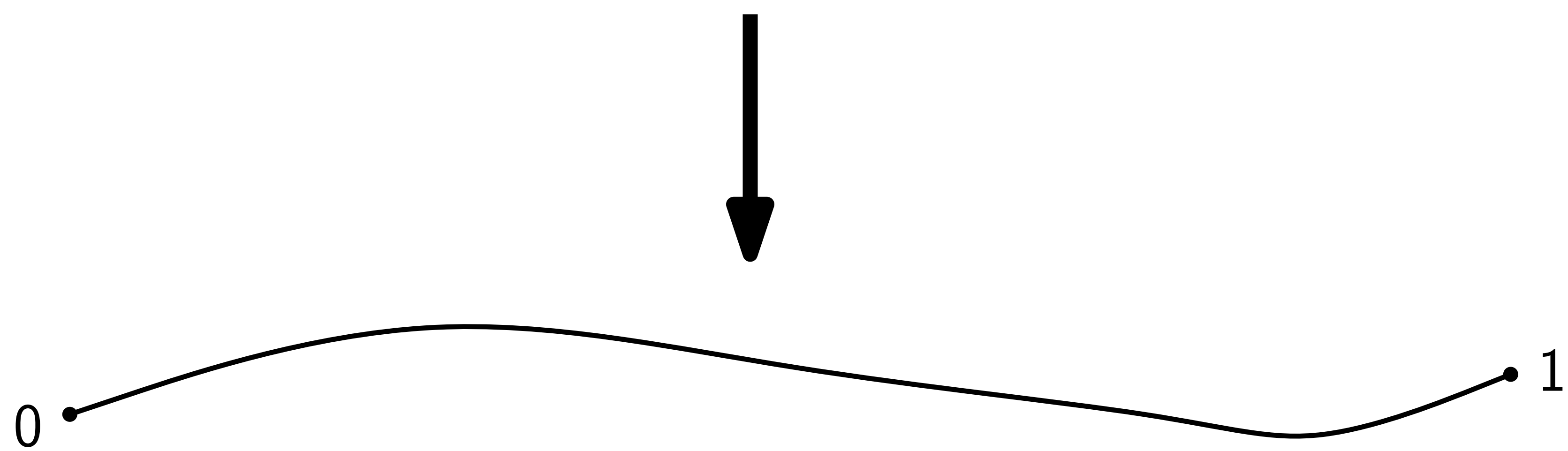
Z-order space-filling curve: visit quadrants recursively in order NW, NE, SW, SE



Ingredients: ... and Z-order

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

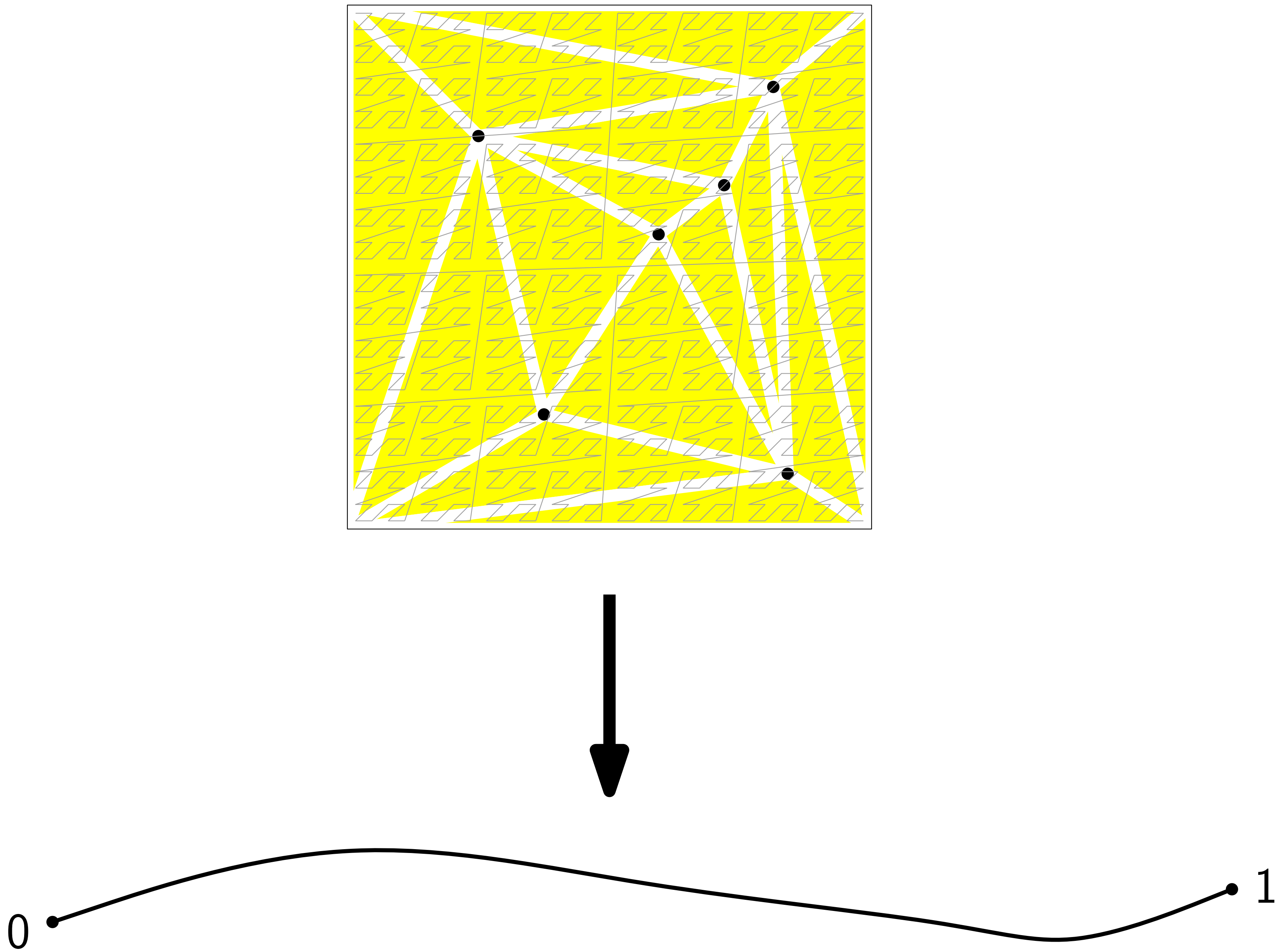




Ingredients: quadtrees and Z-order

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

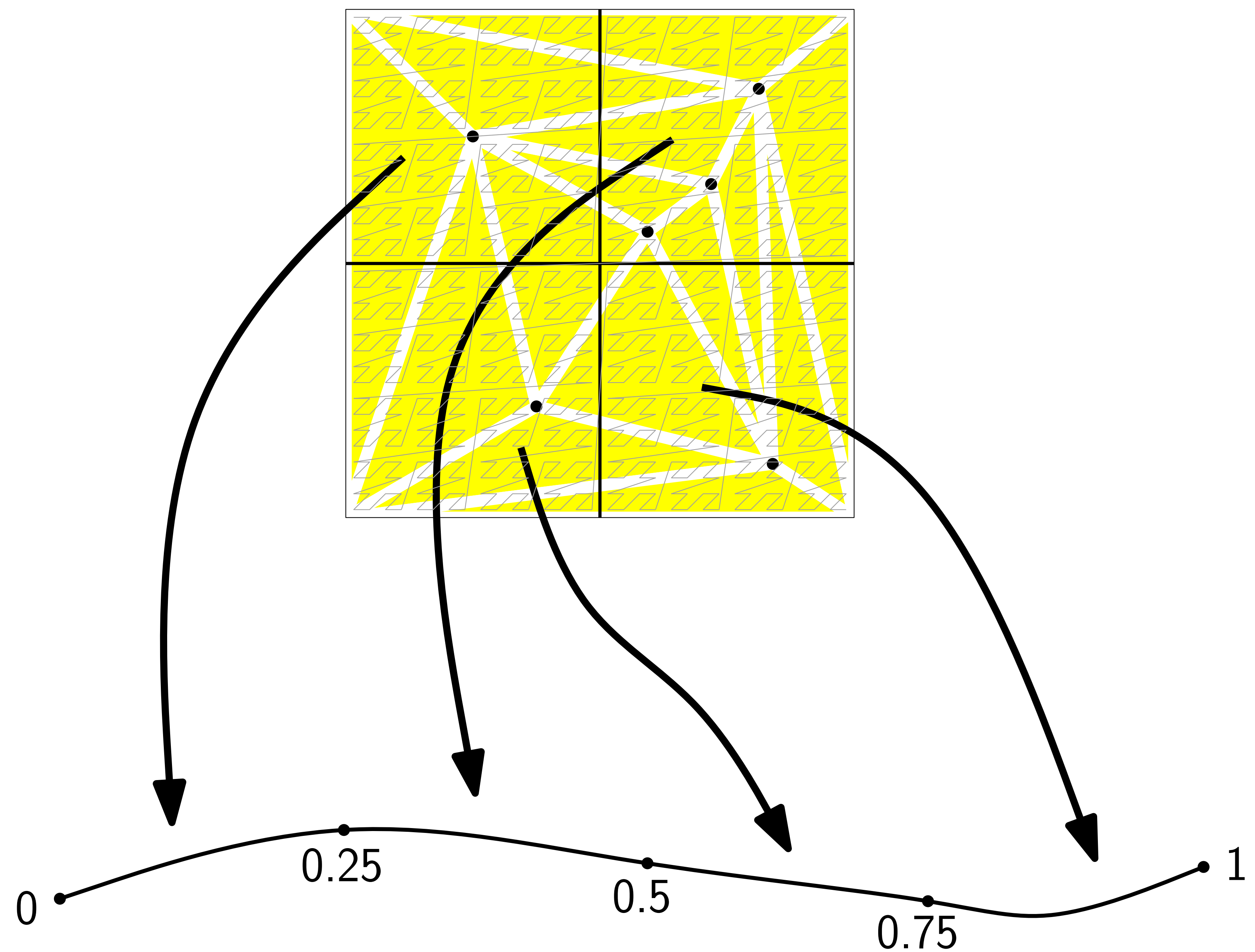
Quadtree subdivision $\equiv$ subdivision of Z-order curve



Ingredients: quadtrees and Z-order

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

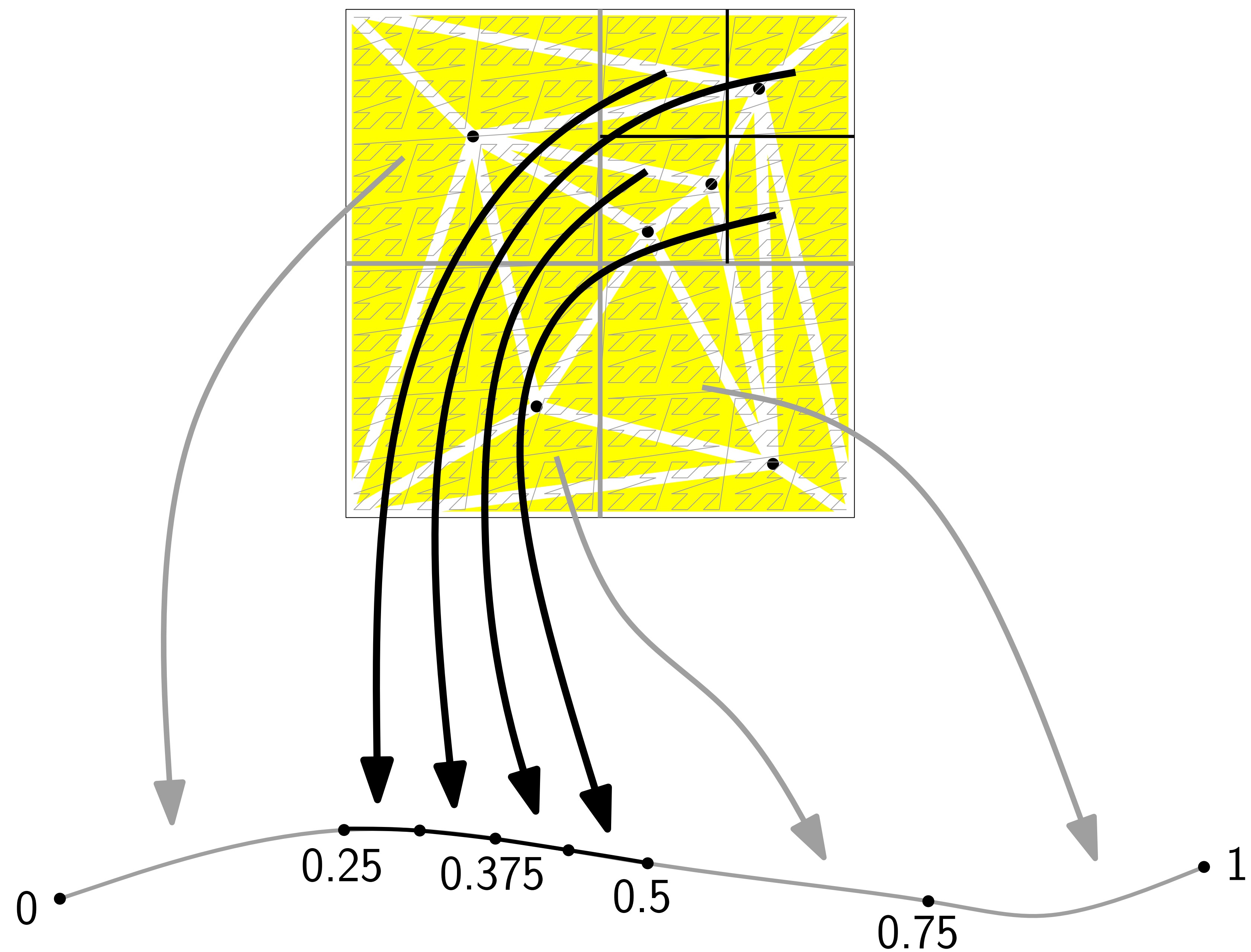
Quadtree subdivision $\equiv$ subdivision of Z-order curve



Ingredients: quadtrees and Z-order

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

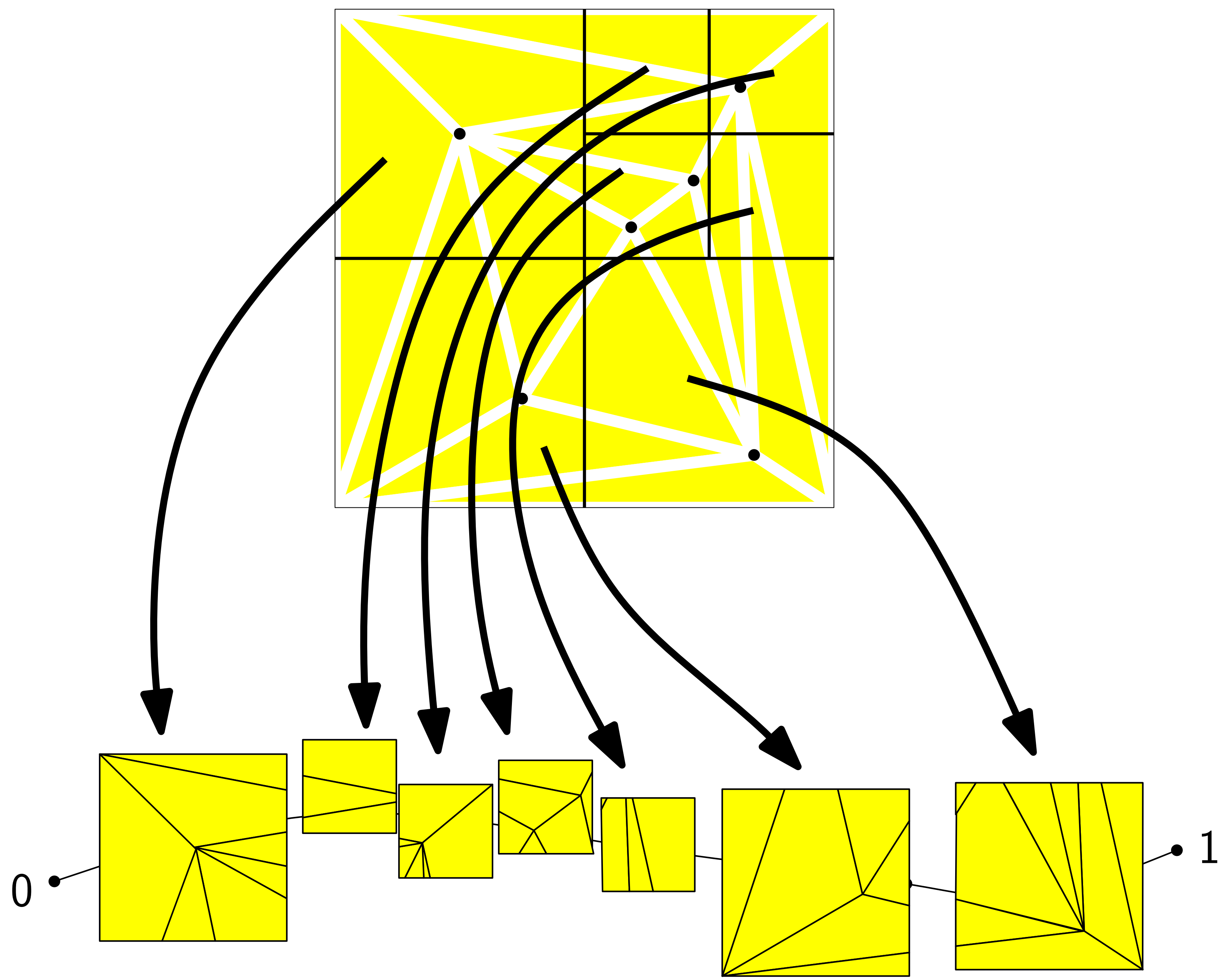
Quadtree subdivision $\equiv$ subdivision of Z-order curve



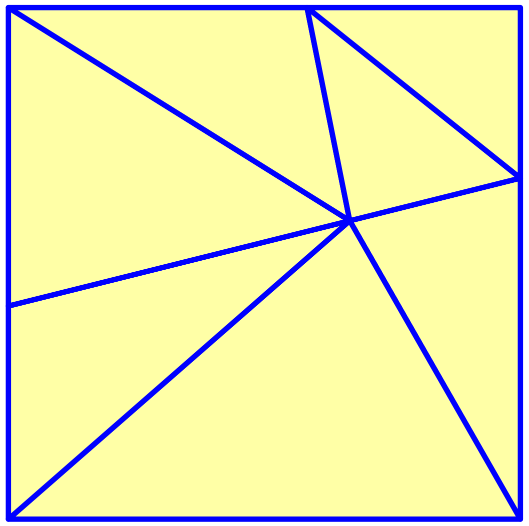
Ingredients: quadtrees and Z-order

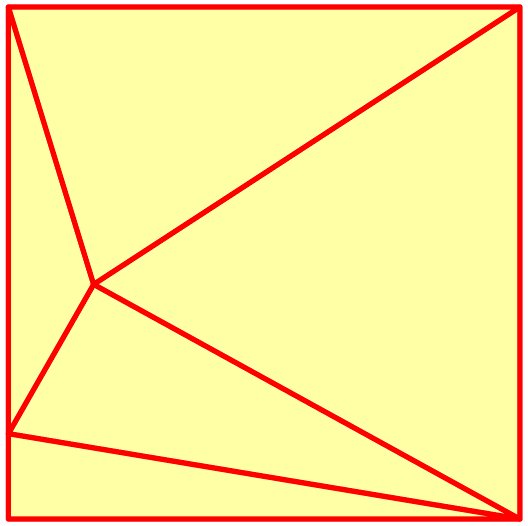
Quadtree cell $\equiv$ interval on Z-order curve

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

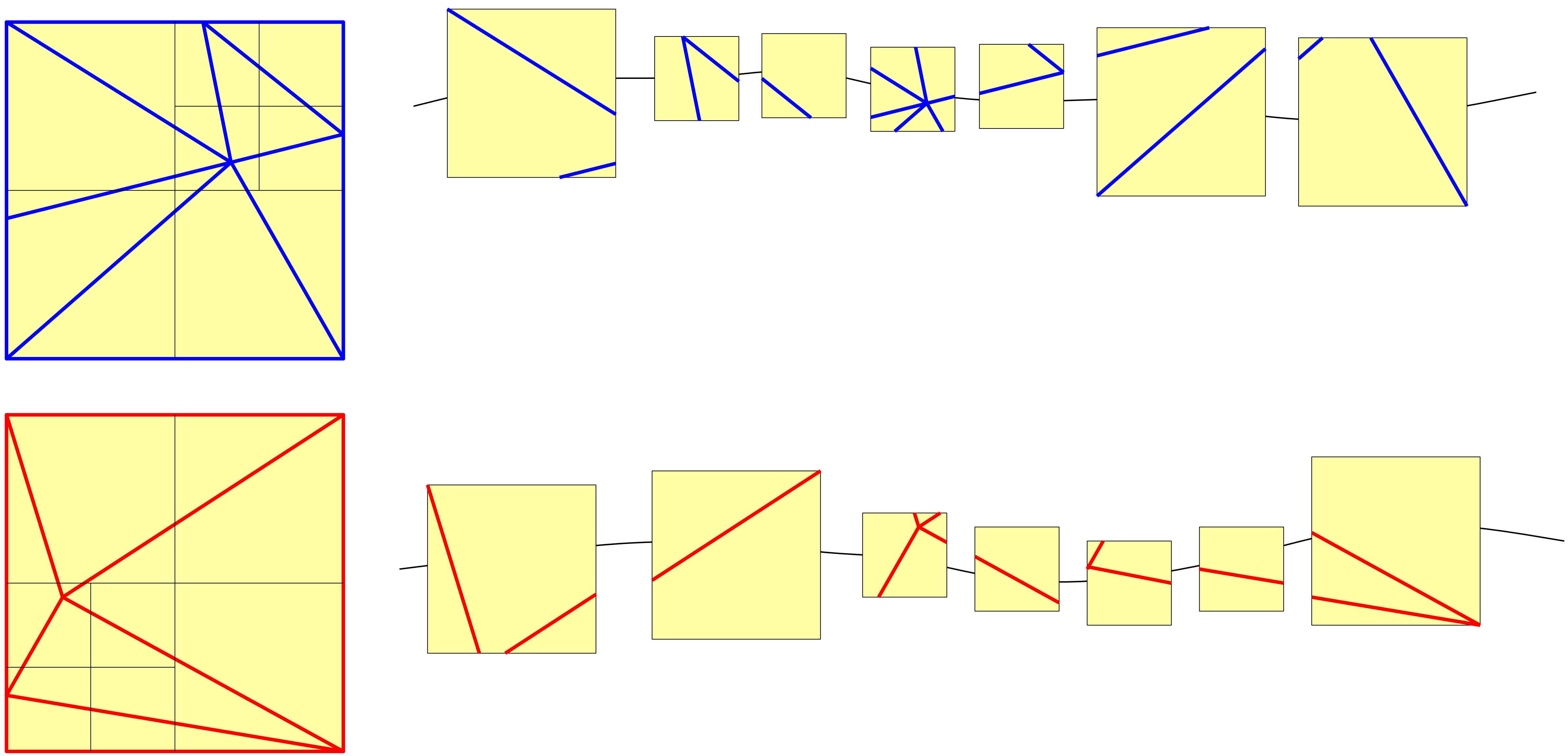


Map overlay with quadtrees in Z-order

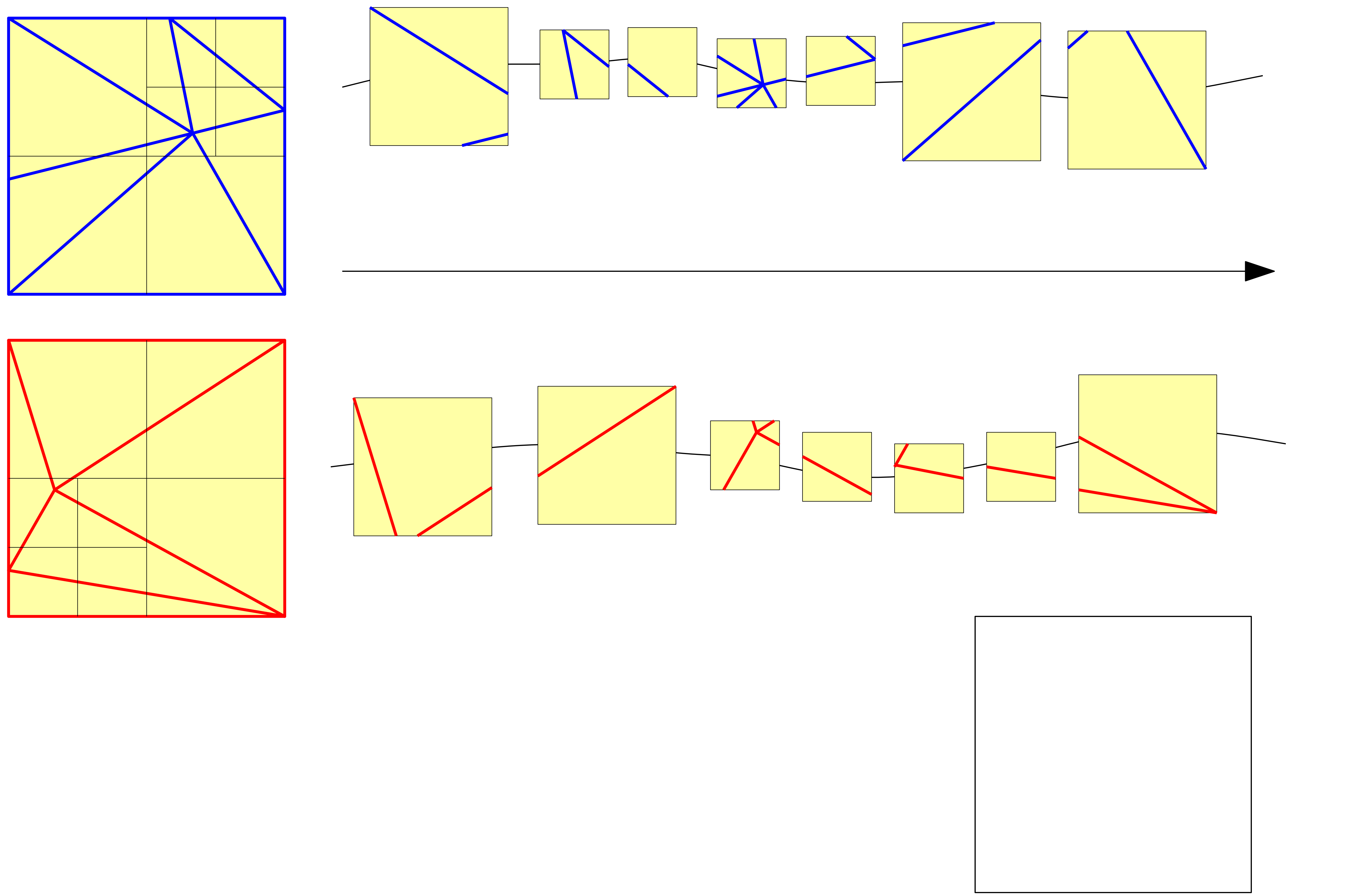




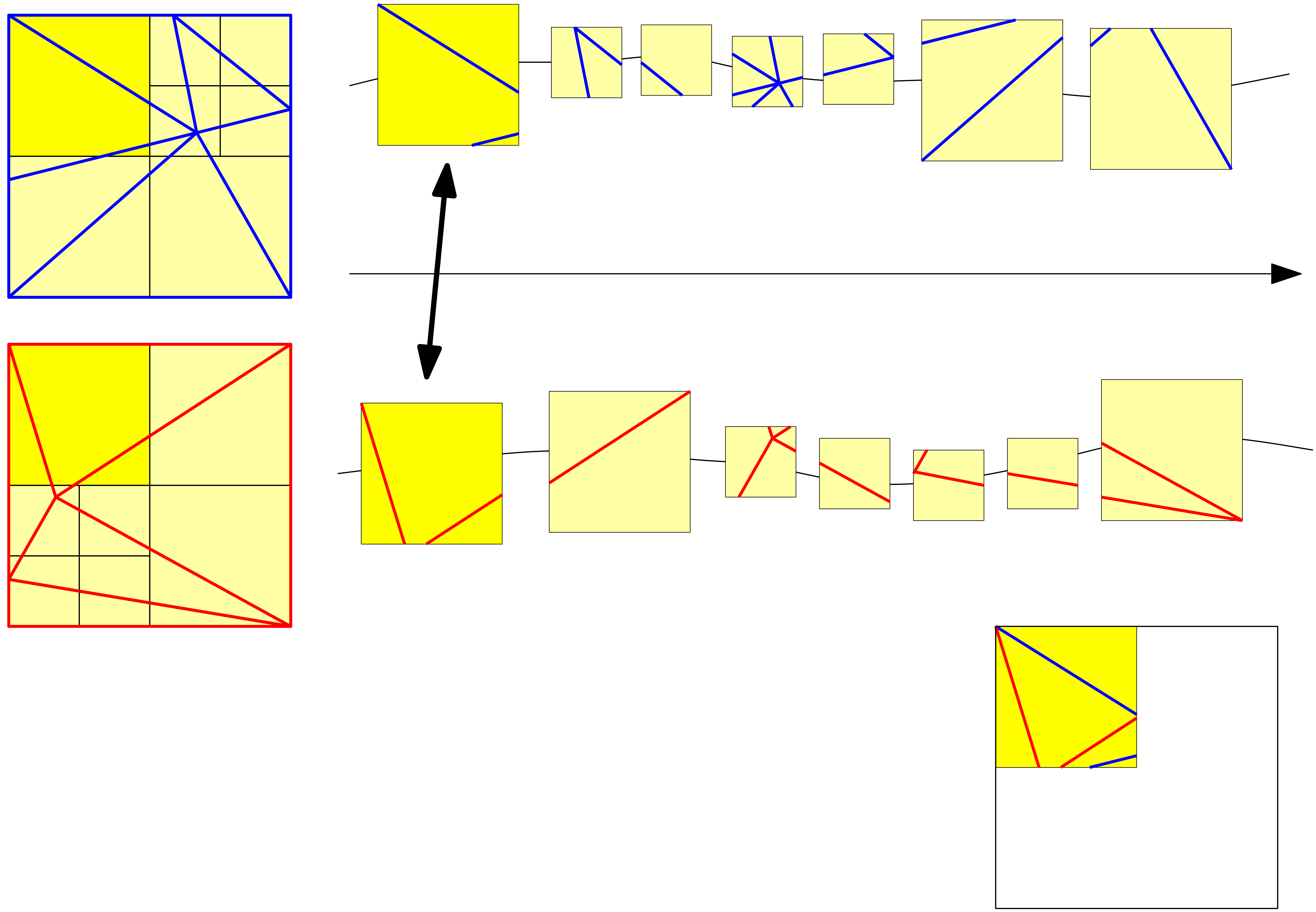
Map overlay with quadtrees in Z-order



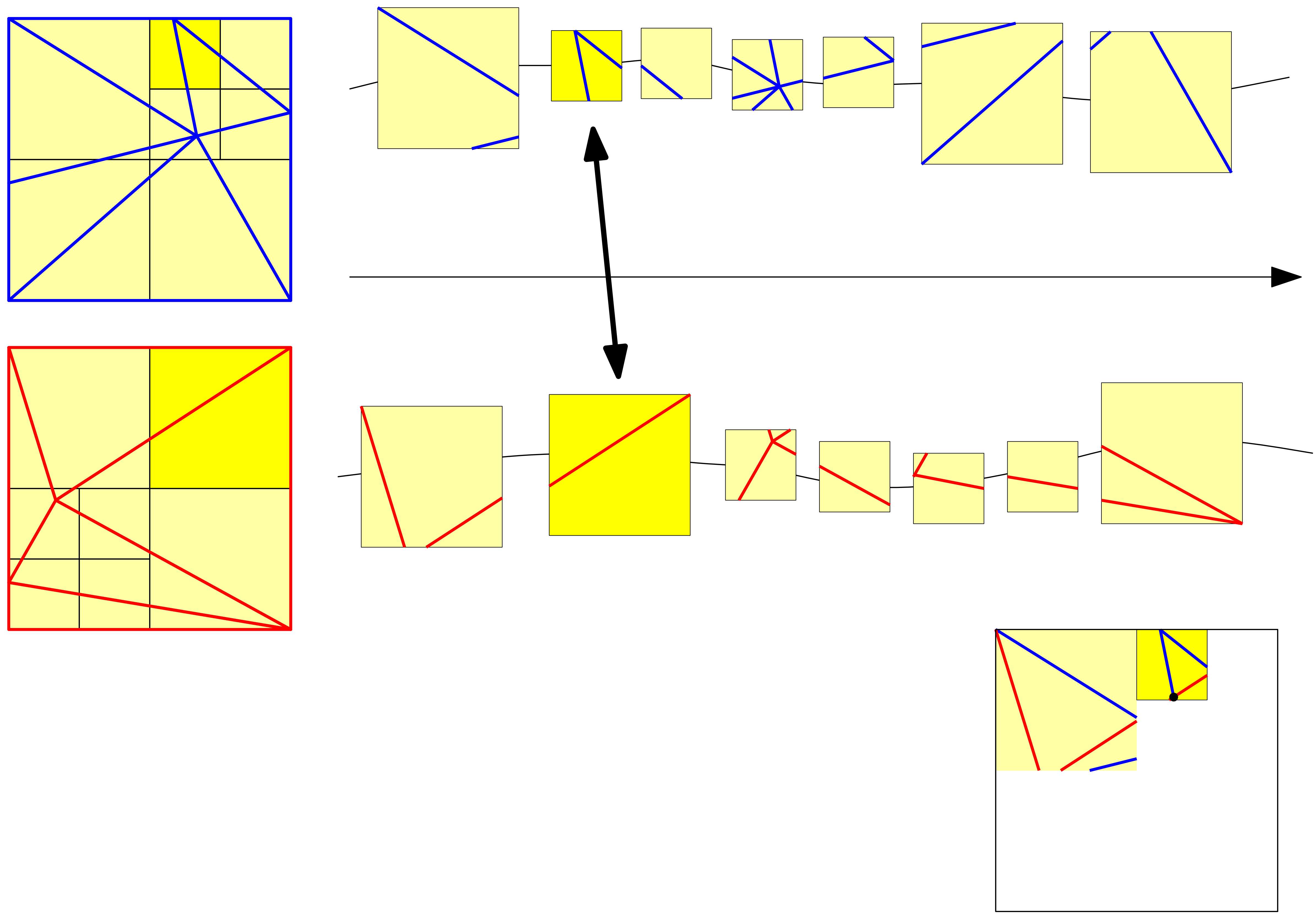
Map overlay with quadtrees in Z-order



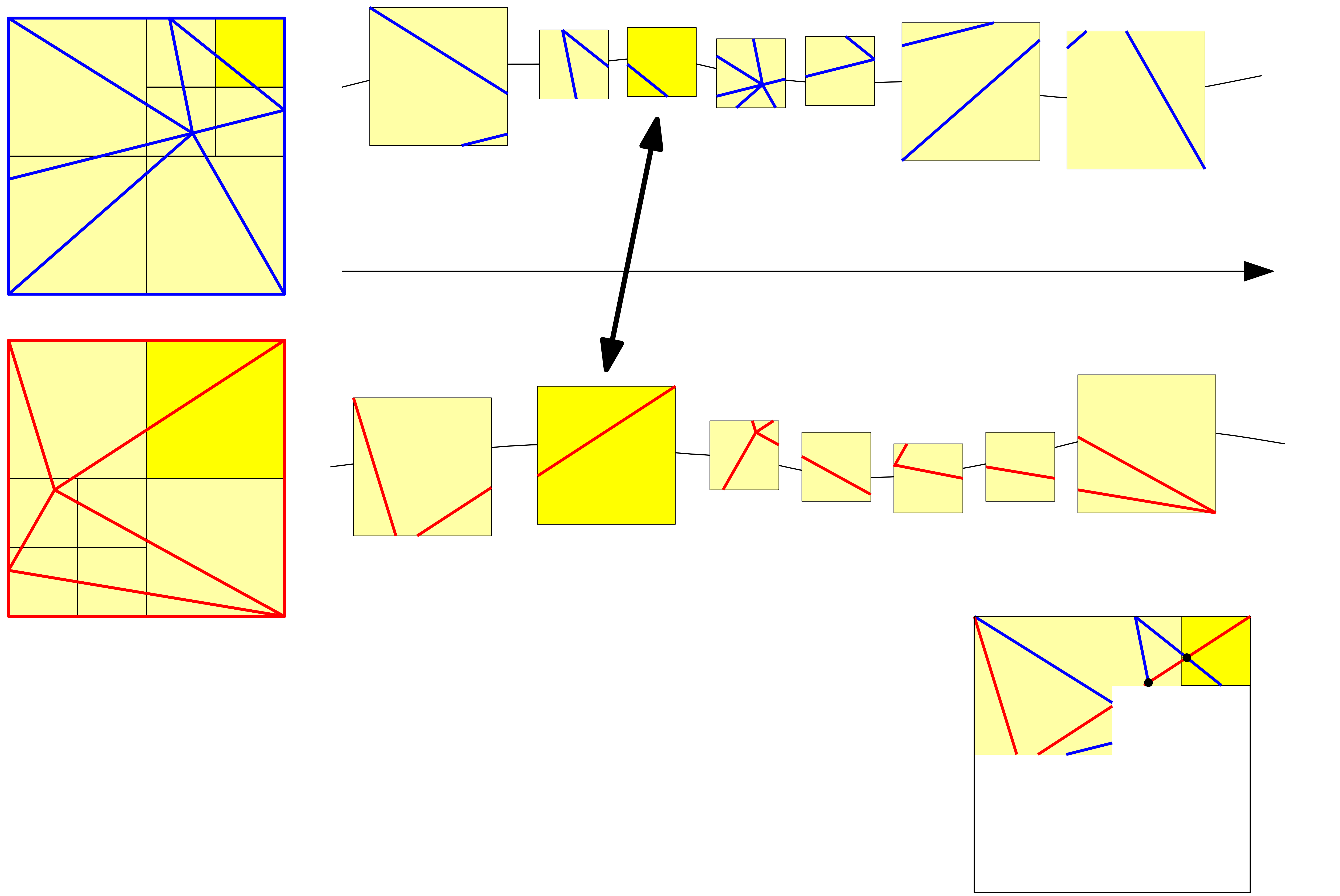
Map overlay with quadtrees in Z-order



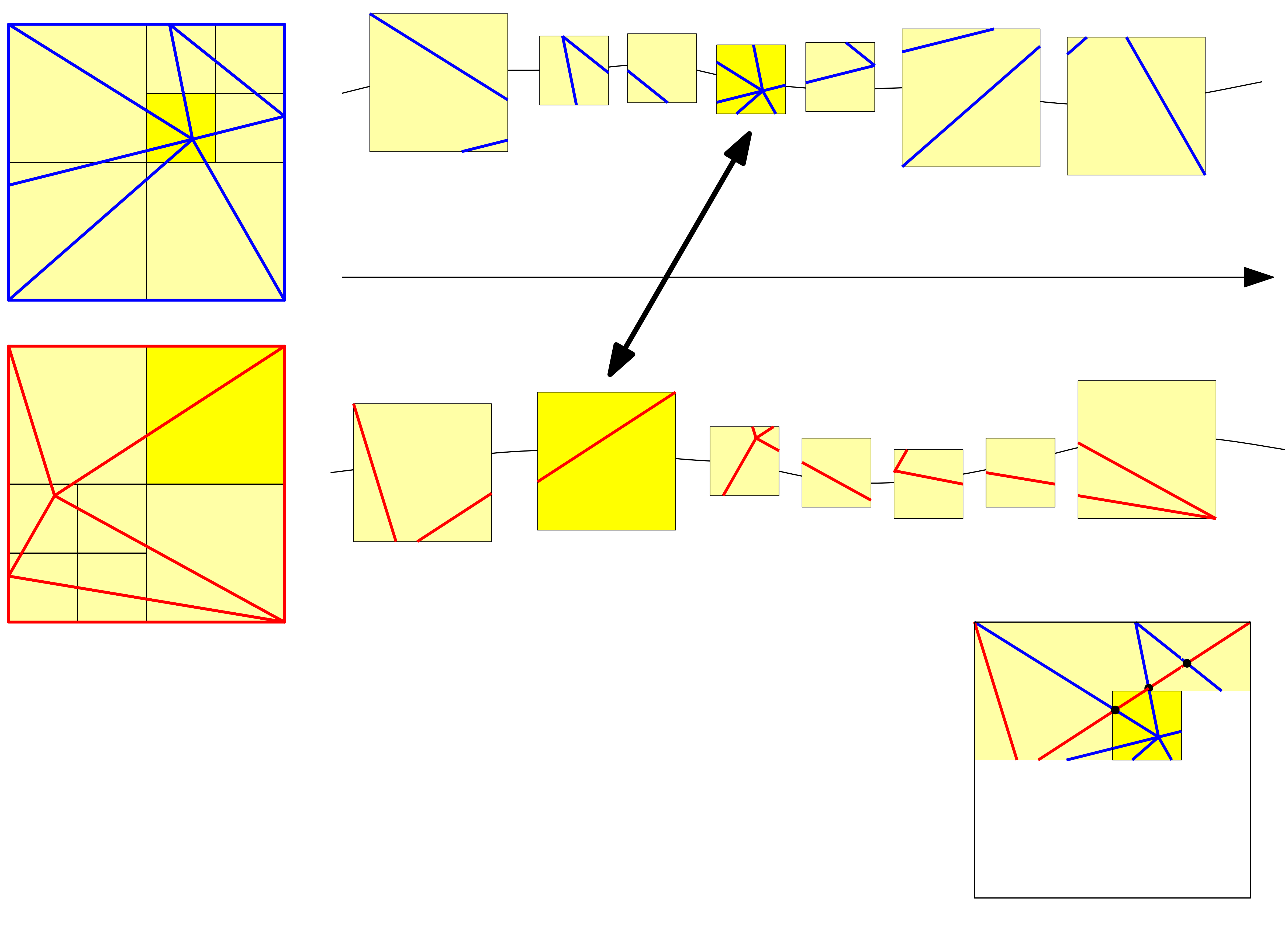
Map overlay with quadtrees in Z-order



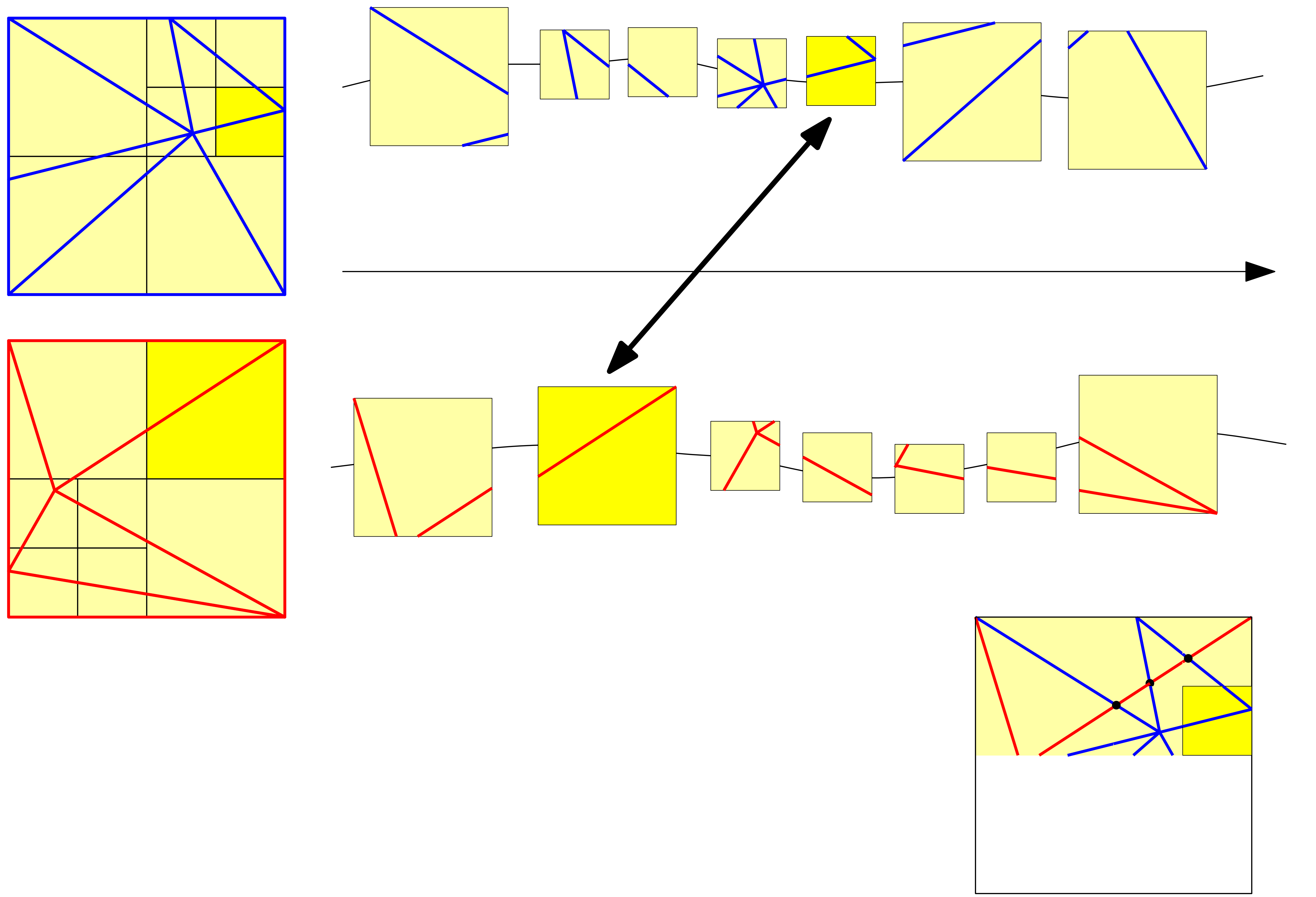
Map overlay with quadtrees in Z-order



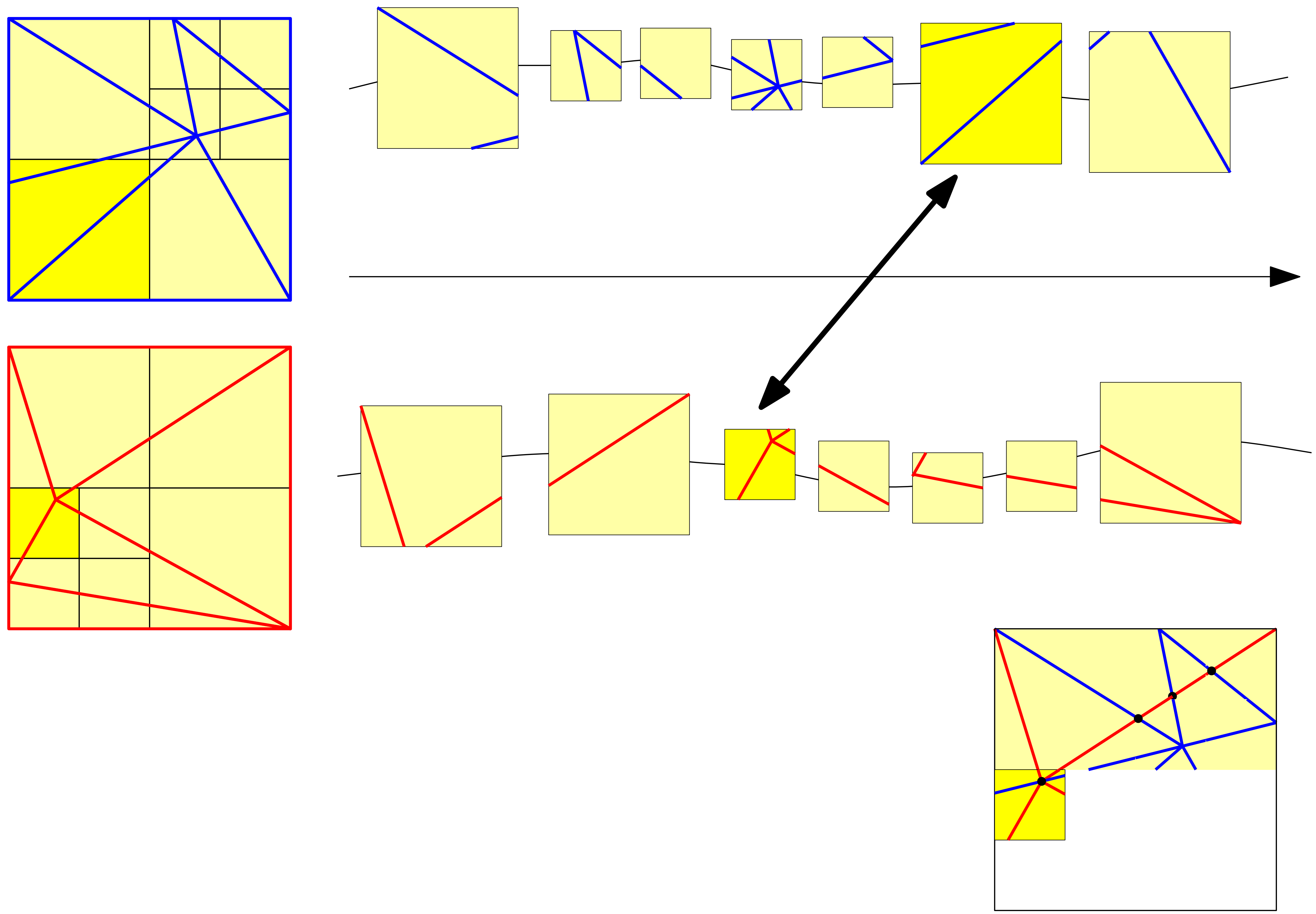
Map overlay with quadtrees in Z-order



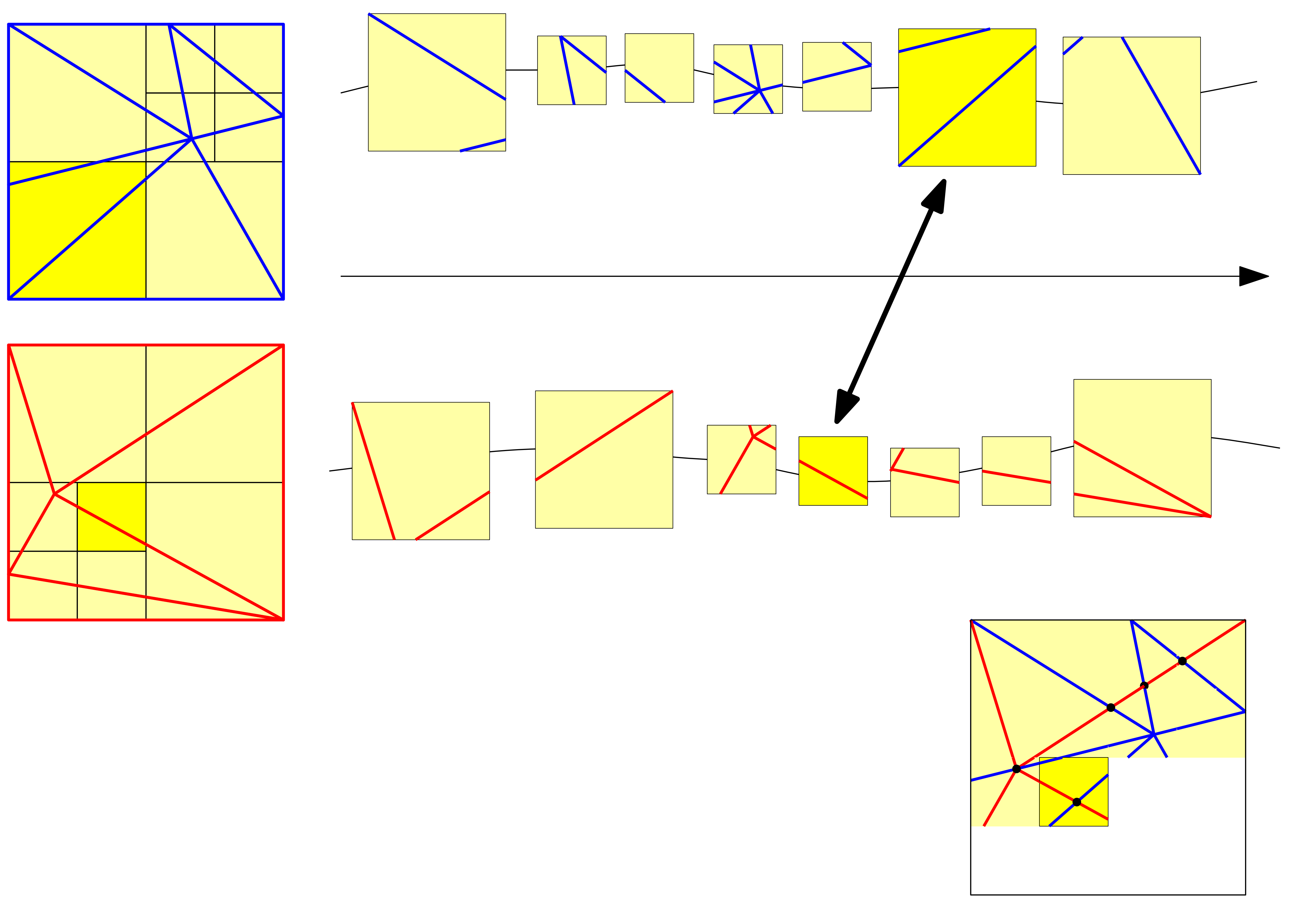
Map overlay with quadtrees in Z-order



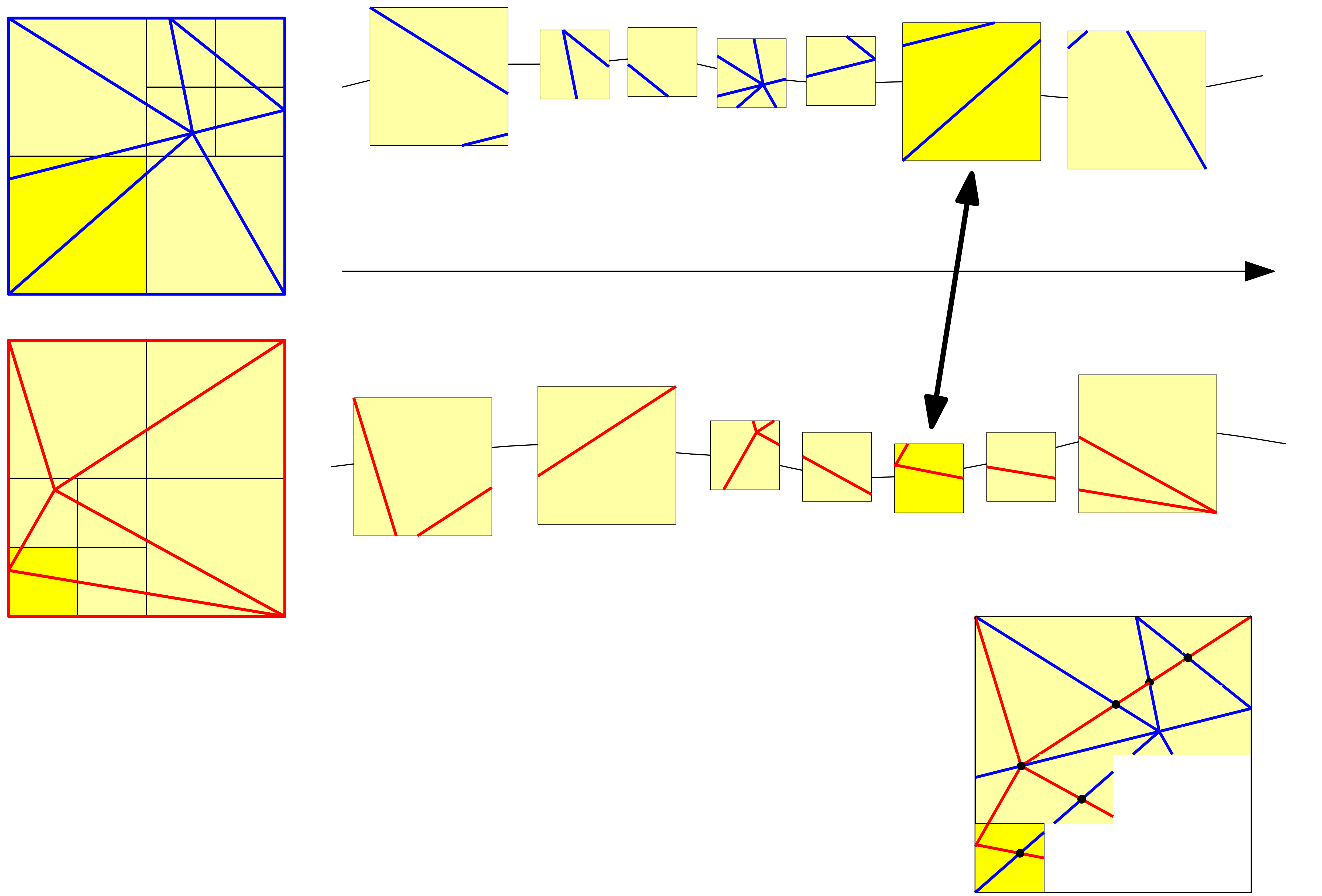
Map overlay with quadtrees in Z-order



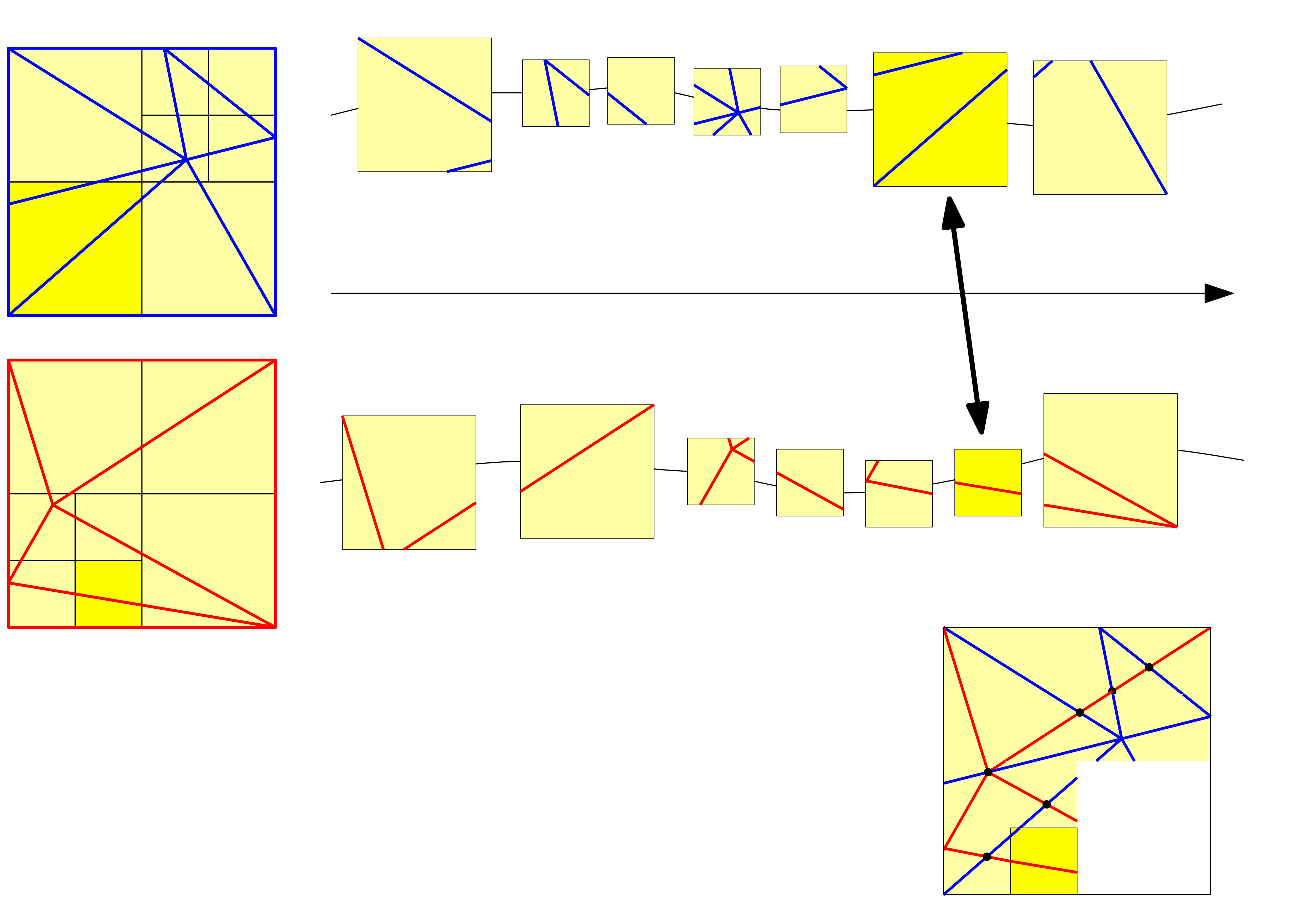
Map overlay with quadtrees in Z-order



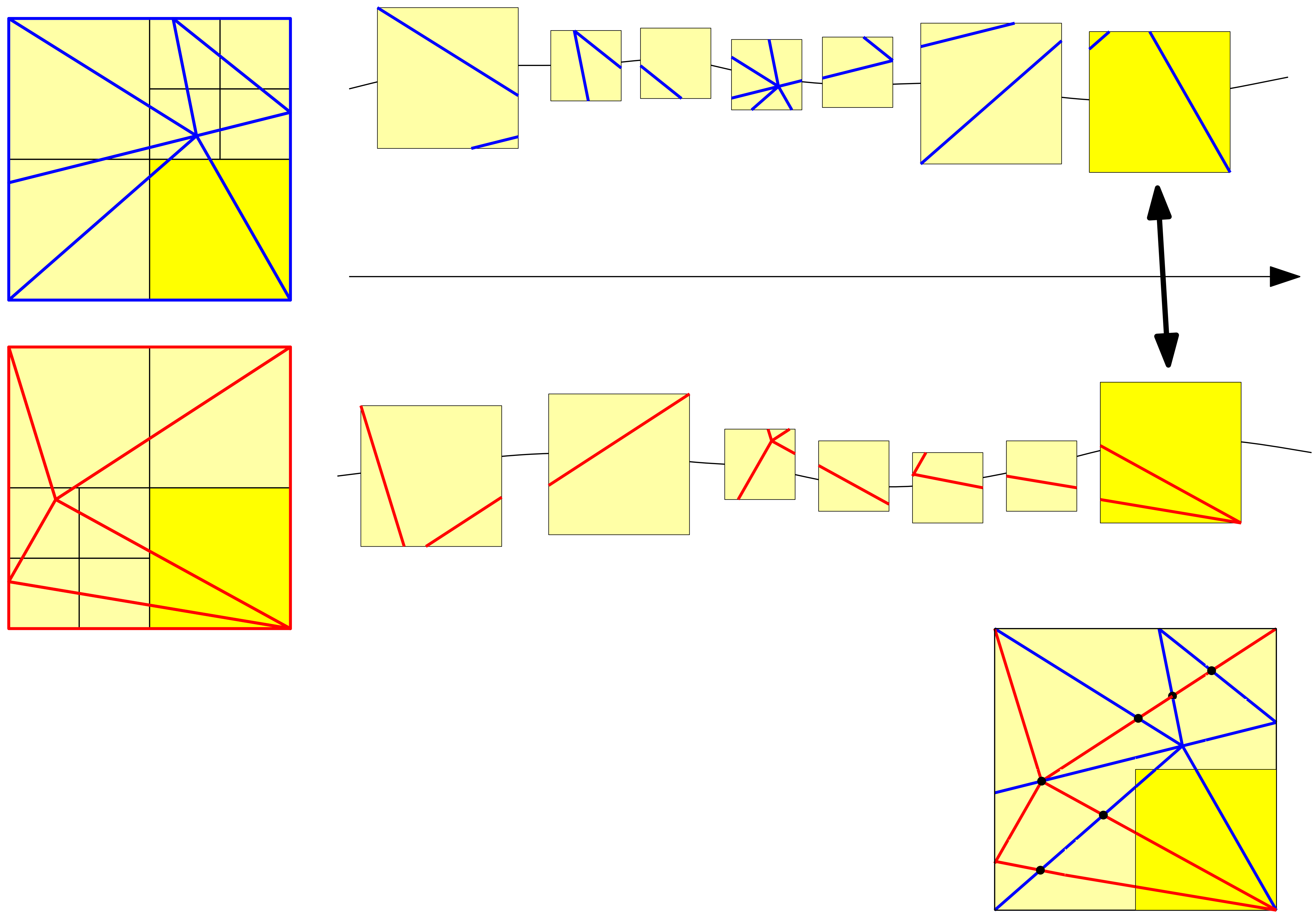
Map overlay with quadtrees in Z-order



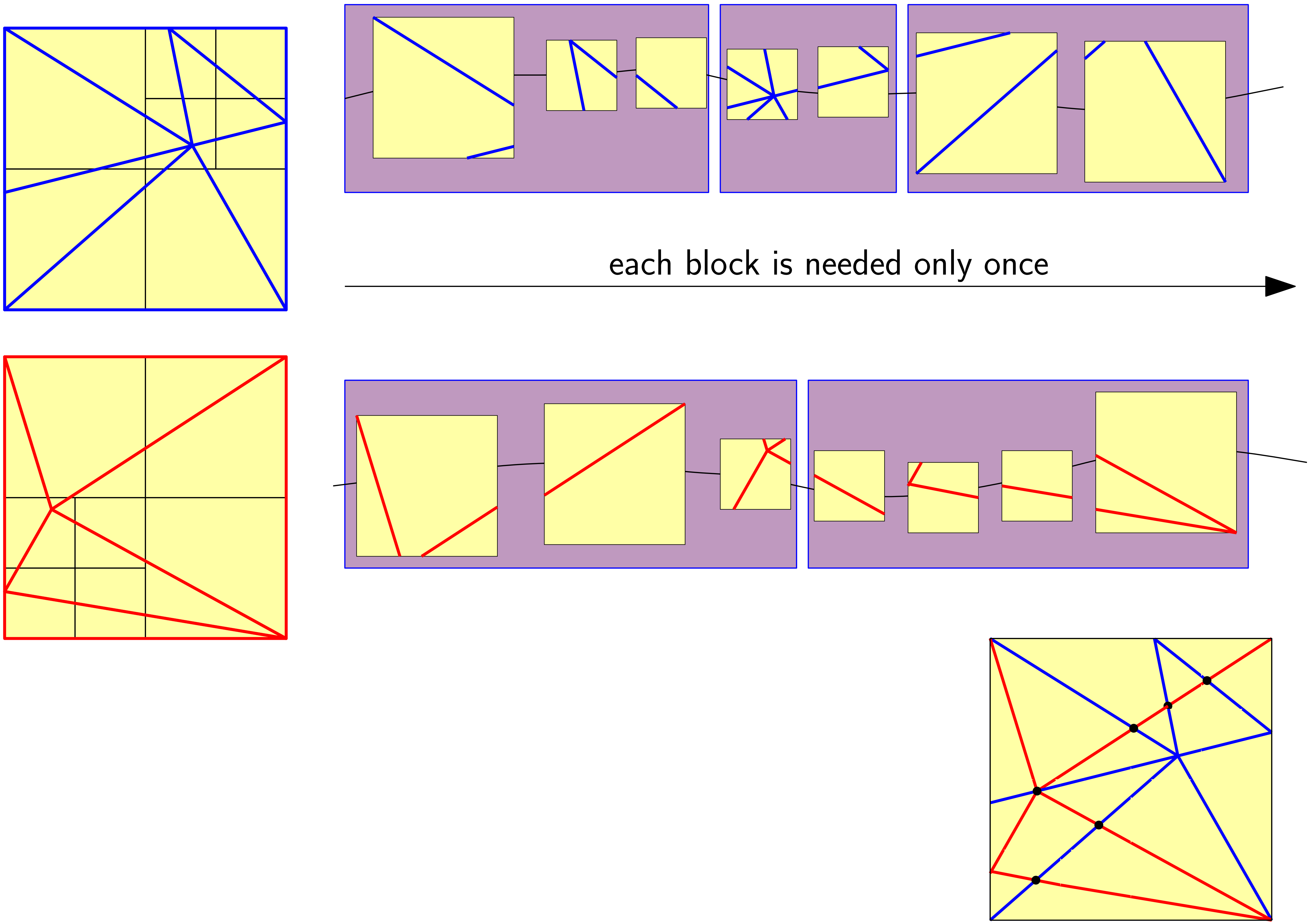
Map overlay with quadtrees in Z-order



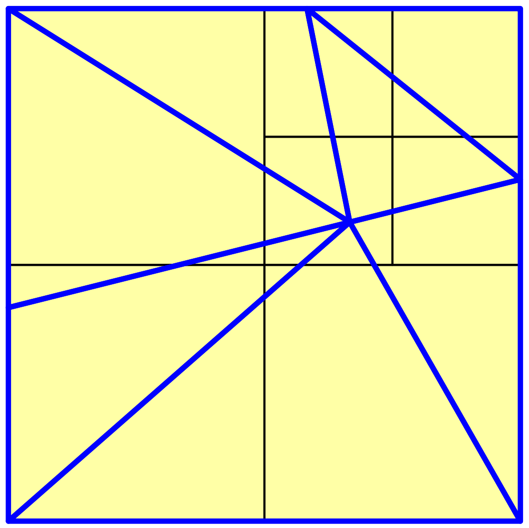
Map overlay with quadtrees in Z-order

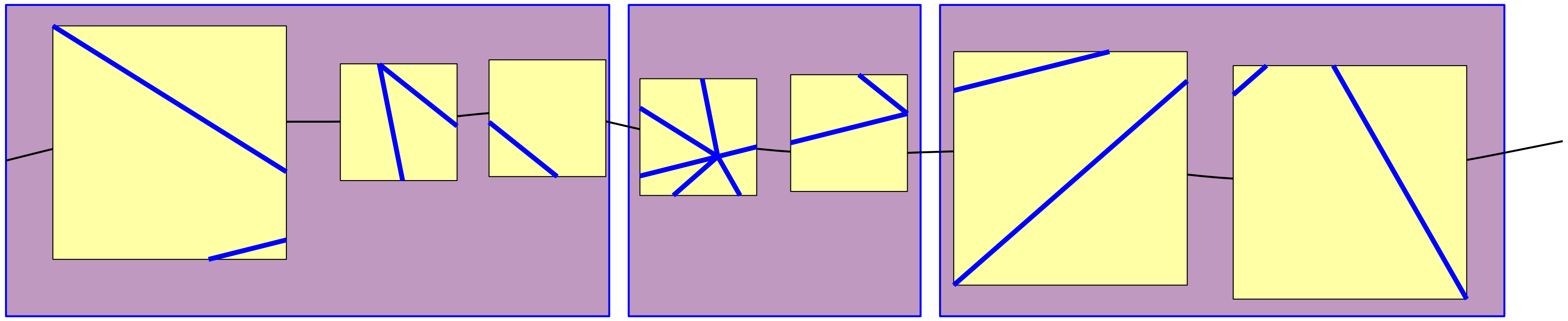


Map overlay with quadtrees in Z-order

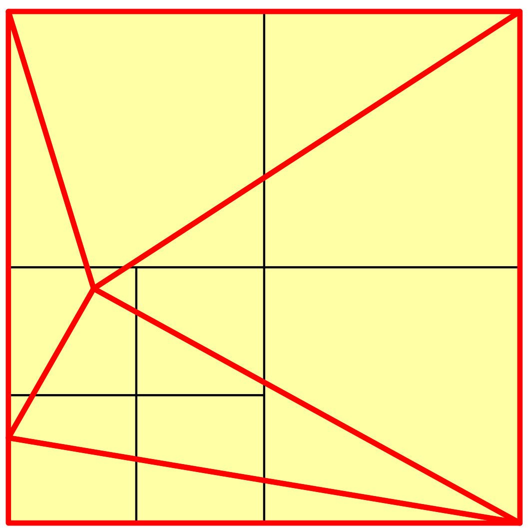


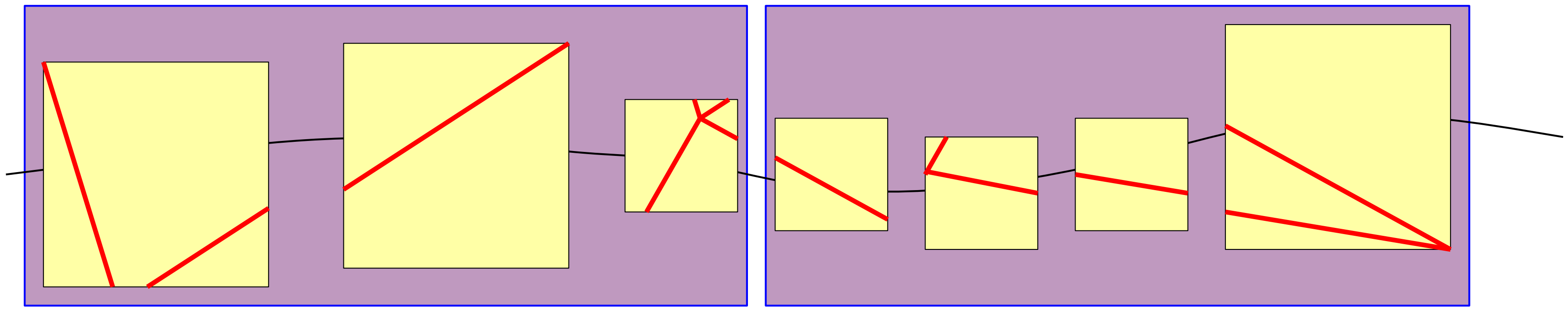
Map overlay with quadtrees in Z-order





each block is needed only once

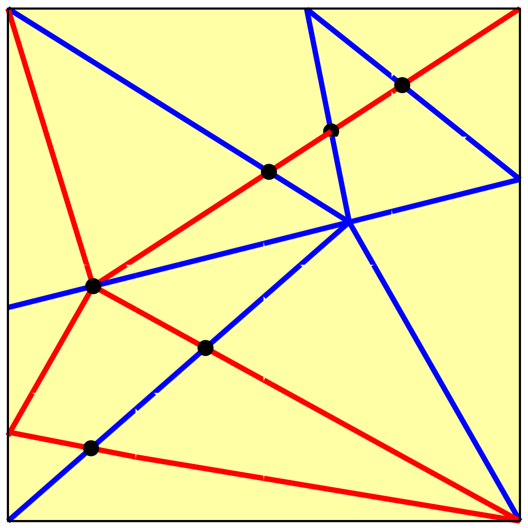




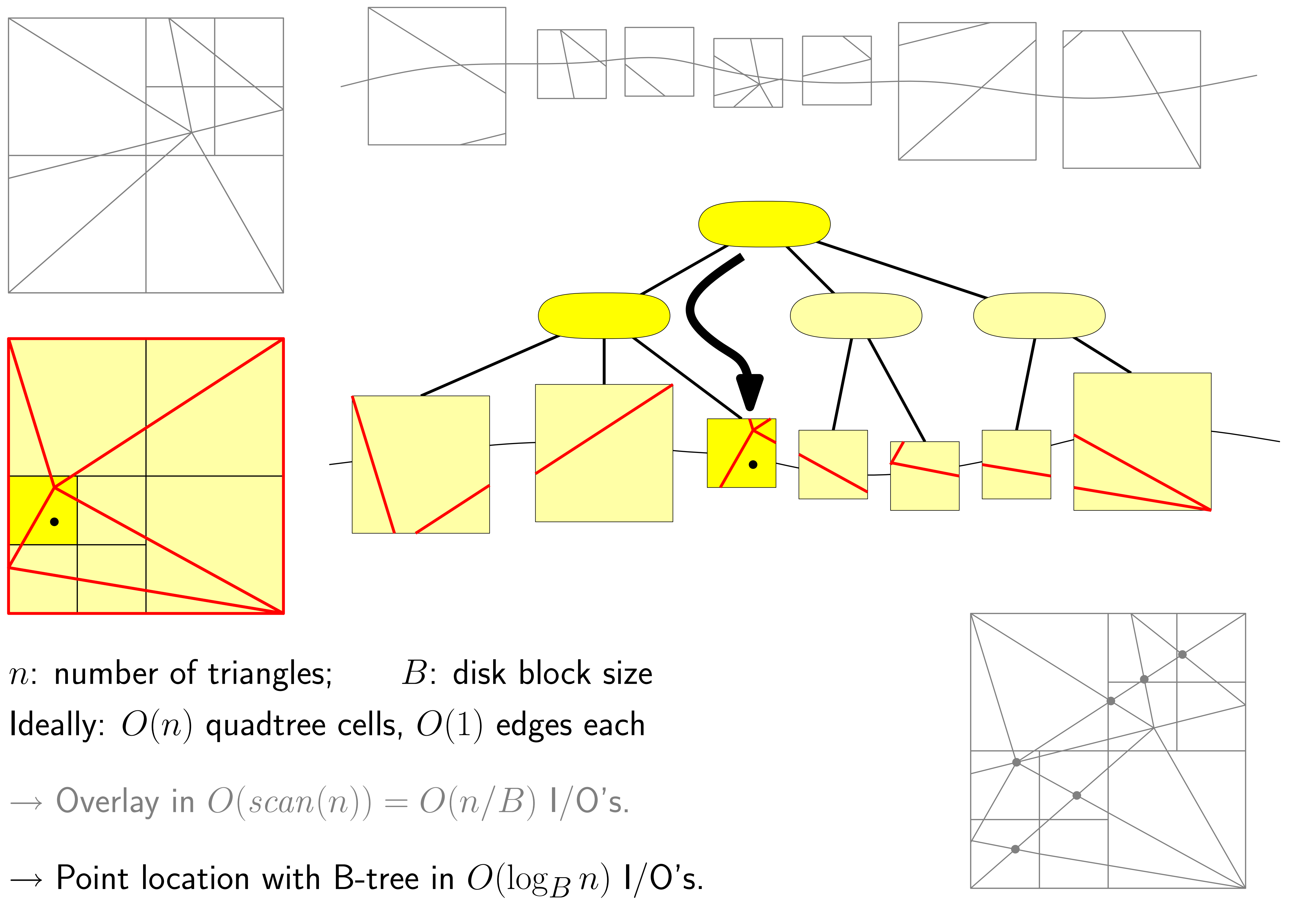
n: number of triangles; B: disk block size

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

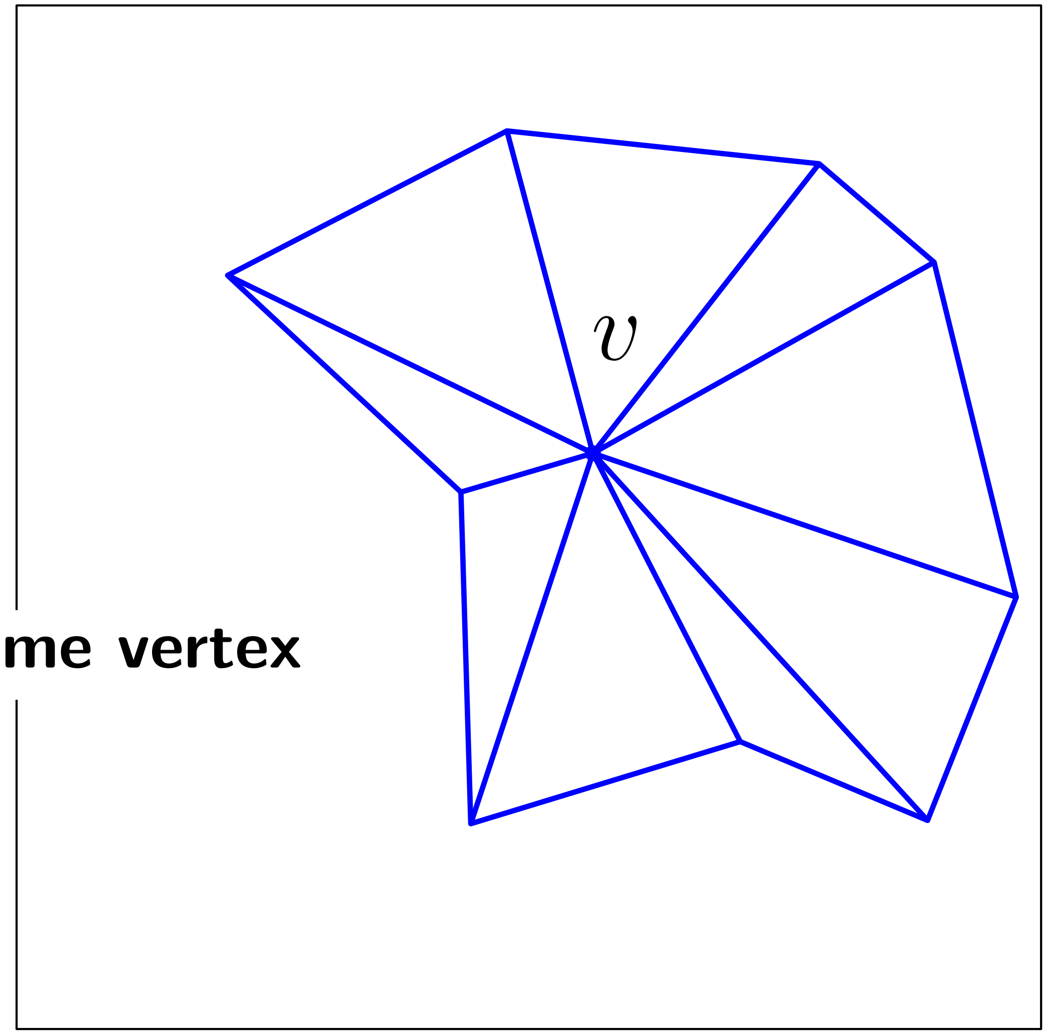
Ideally: $O(n)$ quadtree cells, $O(1)$ edges each

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

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

How to get that quadtree in Z-order (for fat 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 leaves into Z-order (removing duplicates)

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

Input: file with for each vertex its adjacency list.

Algorithm:

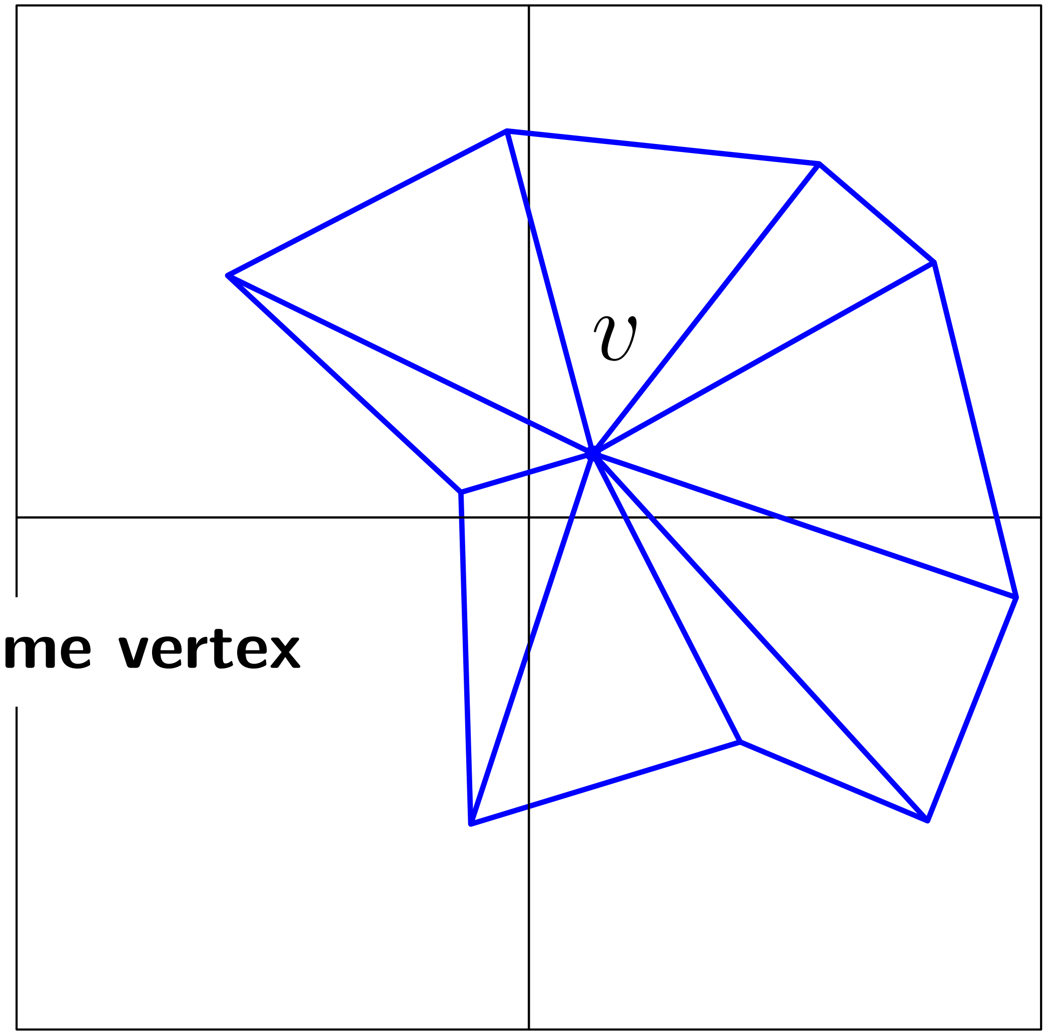
1. For each vertex v:

   - 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 leaves into Z-order (removing duplicates)



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

Input: file with for each vertex its adjacency list.

Algorithm:

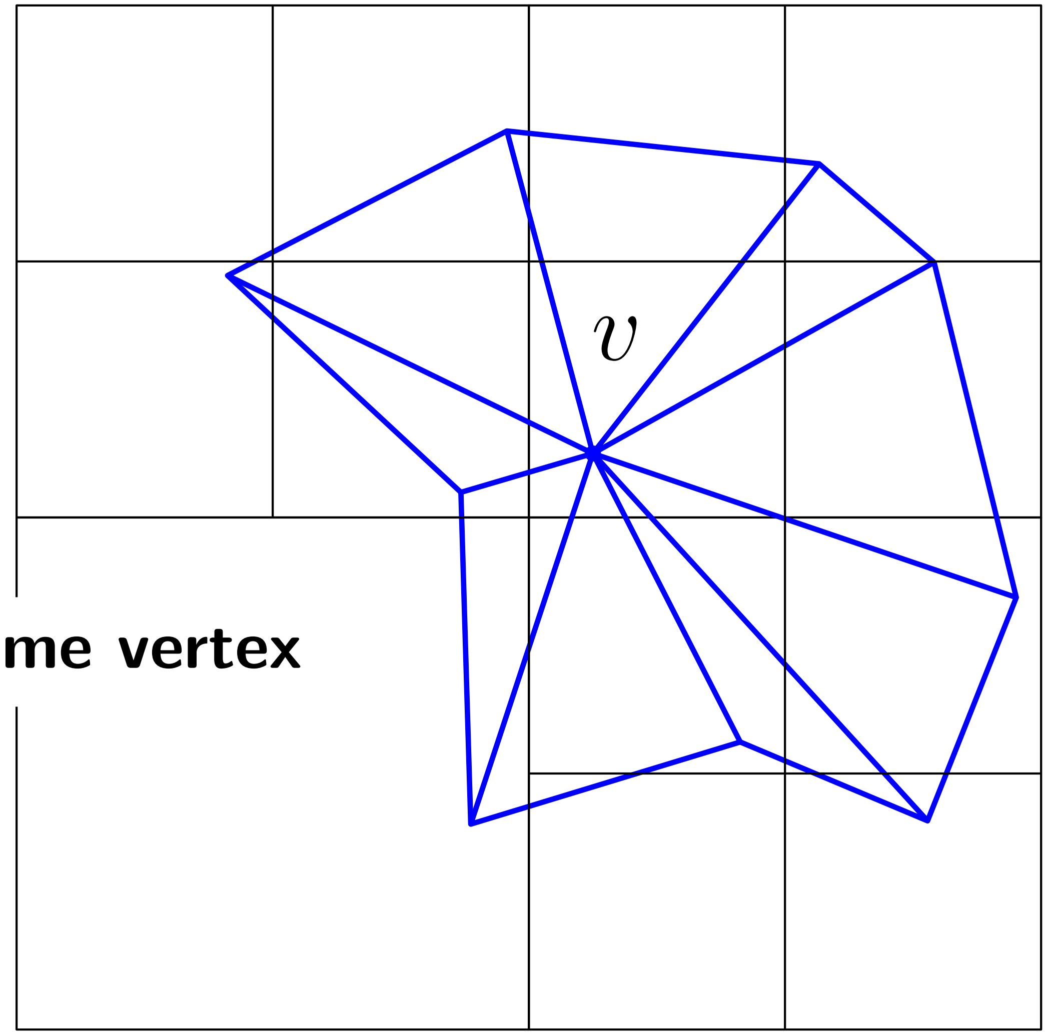
1. For each vertex v:

   - 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 leaves into Z-order (removing duplicates)



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

Input: file with for each vertex its adjacency list.

Algorithm:

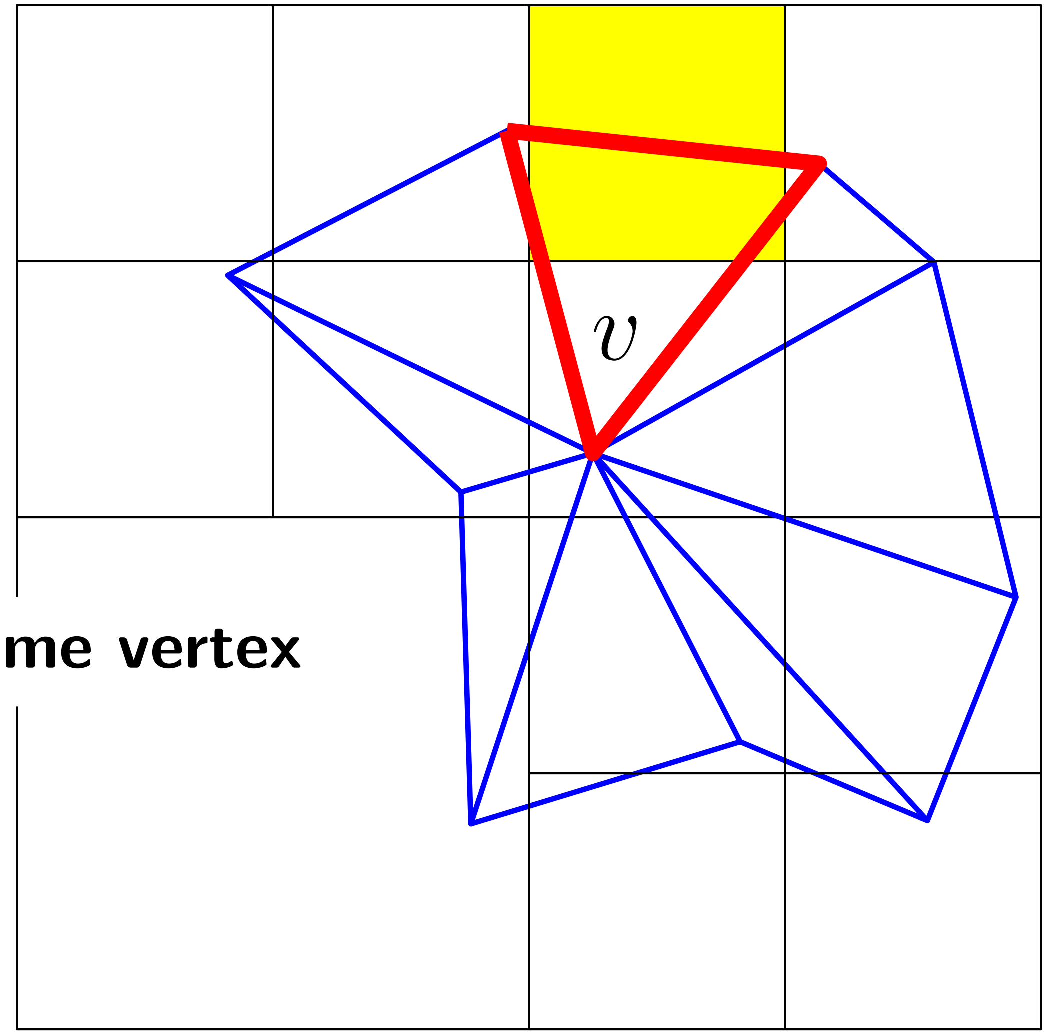
1. For each vertex v:

   - 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 leaves into Z-order (removing duplicates)



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

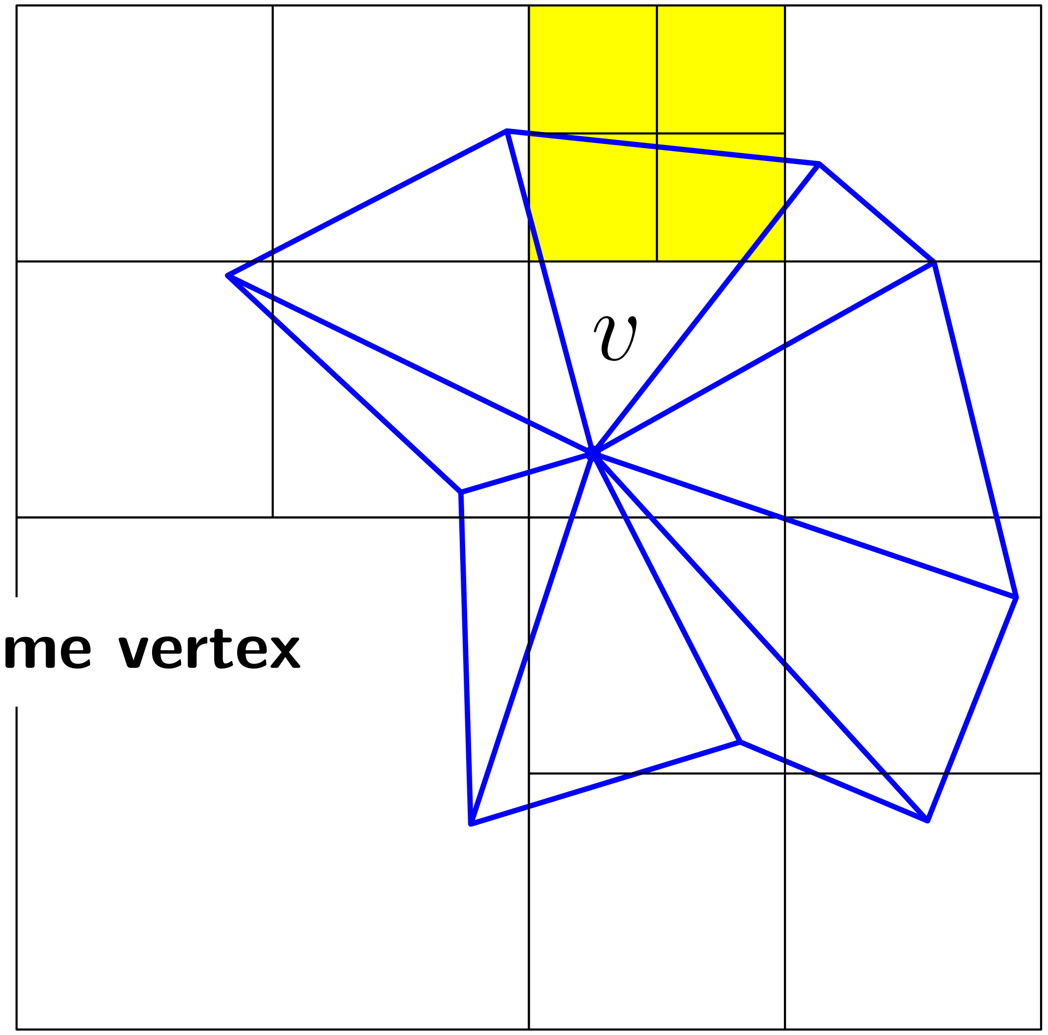
Input: file with for each vertex its adjacency list.

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 leaves into Z-order (removing duplicates)



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

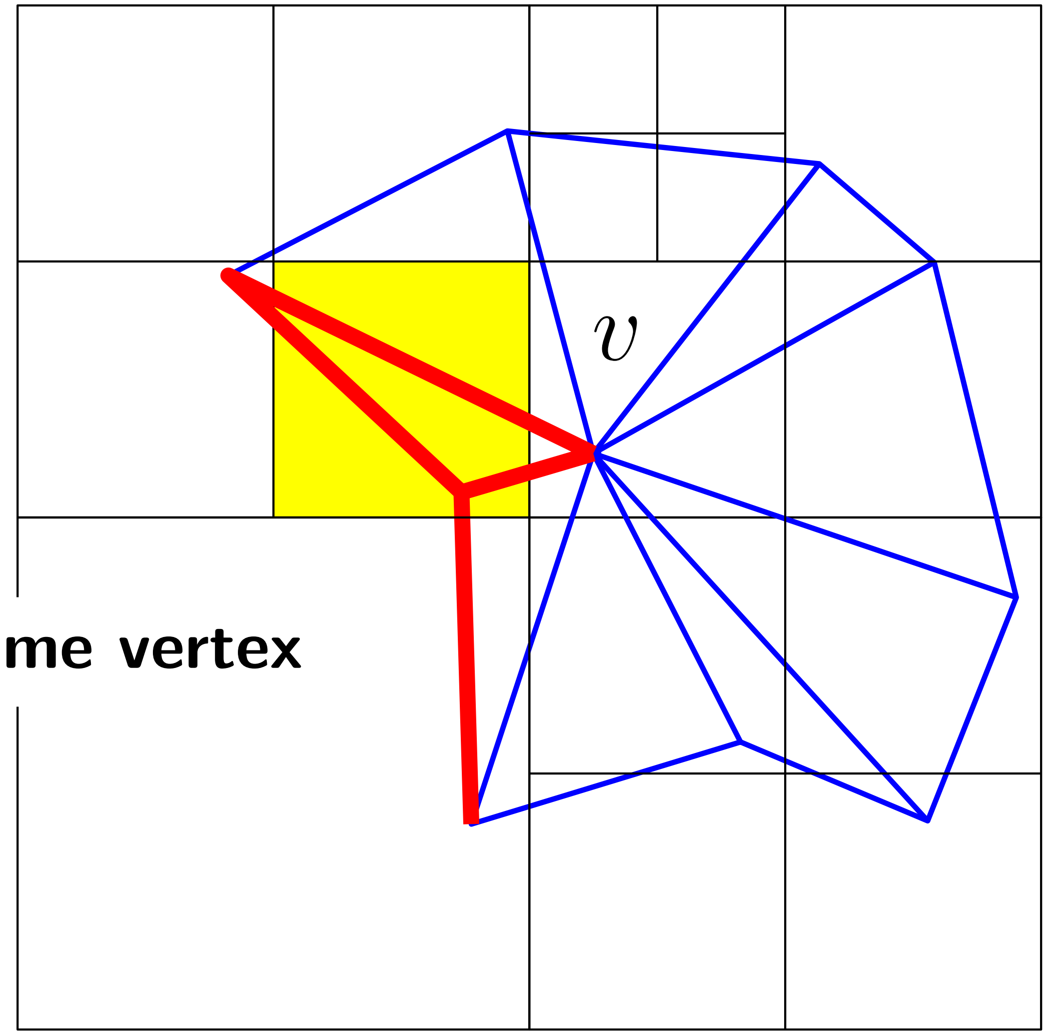
Input: file with for each vertex its adjacency list.

Algorithm:

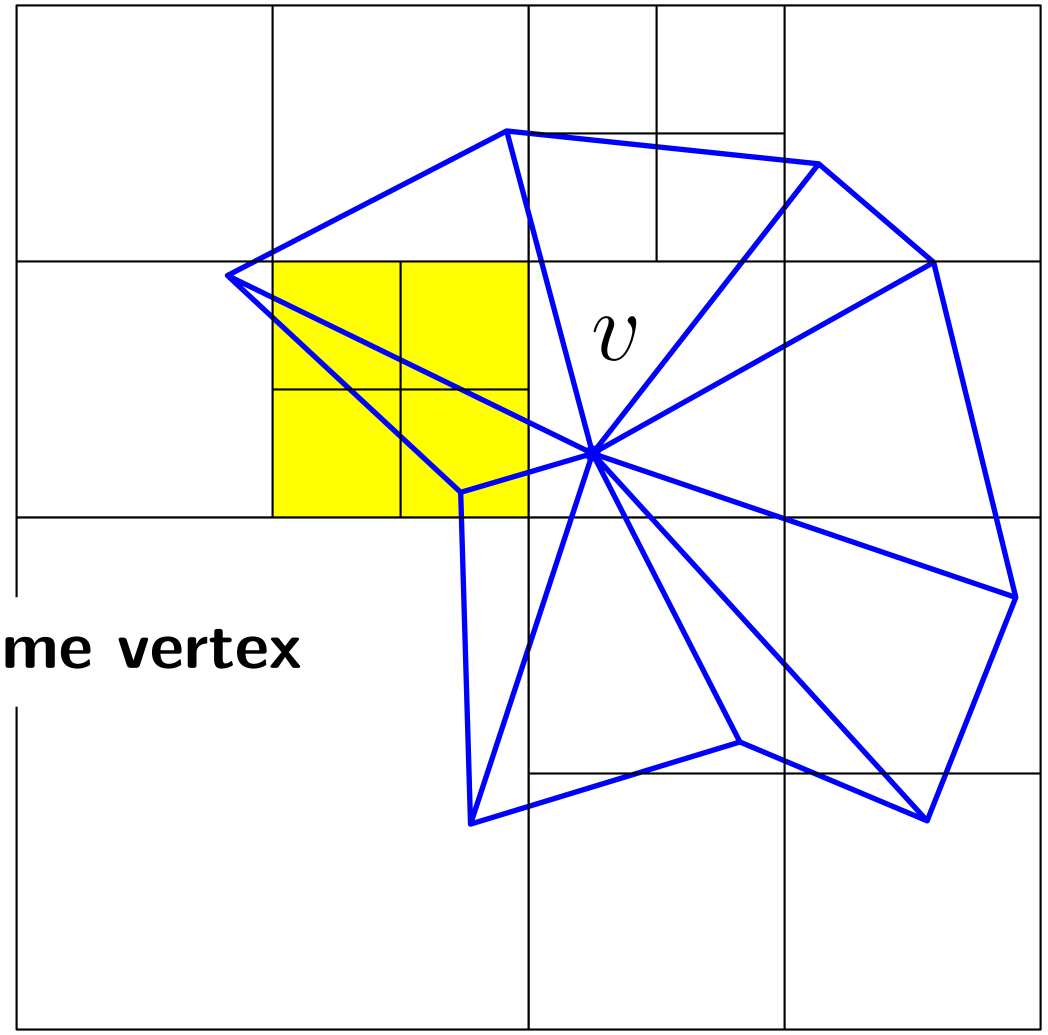
1. For each vertex v:
   - load adjacency list in memory;
   - build quadtree on $star(v)$ with splitting criterion:

     Stop splitting when all edges incident to same vertex

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



How to get that quadtree in Z-order (for fat 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 leaves into Z-order (removing duplicates)

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

Input: file with for each vertex its adjacency list.

Algorithm:

1. For each vertex v:

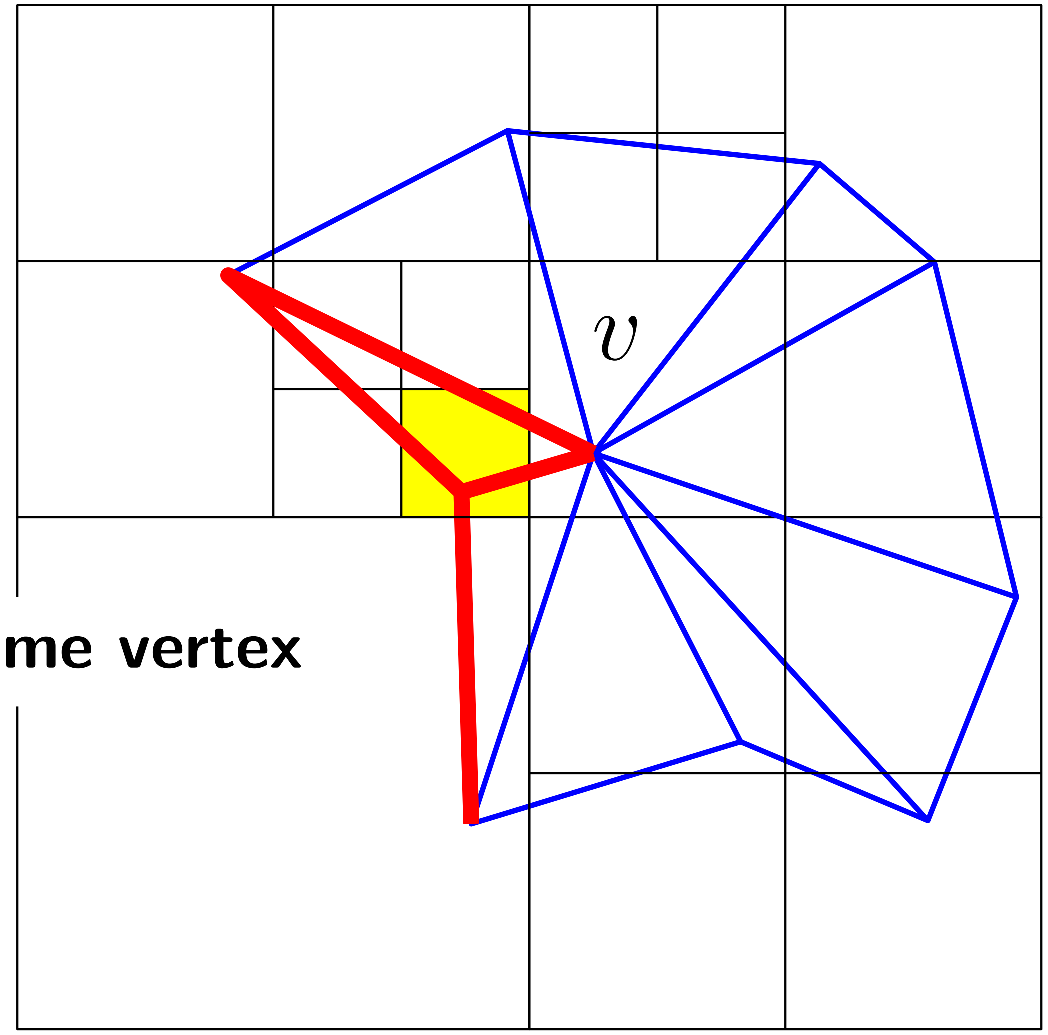
   • load adjacency list in memory;

   • build quadtree on $star(v)$ with splitting criterion:

   Stop splitting when all edges incident to same vertex

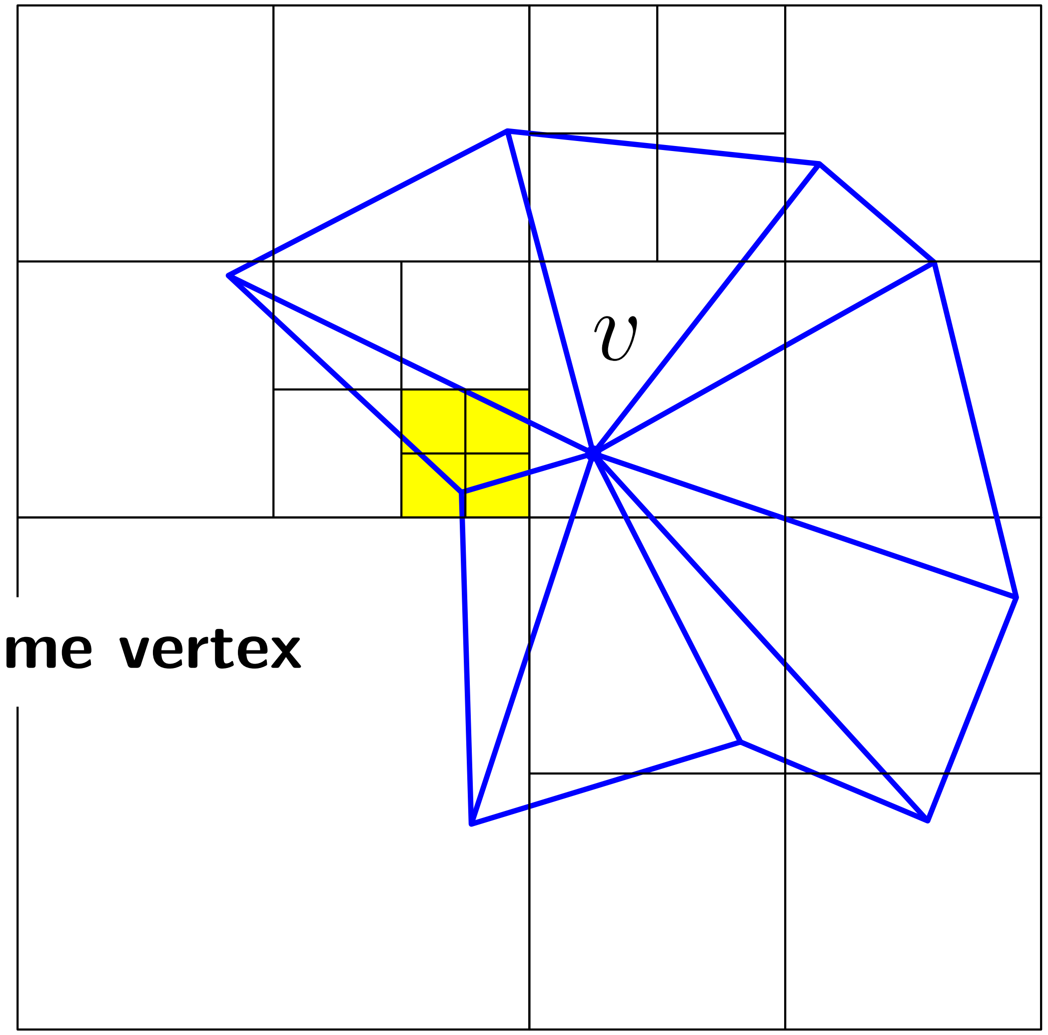
   • output each cell that is completely inside $star(v)$

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



How to get that quadtree in Z-order (for fat 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 leaves into Z-order (removing duplicates)

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

Input: file with for each vertex its adjacency list.

Algorithm:

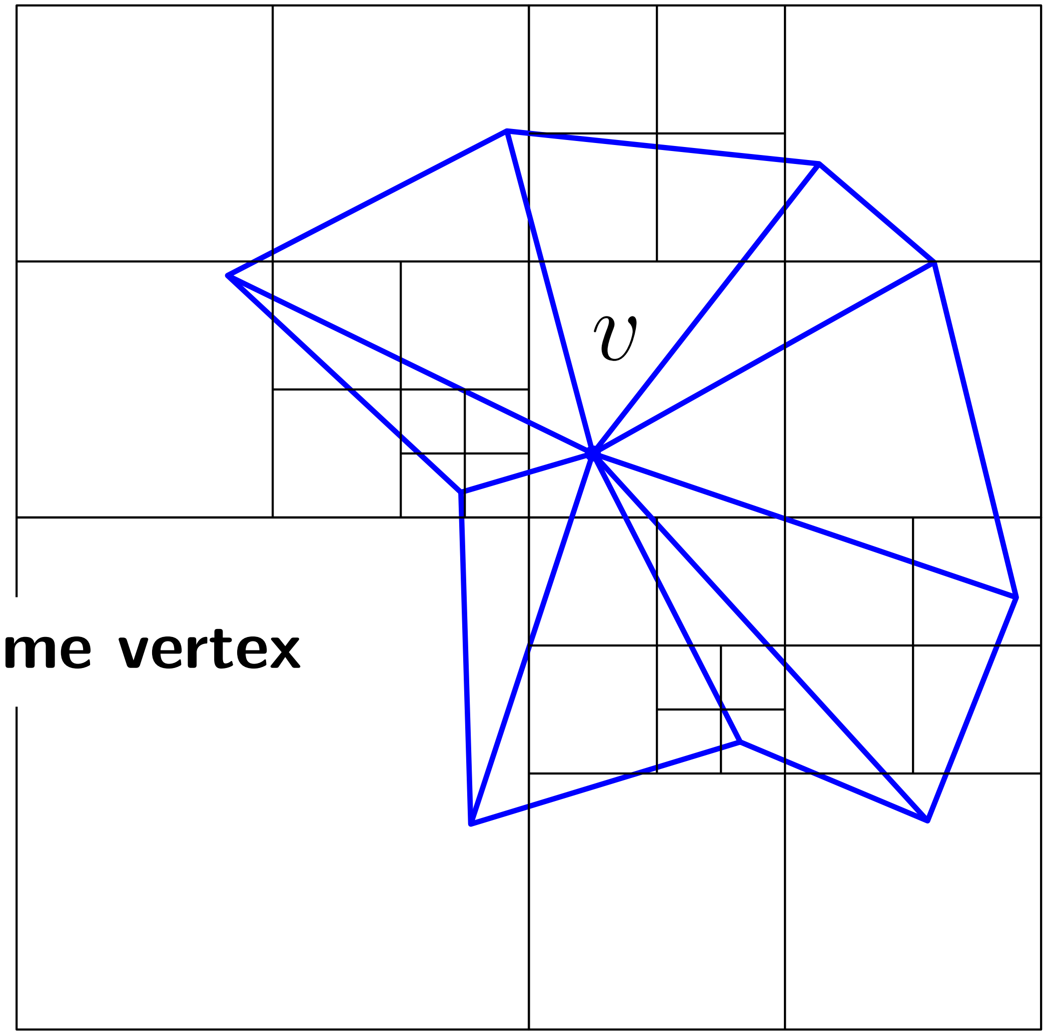
1. For each vertex v:

   - 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 leaves into Z-order (removing duplicates)



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

Input: file with for each vertex its adjacency list.

Algorithm:

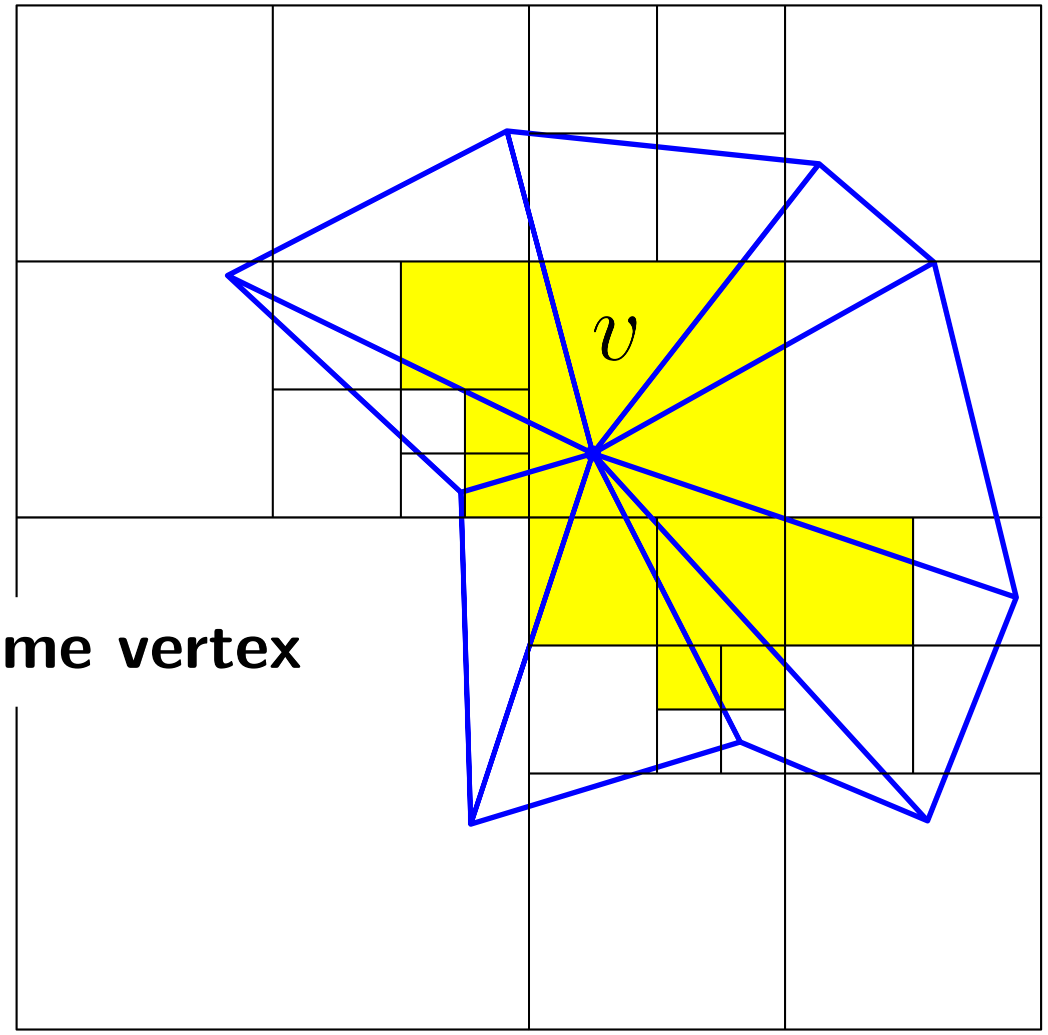
1. For each vertex v:

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

     Stop splitting when all edges incident to same vertex

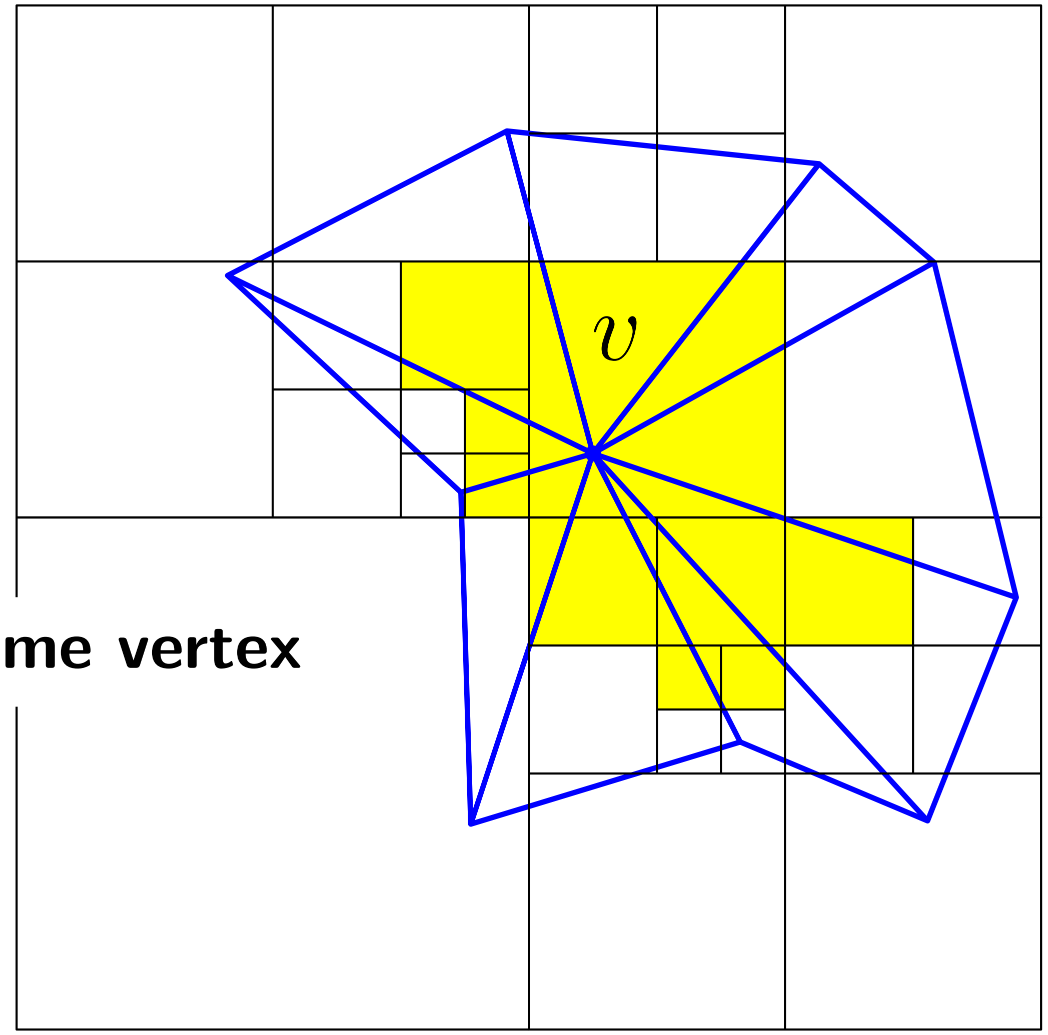
   - output each cell that is completely inside $star(v)$

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



How to get that quadtree in Z-order (for fat 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

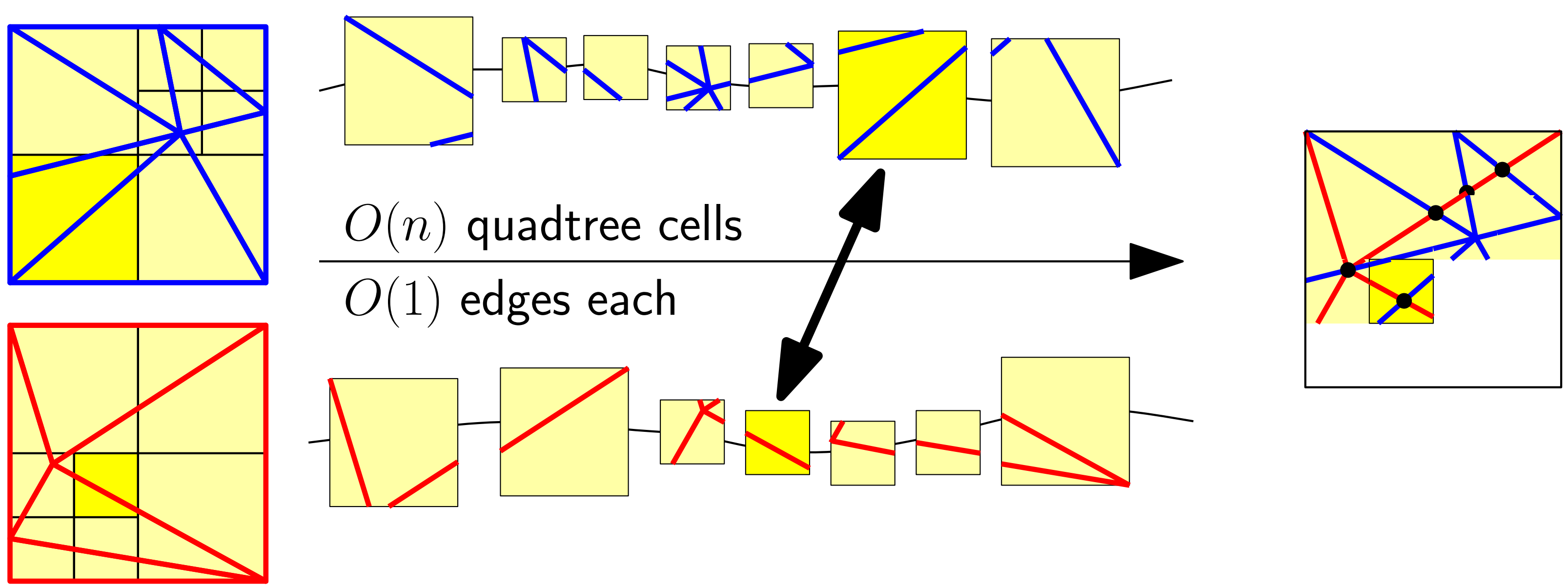
   - output each cell that is completely inside $star(v)$
2. Sort leaves into Z-order (removing duplicates)

To prove (assuming input is n fat 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

I/O-Efficient Map Overlay and Point Location on Low-Density Planar Maps

Mark de Berg   Herman Haverkort   Shripad Thite   Laura Toma



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:

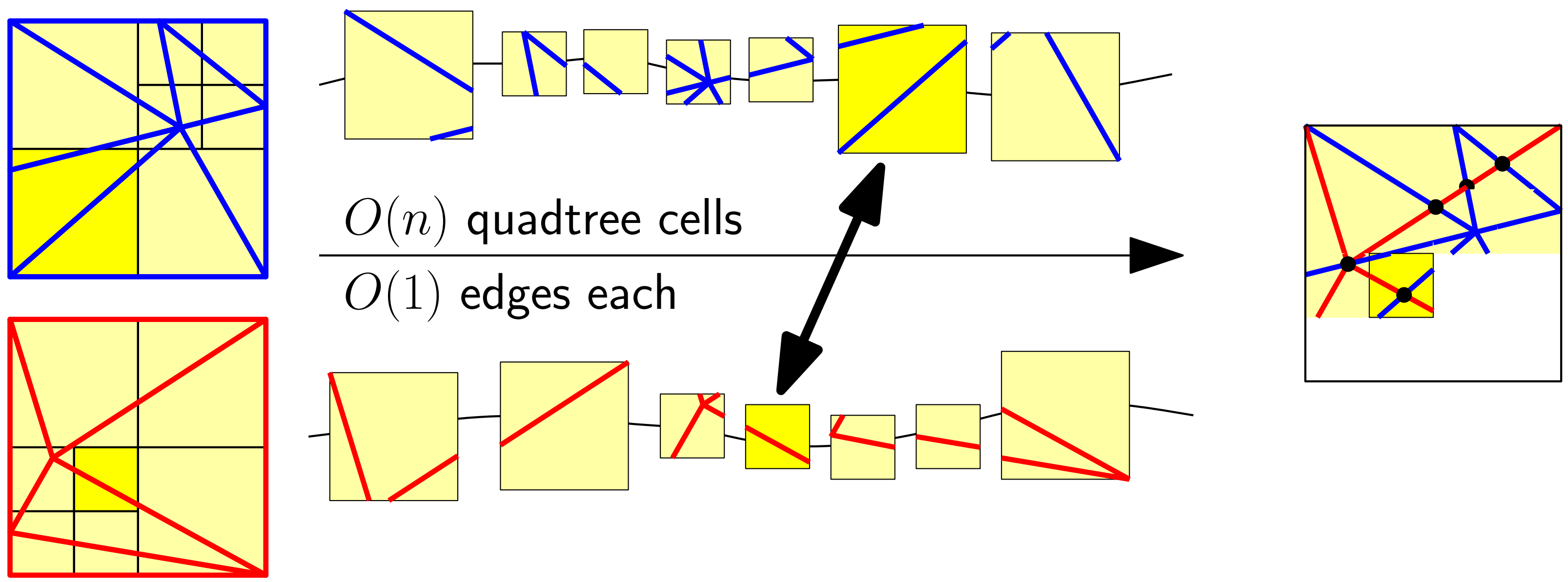
- map overlay in $O(scan(n))$ I/O's;
- point location in $O(\log_B n)$ I/O's;
- for triangulations: basic updates in $O(\log_B n)$ I/O's.

The data structure is built with $O(sort(n))$ I/Os.

*) for any square $\square$, number of intersecting segments bigger than $\square$ is at most a constant

I/O-Efficient Map Overlay and Point Location on Low-Density Planar Maps

Mark de Berg   Herman Haverkort   Shripad Thite   Laura Toma



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;
- point location in $O(\log_B n)$ I/O's;
- for triangulations: basic updates in $O(\log_B n)$ I/O's.

The data structure is built with $O(sort(n))$ I/Os.

That's all folks

*) for any square □, number of intersecting segments bigger than □ is at most a constant