Computing Visibility on Terrains in External Memory

Herman Haverkort   Laura Toma   Yi Zhuang

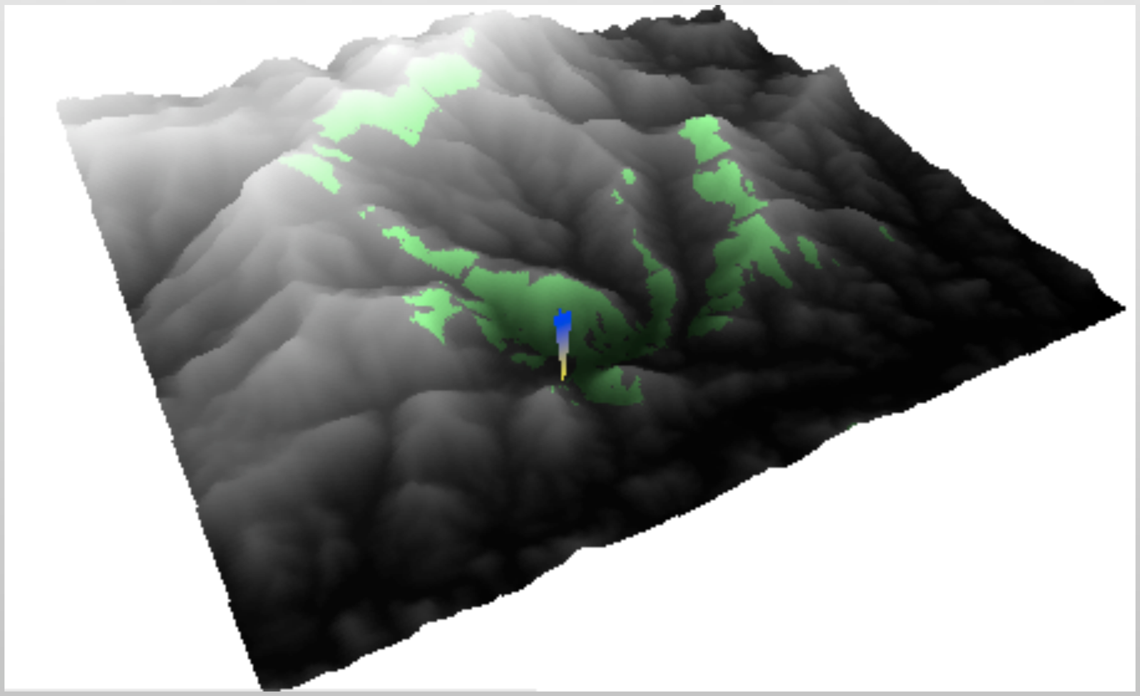
TU. Eindhoven
Netherlands

Bowdoin College
USA

ALENEX 2007
New Orleans, USA

Visibility

- Problem: visibility map (viewshed) of v
  - terrain T
  - arbitrary viewpoint v
  - the set of points in T visible from v


Sierra Nevada, 30m resolution

This project addresses the problem of computing efficiently visibility on terrains in external memory.

Given an arbitrary viewpoint v, compute the set of points in the terrain that are visible from v.

This is called the viewshed (GIS terms) or visibility map (geometric terms) of v.

An example is sh. in the picture -- the viewpoint is depicted with a blue arrow, and the visible part is drawn in green.

Visibility

- Problem: visibility map (viewshed) of v
  - terrain T
  - arbitrary viewpoint v
  - the set of points in T visible from v

- Applications
  - graphics
  - games
  - GIS
    - military applications, path planning, navigation
    - placement of fire towers, radar sites, cell phone towers (terrain guarding)

Computing visibility comes up and is important in many applications , ranging from graphics, game design to military applications, navigation, path planning , and variations of terrain guarding problems, like placement of radar sites and phone towers.

Massive terrains

- Why massive terrains?
  - Large amounts of data are becoming available
    - NASA SRTM project: 30m resolution over the entire globe (~10TB)
    - LIDAR data: sub-meter resolution
- Traditional algorithms don't scale
  - Buy more RAM?
    - Data grows faster than memory
  - Data on disk
  - Disks are MUCH slower than memory
- => I/O-bottleneck

We are interested in computing visibility efficiently on very large terrains.

Why? because massive data has become widely available ---- for e.g. SRTM collected 10TB data, LIDAR provide for meter resotution.

Thus the areas that we handle in GIS are larger and larger, and traditional algorithms, designed in the standard RAM model, do not scale. Data does not fit in memory, sits on disk, and, since disks are MUCH slower than CPU, the bottleneck is the I/O---that is, teh data transfer between main memory and disk.

I/O-efficient algorithms

- I/O-model [AV'88]
  - data on disk, arranged in blocks
  - I/O-operation = reading/writing one block from/to disk

n=grid size   M=memory size   B=block size

- I/O-efficiency: nb. I/O-operations

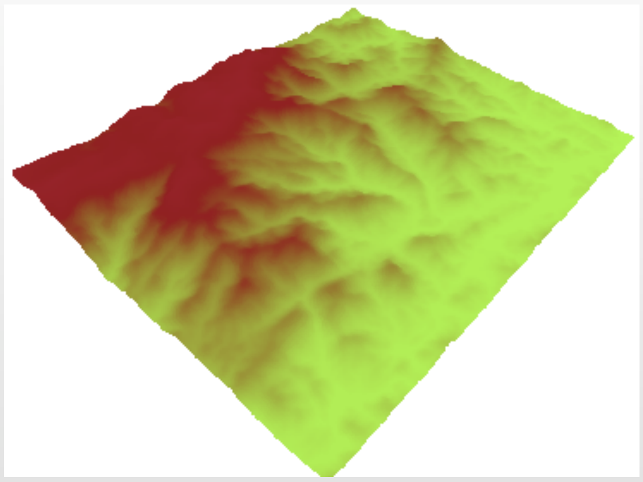
- Basic I/O bounds

$$\mathrm{scan}(n) = \Theta\left(\frac{n}{B}\right) \quad < \quad \mathrm{sort}(n) = \Theta\left(\frac{n}{B}\log_{M/B} n/M\right) \quad \ll \quad n$$

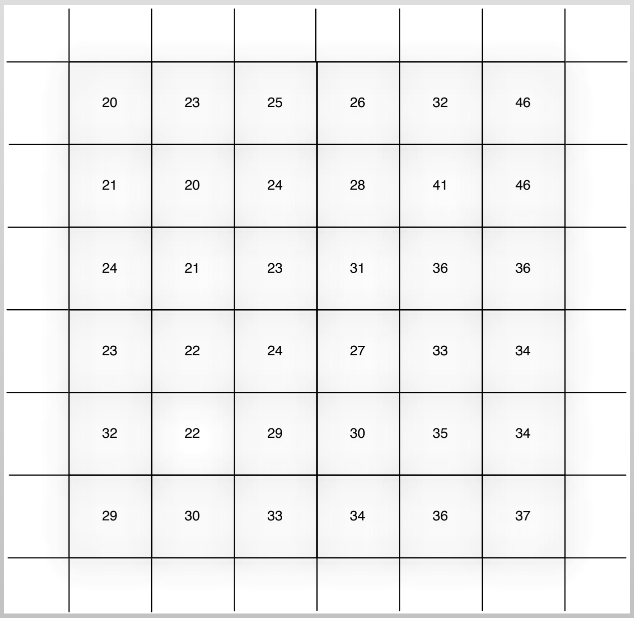
In order to compute efficiently on large data one must optimize not only the CPU, but also the I/O. We use the now standard model for designing I/O-efficient algorithms introduced by A&V. The model assumes data sits on disks arranged in blocks. In order to compute on an item the algorithm must load the corresponding block in memory. This is an I/O operation --- reading or writing one block of data from/to disk.

Notation: M memory size, B block size, n input size. scanning .. sorting.

Terrain data

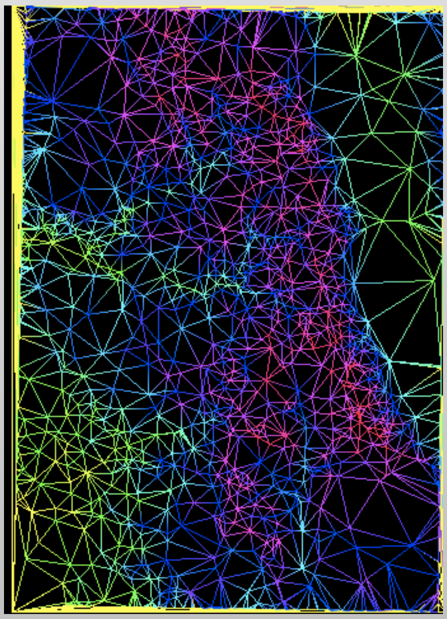


- Most often: grid terrain



20	23	25	26	32	46
21	20	24	28	41	46
24	21	23	31	36	36
23	22	24	27	33	34
32	22	29	30	35	34
29	30	33	34	36	37

- TIN (triangulated polyhedral terrain)



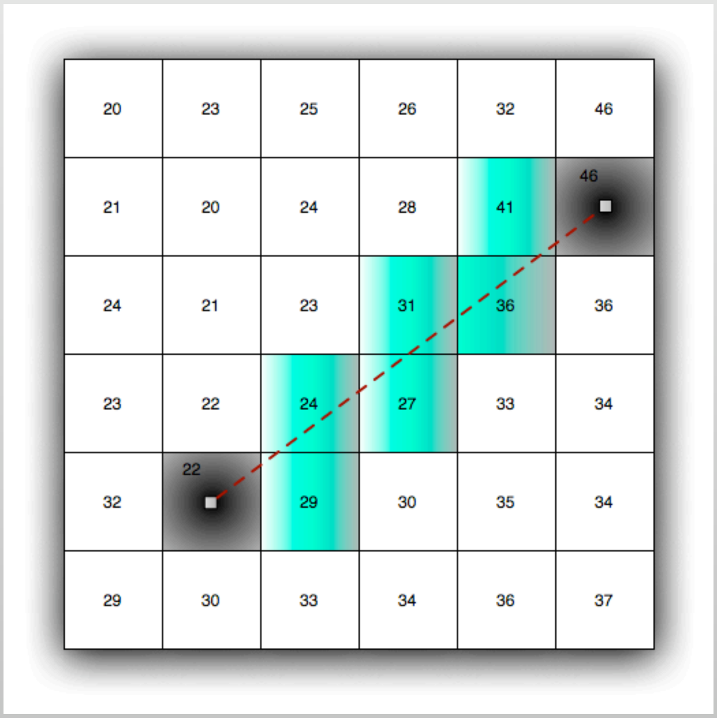
(So, we want to compute visibility on terrains I/O-efficiently)

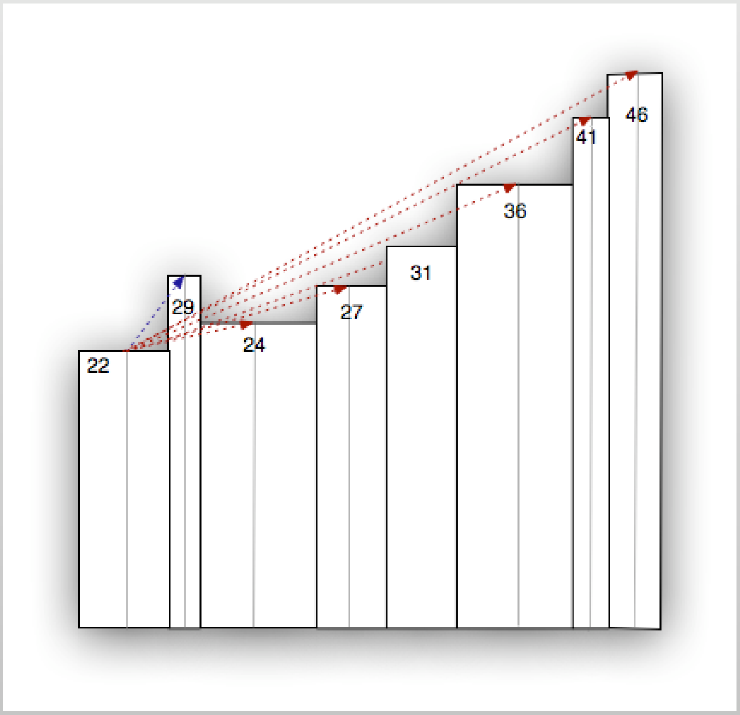
Terrains in GIS are represented mostly as grids --- these are matrices of elevation values sampled uniformly from the terrain.

Terrains can also be represented as TIN (triangular irregular network), or triangulated polyhedral terrain in geometric terms. Each representation has its pros and cons. Grids have uniform resolution and some redundancy, but so far they are the most widely used.

Visibility on grids

- line-of-sight model
  - a grid cell with center q is visible from viewpoint v iff the line segment vq does not cross any cell that is above vq





Assume we have a grid of elevations, and assume that each elevation represents the center of the cell.

Let v be an arbitrary viewpoint, for simplicity in the figure its in the center of one of the cells of the grid.
The LOS (line of sight) from v to a cell is the line from v to the center of the cell.

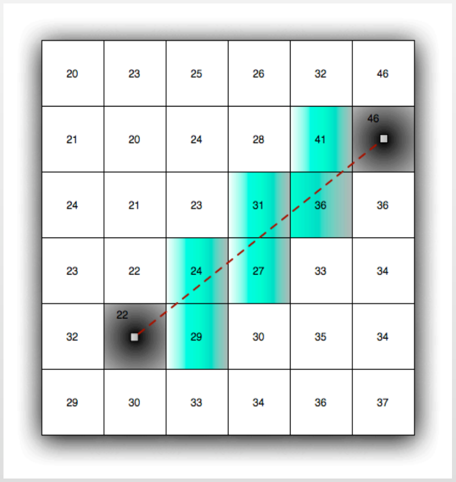
We say that a cell is visible from v iff the LOS from v to the cell does not cross any cell with higher slope ---
that is, if the LOS does not intersect any other grid cell q' such that slope of vq' is higher than slope vq.

(see example)

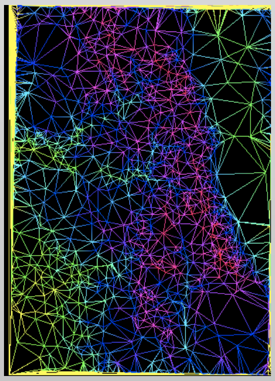
(probably no time for this)
note: this model was introduced by van Kreveld, and was presumably used in related work as well. it is a discrete model , where a cell is either visible or not --- no partial visibility. a corner of a high cell can hide another cell completely.

Visibility: Related work

- grids
  - straightforward algorithm $O(n^2)$
  - $O(n \lg n)$ by van Kreveld
  - experimental
    - Fisher [F93, F94], Franklin & Ray [FR94], Franklin [F02]
    - no worst-case guarantees



- TINs
  - surveys: de Floriani & Magillo [FM94], Cole & Sharir [CS89]
  - recently: watchtowers and terrain guarding [SoCG'05, SODA'06]



(go very fast on this)

Given a viewpoint, we want to compute a visibility grid, which for each cell it records whether it is visible or not (and if not, its vertical distance to visibility).

To compute whether two points are visible takes O(n) (O(\sqrt n) on a square grid), and thus the straightforward way to compute the visibility grid takes O(n^2) (O(n ^{3/2}) on a square grid).
This was improved to nlgn by van Kreveld.
There are quite a few experimental results on visibility ffrom the GIS community, they dont give worst-case guarantees.

Surveys of visibility results on TINs are given by de Floriani and Magillo, and Cole and Sharir. Various variations of the problems have been studied, most recently there are results on watchtowers and terrain guarding.

van Kreveld's algorithm

remember that to compute if two points are visible we need to find all cells intersected by the LOS and determine if any slope is higher than the slope of the LOS.

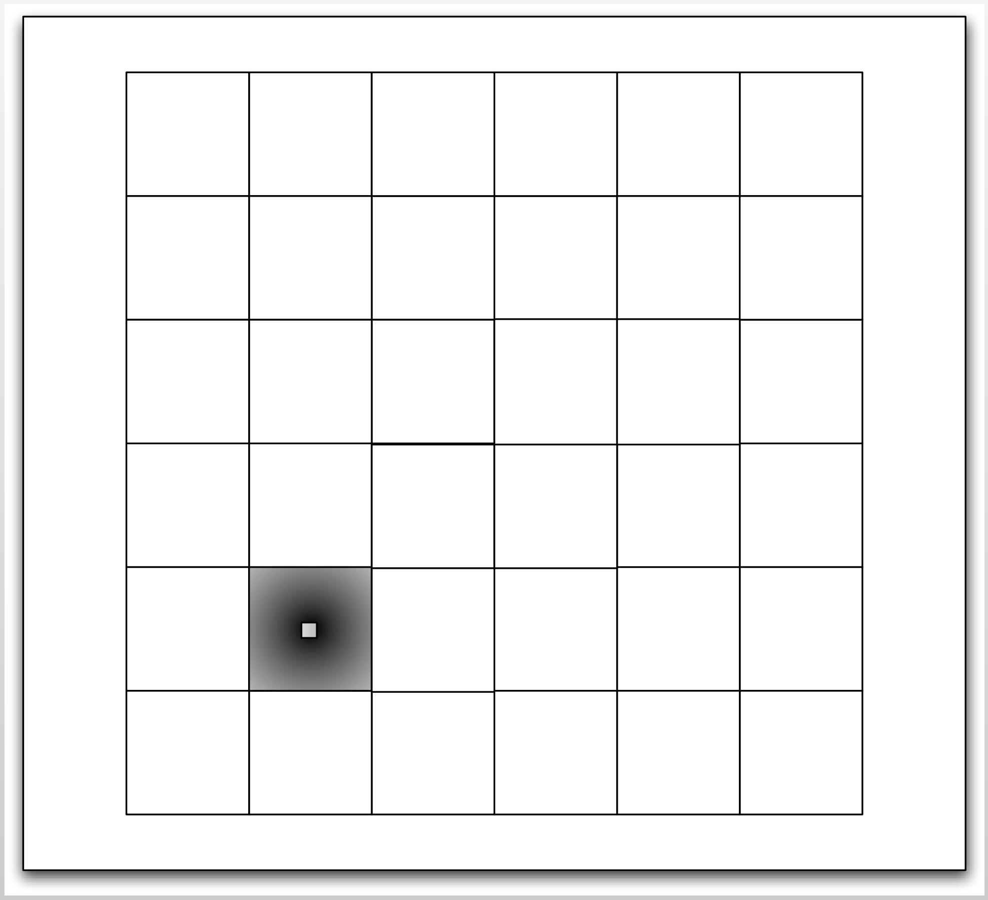
Kreveld's idea is to use a line sweep, rotating around the viewpoint, and to compute the visibility of a cell when the line passes over its center.
Each cell has 3 associated events: when it is first intersected by the line, when its center is intersected, and when it is last intersected by the sweep line.

The active structure maintains the cells intersected by the sweep line at any time. When a cell is intersected for the 1st time it is inserted in the active structure, when its intersected last it is deleted, and when its centerpoint is intersected the active str is queried to determine whether the cell is visible.

For this the active structure is a binary search tree, where cells intersected by the sweep line are stored in order of the distance of their centerpoint from v. Note that when a cell is queried, all cells that are intersected by the LOS are in the active structure, to the left of the cell. If we augment the tree we can find the maximum slope to the left of a point in lg n time.

time: 3n events, lg n time each -> n lg n.

van Kreveld's algorithm



remember that to compute if two points are visible we need to find all cells intersected by the LOS and determine if any slope is higher than the slope of the LOS.

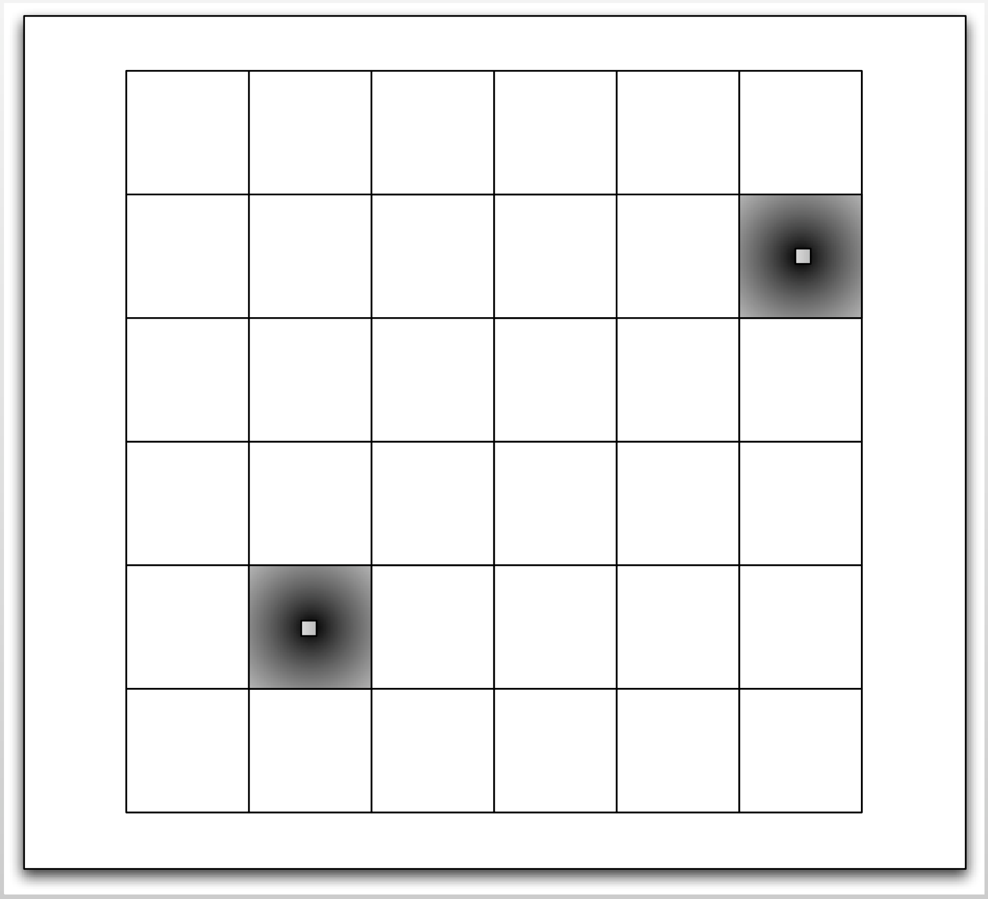
Kreveld's idea is to use a line sweep, rotating around the viewpoint, and to compute the visibility of a cell when the line passes over its center.
Each cell has 3 associated events: when it is first intersected by the line, when its center is intersected, and when it is last intersected by the sweep line.

The active structure maintains the cells intersected by the sweep line at any time. When a cell is intersected for the 1st time it is inserted in the active structure, when its intersected last it is deleted, and when its centerpoint is intersected the active str is queried to determine whether the cell is visible.

For this the active structure is a binary search tree, where cells intersected by the sweep line are stored in order of the distance of their centerpoint from v. Note that when a cell is queried, all cells that are intersected by the LOS are in the active structure, to the left of the cell. If we augment the tree we can find the maximum slope to the left of a point in lg n time.

time: 3n events, lg n time each -> n lg n.

van Kreveld's algorithm



remember that to compute if two points are visible we need to find all cells intersected by the LOS and determine if any slope is higher than the slope of the LOS.

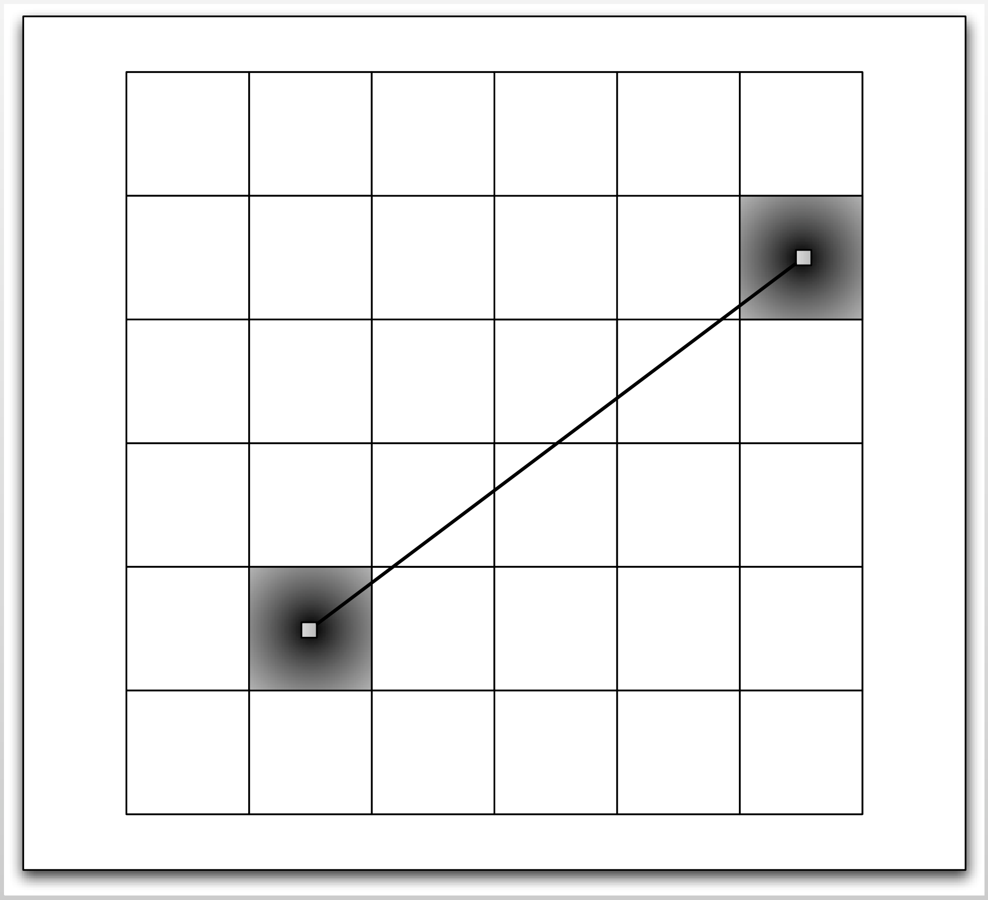
Kreveld's idea is to use a line sweep, rotating around the viewpoint, and to compute the visibility of a cell when the line passes over its center.
Each cell has 3 associated events: when it is first intersected by the line, when its center is intersected, and when it is last intersected by the sweep line.

The active structure maintains the cells intersected by the sweep line at any time. When a cell is intersected for the 1st time it is inserted in the active structure, when its intersected last it is deleted, and when its centerpoint is intersected the active str is queried to determine whether the cell is visible.

For this the active structure is a binary search tree, where cells intersected by the sweep line are stored in order of the distance of their centerpoint from v. Note that when a cell is queried, all cells that are intersected by the LOS are in the active structure, to the left of the cell. If we augment the tree we can find the maximum slope to the left of a point in lg n time.

time: 3n events, lg n time each -> n lg n.

van Kreveld's algorithm



remember that to compute if two points are visible we need to find all cells intersected by the LOS and determine if any slope is higher than the slope of the LOS.

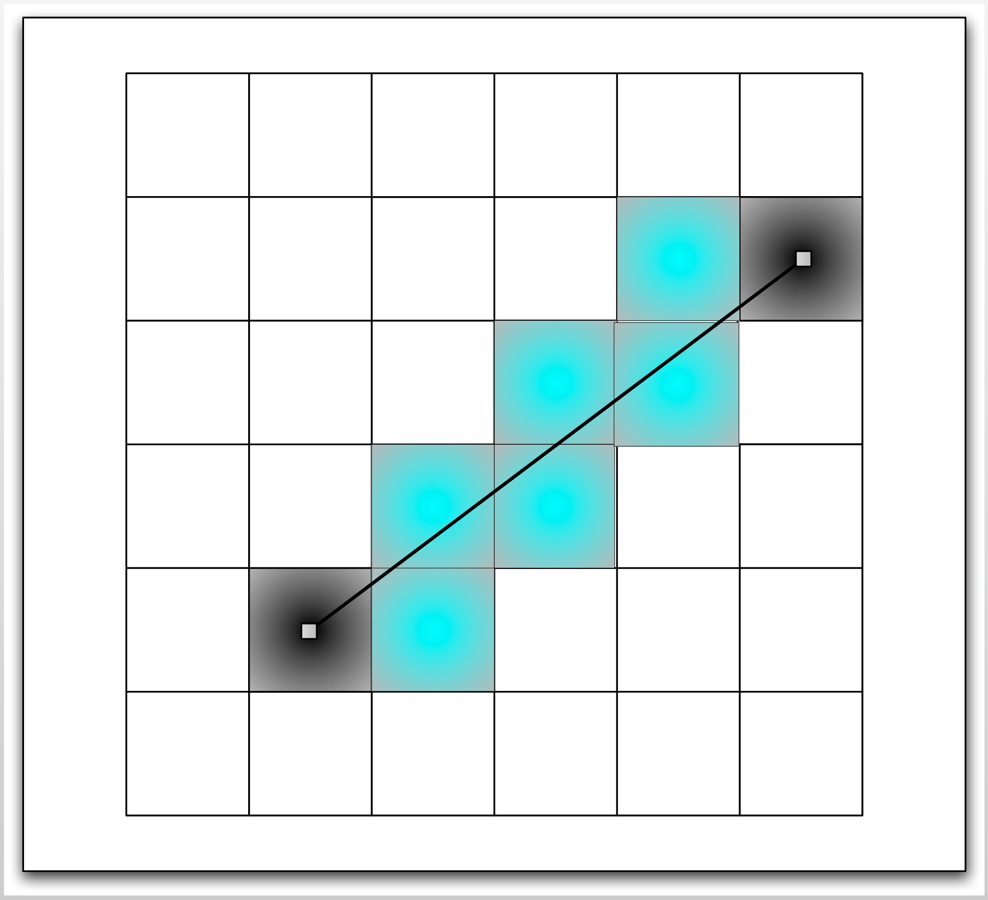
Kreveld's idea is to use a line sweep, rotating around the viewpoint, and to compute the visibility of a cell when the line passes over its center.
Each cell has 3 associated events: when it is first intersected by the line, when its center is intersected, and when it is last intersected by the sweep line.

The active structure maintains the cells intersected by the sweep line at any time. When a cell is intersected for the 1st time it is inserted in the active structure, when its intersected last it is deleted, and when its centerpoint is intersected the active str is queried to determine whether the cell is visible.

For this the active structure is a binary search tree, where cells intersected by the sweep line are stored in order of the distance of their centerpoint from v. Note that when a cell is queried, all cells that are intersected by the LOS are in the active structure, to the left of the cell. If we augment the tree we can find the maximum slope to the left of a point in lg n time.

time: 3n events, lg n time each -> n lg n.

van Kreveld's algorithm



remember that to compute if two points are visible we need to find all cells intersected by the LOS and determine if any slope is higher than the slope of the LOS.

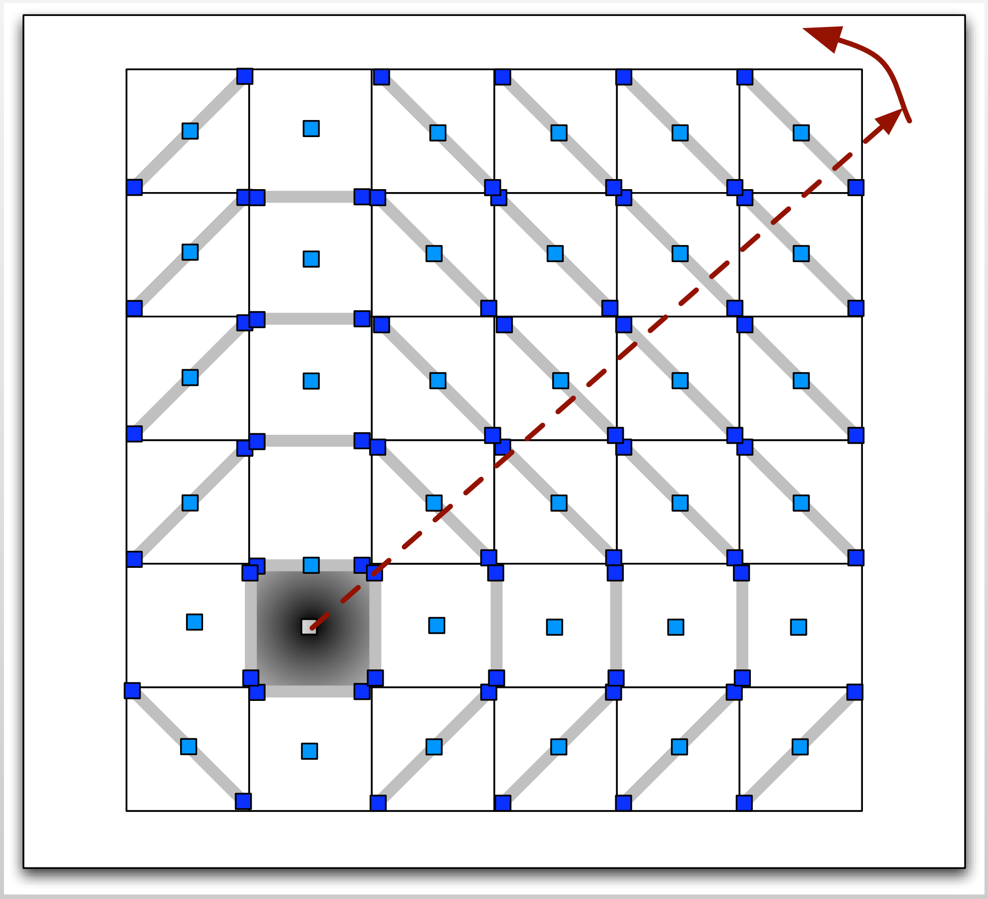
Kreveld's idea is to use a line sweep, rotating around the viewpoint, and to compute the visibility of a cell when the line passes over its center.
Each cell has 3 associated events: when it is first intersected by the line, when its center is intersected, and when it is last intersected by the sweep line.

The active structure maintains the cells intersected by the sweep line at any time. When a cell is intersected for the 1st time it is inserted in the active structure, when its intersected last it is deleted, and when its centerpoint is intersected the active str is queried to determine whether the cell is visible.

For this the active structure is a binary search tree, where cells intersected by the sweep line are stored in order of the distance of their centerpoint from v. Note that when a cell is queried, all cells that are intersected by the LOS are in the active structure, to the left of the cell. If we augment the tree we can find the maximum slope to the left of a point in lg n time.

time: 3n events, lg n time each -> n lg n.

van Kreveld's algorithm



remember that to compute if two points are visible we need to find all cells intersected by the LOS and determine if any slope is higher than the slope of the LOS.

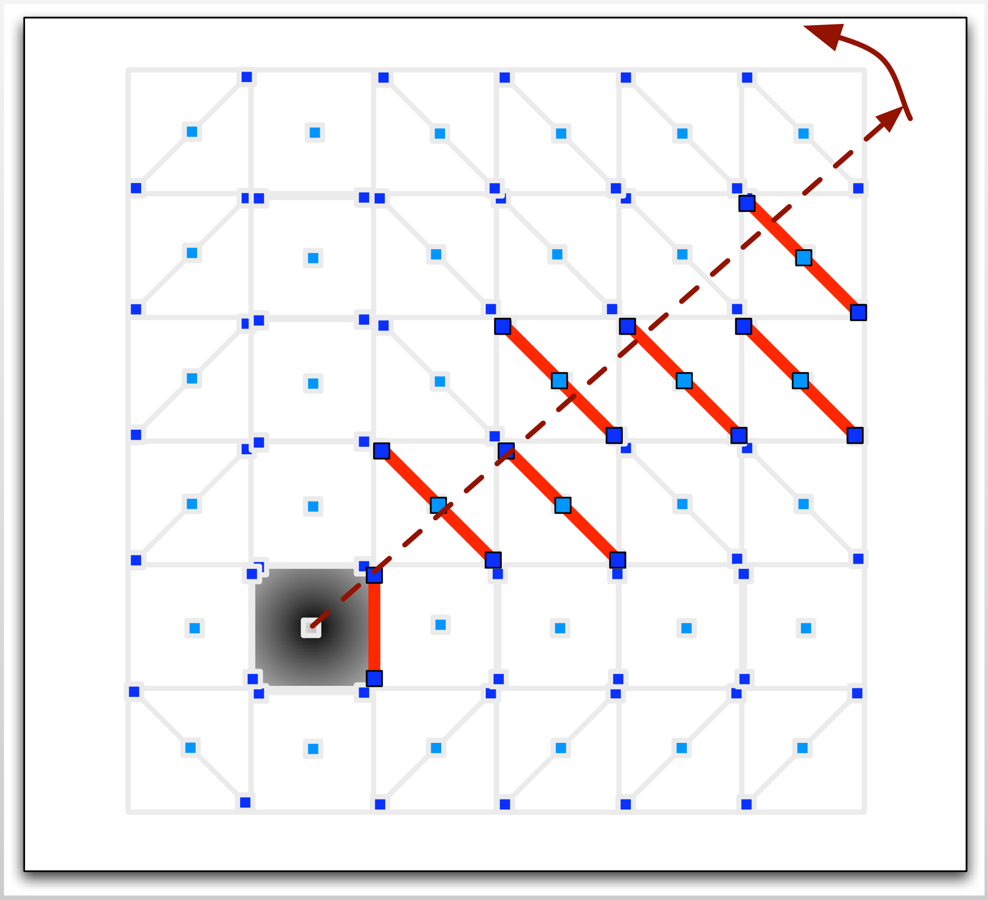
Kreveld's idea is to use a line sweep, rotating around the viewpoint, and to compute the visibility of a cell when the line passes over its center.
Each cell has 3 associated events: when it is first intersected by the line, when its center is intersected, and when it is last intersected by the sweep line.

The active structure maintains the cells intersected by the sweep line at any time. When a cell is intersected for the 1st time it is inserted in the active structure, when its intersected last it is deleted, and when its centerpoint is intersected the active str is queried to determine whether the cell is visible.

For this the active structure is a binary search tree, where cells intersected by the sweep line are stored in order of the distance of their centerpoint from v. Note that when a cell is queried, all cells that are intersected by the LOS are in the active structure, to the left of the cell. If we augment the tree we can find the maximum slope to the left of a point in lg n time.

time: 3n events, lg n time each -> n lg n.

van Kreveld's algorithm



remember that to compute if two points are visible we need to find all cells intersected by the LOS and determine if any slope is higher than the slope of the LOS.

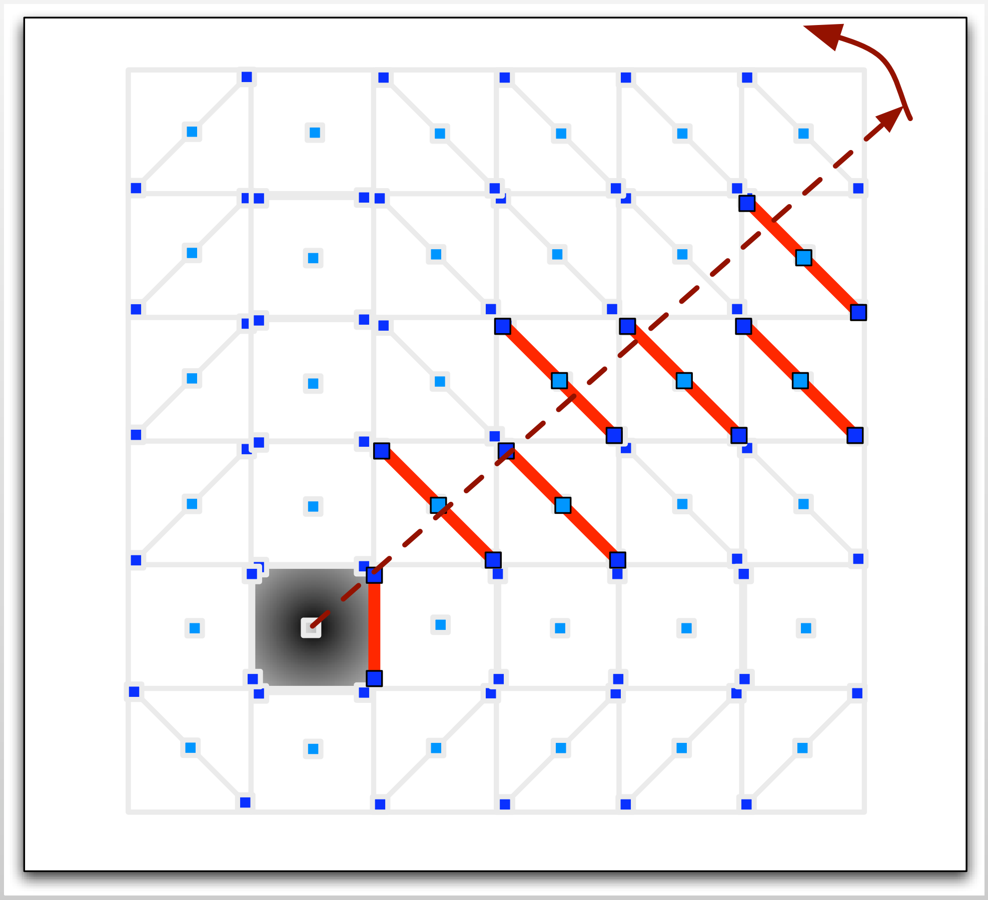
Kreveld's idea is to use a line sweep, rotating around the viewpoint, and to compute the visibility of a cell when the line passes over its center.
Each cell has 3 associated events: when it is first intersected by the line, when its center is intersected, and when it is last intersected by the sweep line.

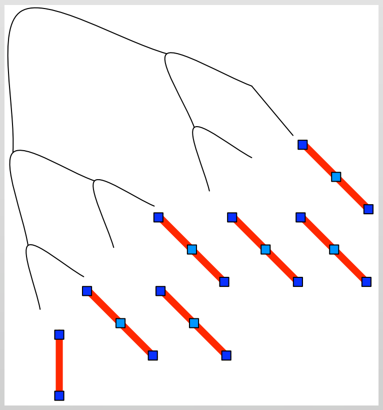
The active structure maintains the cells intersected by the sweep line at any time. When a cell is intersected for the 1st time it is inserted in the active structure, when its intersected last it is deleted, and when its centerpoint is intersected the active str is queried to determine whether the cell is visible.

For this the active structure is a binary search tree, where cells intersected by the sweep line are stored in order of the distance of their centerpoint from v. Note that when a cell is queried, all cells that are intersected by the LOS are in the active structure, to the left of the cell. If we augment the tree we can find the maximum slope to the left of a point in lg n time.

time: 3n events, lg n time each -> n lg n.

van Kreveld's algorithm





remember that to compute if two points are visible we need to find all cells intersected by the LOS and determine if any slope is higher than the slope of the LOS.

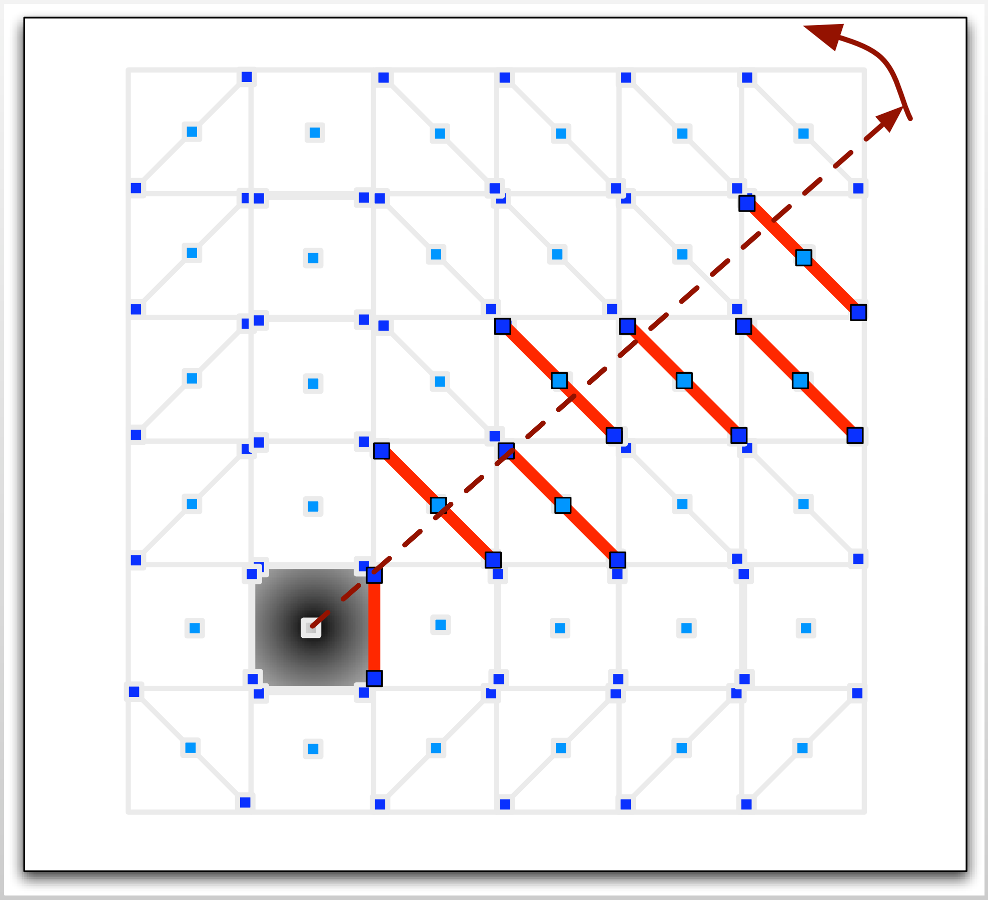
Kreveld's idea is to use a line sweep, rotating around the viewpoint, and to compute the visibility of a cell when the line passes over its center.
Each cell has 3 associated events: when it is first intersected by the line, when its center is intersected, and when it is last intersected by the sweep line.

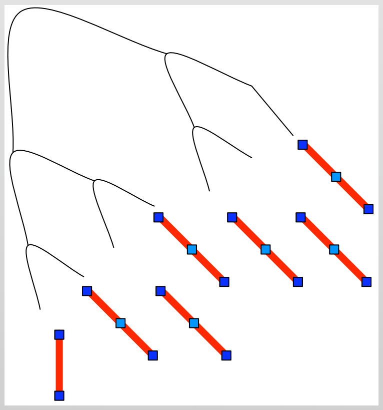
The active structure maintains the cells intersected by the sweep line at any time. When a cell is intersected for the 1st time it is inserted in the active structure, when its intersected last it is deleted, and when its centerpoint is intersected the active str is queried to determine whether the cell is visible.

For this the active structure is a binary search tree, where cells intersected by the sweep line are stored in order of the distance of their centerpoint from v. Note that when a cell is queried, all cells that are intersected by the LOS are in the active structure, to the left of the cell. If we augment the tree we can find the maximum slope to the left of a point in lg n time.

time: 3n events, lg n time each -> n lg n.

van Kreveld's algorithm





- 3n events, O(lg n) per event --> O(n lg n)   CPU time

remember that to compute if two points are visible we need to find all cells intersected by the LOS and determine if any slope is higher than the slope of the LOS.

Kreveld's idea is to use a line sweep, rotating around the viewpoint, and to compute the visibility of a cell when the line passes over its center.
Each cell has 3 associated events: when it is first intersected by the line, when its center is intersected, and when it is last intersected by the sweep line.

The active structure maintains the cells intersected by the sweep line at any time. When a cell is intersected for the 1st time it is inserted in the active structure, when its intersected last it is deleted, and when its centerpoint is intersected the active str is queried to determine whether the cell is visible.

For this the active structure is a binary search tree, where cells intersected by the sweep line are stored in order of the distance of their centerpoint from v. Note that when a cell is queried, all cells that are intersected by the LOS are in the active structure, to the left of the cell. If we augment the tree we can find the maximum slope to the left of a point in lg n time.

time: 3n events, lg n time each -> n lg n.

van Kreveld's algorithm
-in external memory-

- requires 4 structures in memory
  - input elevation grid: stored in row-column order, read in sweep order
  - output visibility grid: stored in row-column order, written in sweep order
  - event list
  - status structure

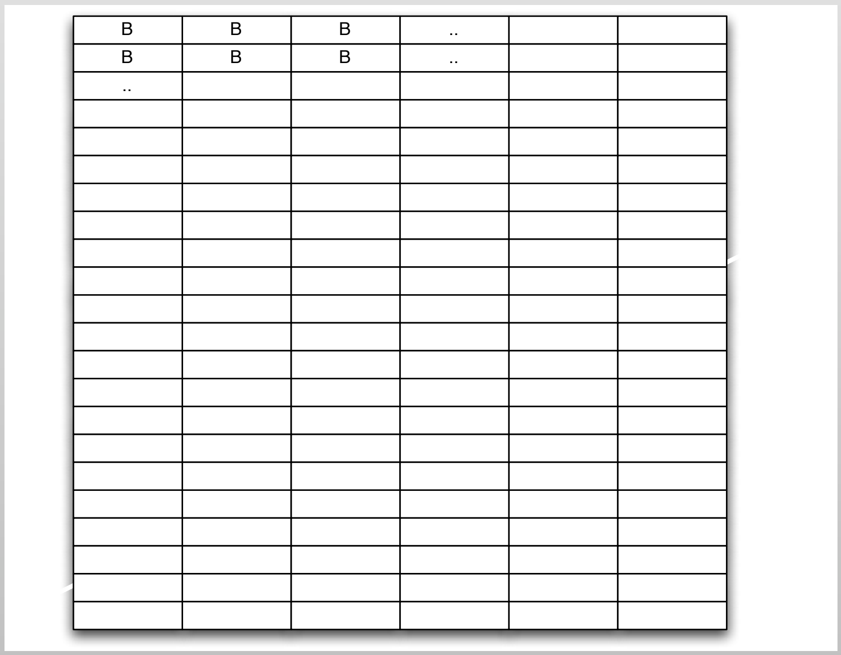
This algorithm is not efficient in external memory.

It requires 4 structures to be in memory: elevation grid, visibility grid, event list and status structure. The grids are stored in row-major order, and are accessed in rotational order. As you can see from the figure, there is some locality, but overall, it requires one I/O per element.

When n is larger than M, as we'll see from the experimental section, the algorithm starts thrashing.

van Kreveld's algorithm
-in external memory-

- requires 4 structures in memory
  - input elevation grid: stored in row-column order, read in sweep order
  - output visibility grid: stored in row-column order, written in sweep order
  - event list
  - status structure



This algorithm is not efficient in external memory.

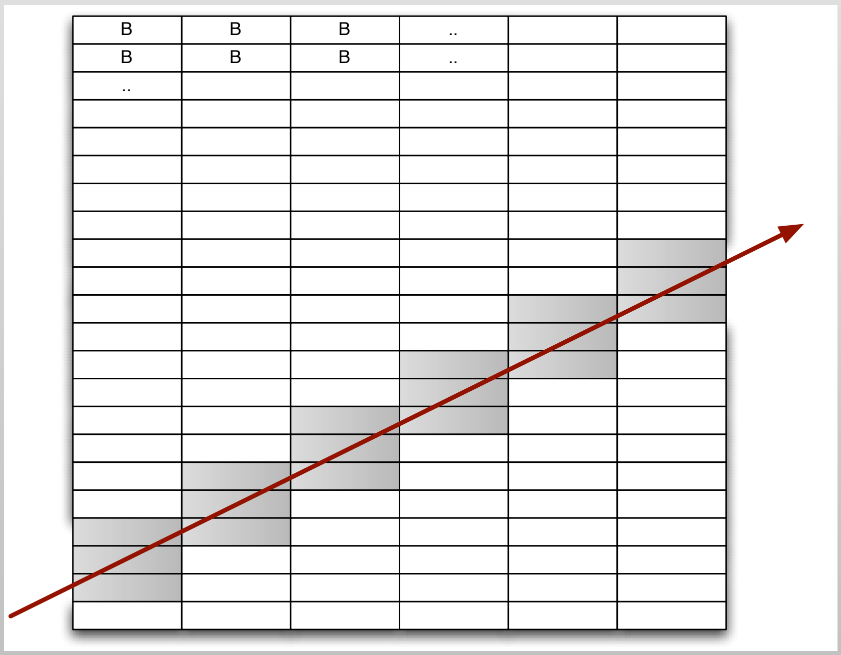
It requires 4 structures to be in memory: elevation grid, visibility grid, event list and status structure. The grids are stored in row-major order, and are accessed in rotational order. As you can see from the figure, there is some locality, but overall, it requires one I/O per element.

When n is larger than M, as we'll see from the experimental section, the algorithm starts thrashing.

van Kreveld's algorithm

-in external memory-

- requires 4 structures in memory
  - input elevation grid: stored in row-column order, read in sweep order
  - output visibility grid: stored in row-column order, written in sweep order
  - event list
  - status structure



This algorithm is not efficient in external memory.

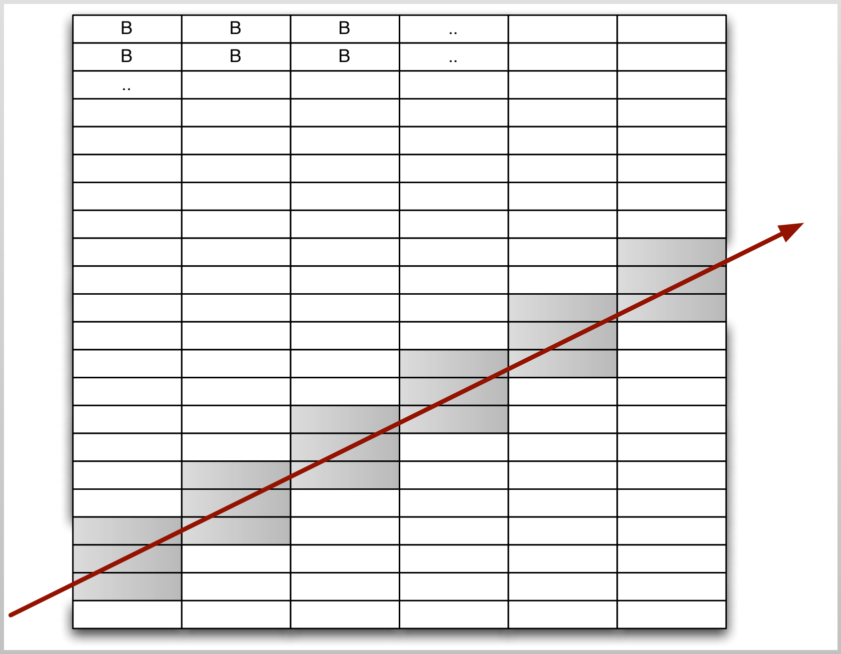
It requires 4 structures to be in memory: elevation grid, visibility grid, event list and status structure. The grids are stored in row-major order, and are accessed in rotational order. As you can see from the figure, there is some locality, but overall, it requires one I/O per element.

When n is larger than M, as we'll see from the experimental section, the algorithm starts thrashing.

van Kreveld's algorithm

-in external memory-

- requires 4 structures in memory
  - input elevation grid: stored in row-column order, read in sweep order
  - output visibility grid: stored in row-column order, written in sweep order
  - event list
  - status structure
- if $n > M$: $O(1)$ I/O per element, $O(n)$ I/Os total



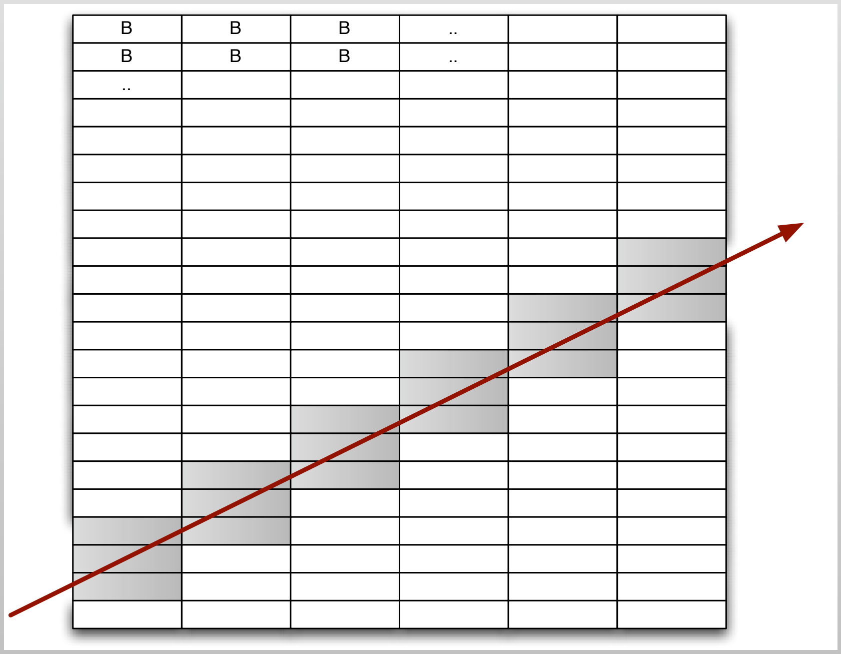
This algorithm is not efficient in external memory.

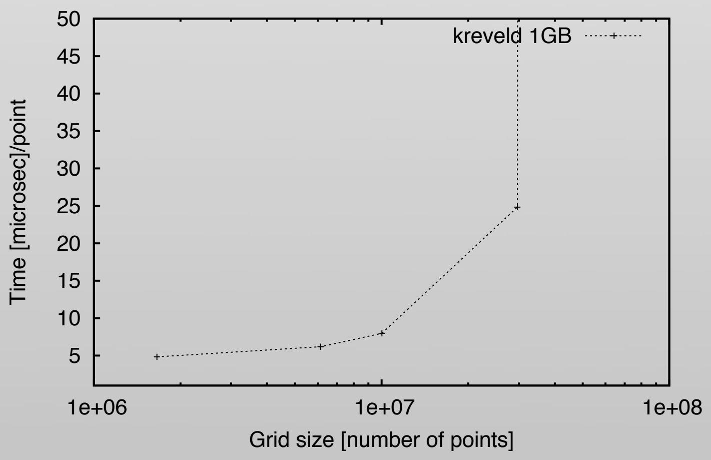
It requires 4 structures to be in memory: elevation grid, visibility grid, event list and status structure. The grids are stored in row-major order, and are accessed in rotational order. As you can see from the figure, there is some locality, but overall, it requires one I/O per element.

When n is larger than M, as we'll see from the experimental section, the algorithm starts thrashing.

van Kreveld's algorithm
-in external memory-

- requires 4 structures in memory
  - input elevation grid: stored in row-column order, read in sweep order
  - output visibility grid: stored in row-column order, written in sweep order
  - event list
  - status structure
- if n > M: O(1) I/O per element, O(n) I/Os total





This algorithm is not efficient in external memory.

It requires 4 structures to be in memory: elevation grid, visibility grid, event list and status structure. The grids are stored in row-major order, and are accessed in rotational order. As you can see from the figure, there is some locality, but overall, it requires one I/O per element.

When n is larger than M, as we'll see from the experimental section, the algorithm starts thrashing.

Our results

n = grid size   M=memory   size B=block size

- The visibility grid of an arbitrary viewpoint on a grid of size n can be computed with O(n) space and O(sort(n)) I/Os

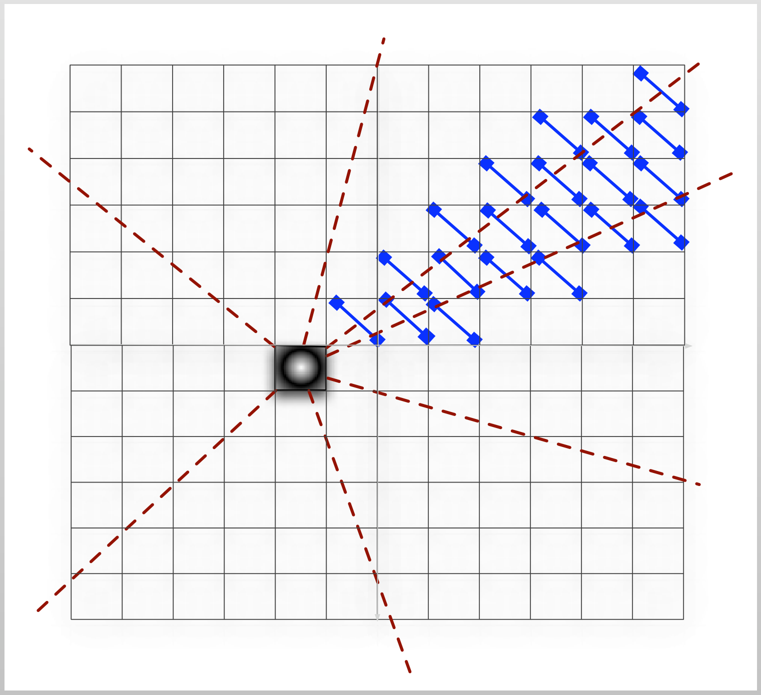
- Experimental evaluation
  - ioviewshed
  - standard algorithm (Kreveld)
  - visibility algorithm in GRASS GIS

(go very fast)

We give an algorithm to compute the visibility grid in sort(n) I/Os and an experimental evaluation to test it. We compare it with van Kreveld's algorithm and a module that computes visibility from GRASS GIS.

Computing visibility in external memory

- Distribution sweeping [GTVV FOCS93]
  - divide input in M/B sectors each containing an equal nb of points
  - solve each sector recursively
  - handle sector interactions

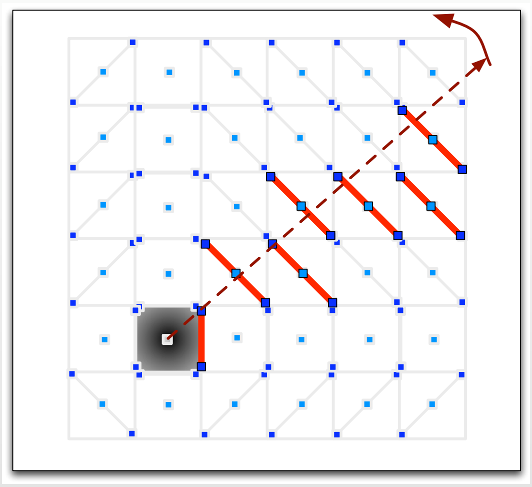


(go very fast)

The algorithm is based on distribution sweeping, which was introduced by Goodrich, Tsay, Vengroff and Vitter.

The idea is to [read slide]

The base case



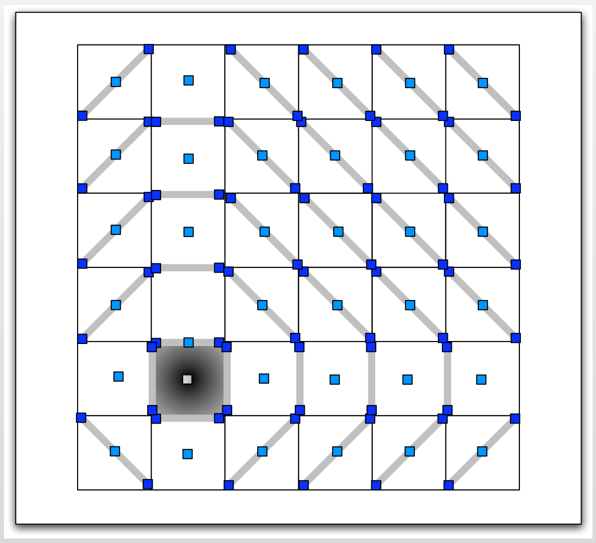
- usually, stop recursion when $n < M$
- our idea: stop when status structure fits in memory
- run modified Kreveld
  - elevation grid: encode elevation in event
  - event list: store events in a sorted stream on disk
  - visibility grid: when determining visibility of a cell, write it to a stream. Sort the stream at the end to get visibility grid
- total: O(sort(n)) I/Os

Usually, the recursion stops when the problem fits in memory.
In this case, we stop it earlier, when the status structure fits in memory.
At this point (note that the problem will not fit in memory), we run a modified version of Kreveld.

That is:
1 we get rid of elevation grid and encode elevation in events
2. store the events in a stream on disk
3. we assemble the visibility grid only at the end

for all these we use streams and i/o-efficient sorting.
Assuming the status structure fits in memory, then this uses sort(n) I/Os.

The recursion

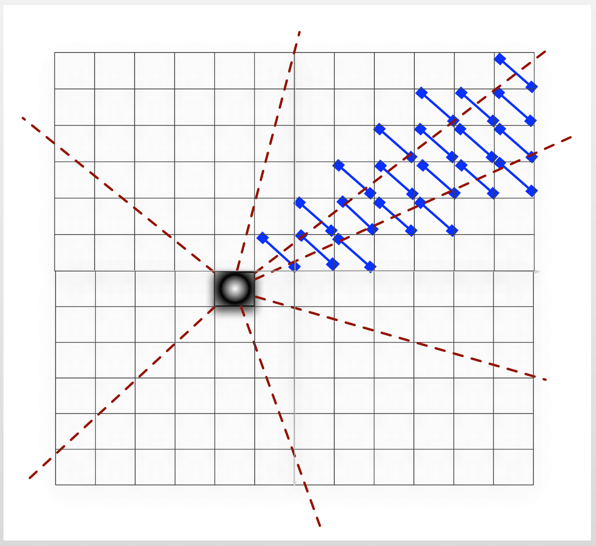


- cell <--> {start, end, query}
- 3n events

Now the recursion:

remember each cell has 3 events, in total 3n.

The recursion



- divide events into O(M/B) sectors of equal size
- $O(\log_{M/B} n)$ recursion levels

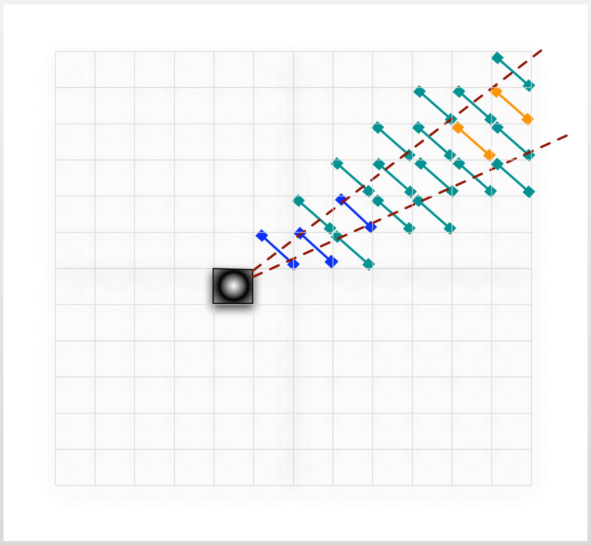
- if O(scan(n)) per recursion level
- --> overall $\mathrm{scan}(n) \cdot O(\log_{M/B} n) = O(\mathrm{sort}(n))$

we divide these events into M/B sectors of equal size. We compute visibility recursively in each sector, and this gives O () recursion levels (well, assuming we dont blow up space).

If we spend scan(n) per level, then overall we get the promised sort(n) bound.

The recursion:

Distributing events to sectors



- query points
- **narrow cells:** crossing at most one sector boundary
- **wide cells:** crossing at least two sector boundaries

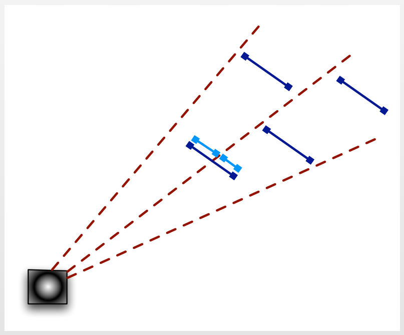
Given the list of events sorted in rotational sweep order, we can find the sector boundaries easily. Knowing the boundaries, we have to distribute the events to sectors:

- for query points, this is trivial.
- there are two types of cells: cells that fall completely within a sector or cross a single boundary. We call these narrow cells.

And cells that span completely more than one sector. we call these wide cells.

The recursion:
Distributing events to sectors

- narrow cells
    - cut and insert in both sectors



Narrow cells: if completely within a sector, insert in that scetor. Otherwise cut in two, and insert each piece in the appropriate sector.
(this keeps space linear)

The harder part is to handle wide cells -- we cannot cut a wide cell and insert it in each spanned sector because space would blow up.

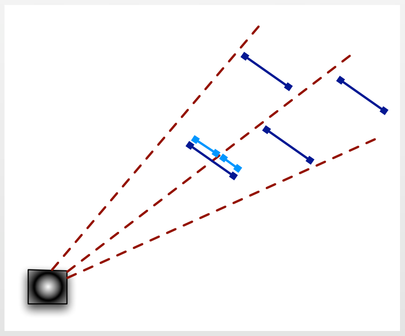
The observation here is...[read slide]

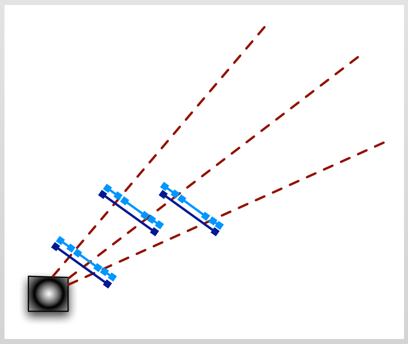
Therefore, what we do is ..... [read slide]

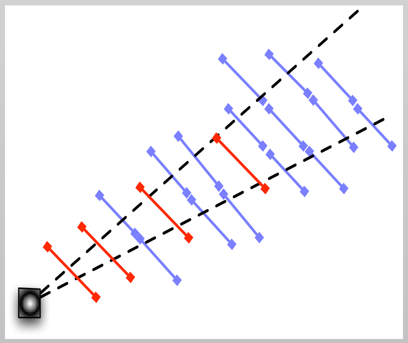
The recursion:

Distributing events to sectors

- narrow cells
    - cut and insert in both sectors
- wide cells
    - cannot insert cell in each sector spanned (space blow-up)
    - the visibility of a cell is determined by
        - the highest of all wide cells that span the sector and are closer to the viewpoint
        - all narrow cells in the sector that are closer to the viewpoint
    - for each sector, process wide cells spanning the sector interleaved with query points and narrow cells in the sector, in increasing order of their distance form viewpoint







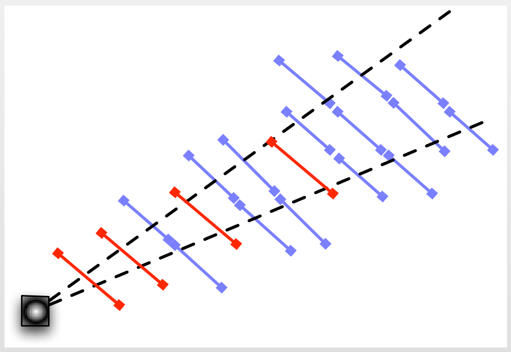
Narrow cells: if completely within a sector, insert in that scetor. Otherwise cut in two, and insert each piece in the appropriate sector.
(this keeps space linear)

The harder part is to handle wide cells -- we cannot cut a wide cell and insert it in each spanned sector because space would blow up.

The observation here is...[read slide]

Therefore, what we do is ..... [read slide]

The recursion



- input: event list in concentric order E_c and in radial order E_r
- radial sweep: scan E_r
  - find sector boundaries
  - compute a list E_r of events in each sector
- concentric sweep: scan E_c
  - for each sector
    - keep a block of events in memory
    - maintain the currently highest wide cell spanning the sector, $High_S$
  - if next event in E_c is
    - wide cell: for each sector spanned, update $High_S$ for that sector
    - narrow cell: if it is not occluded by $High_S$, insert in the buffer of sector. Otherwise skip it.
    - query point: if it is not occluded by $High_S$, insert it in the buffer of sector. Otherwise, mark it as invisible and output it.
- recurse on each sector

O(scan(n)) per recursion level -> O(sort(n)) total

This is the overall algorithm.

Experimental results

- kreveld
  - C
  - uses virtual memory system

- ioviewshed
  - C++
  - uses an I/O core derived from TPIE library

- GRASS visibility module
  - $O(n^2)$ straightforward algorithm
  - GRASS segment library for virtual memory management
  - bypass the VMS, manage data allocation and de-allocation in segments on disk
  - program will always run (no malloc() fails) but ... slow

(I expect only 2-3 minutes will be left at this point. should be enough.)

And now i'll briefly go over the experimental results.
We implemented kreveld and the ioefficient algorithm. we compare also with a module to compute visibility from the open-source GRASS GIS. the module uses the straightforward algorithm , but handles virtual memory using its own VMS library. Because o fthis it has a significant bottleneck.

Experimental results

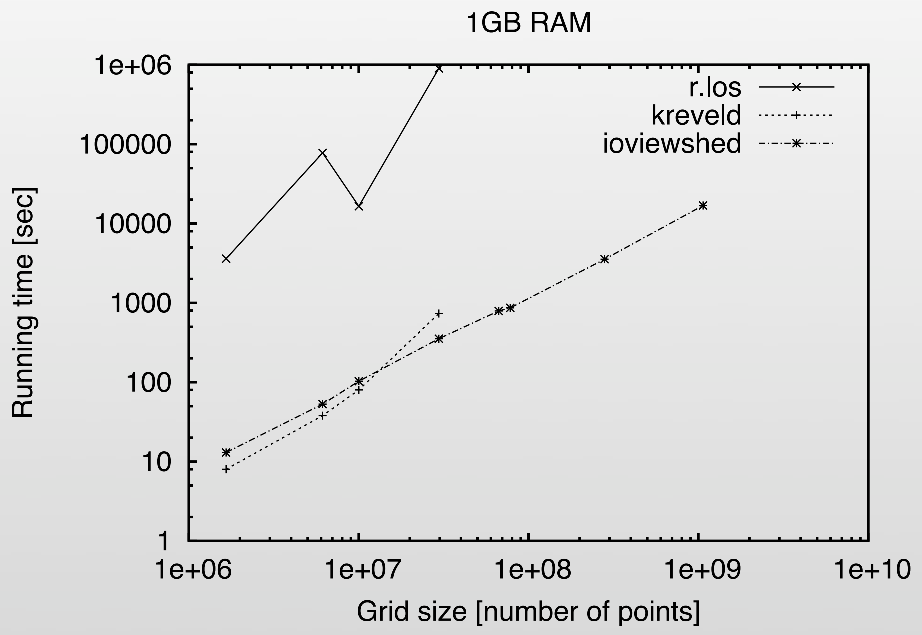
- Experimental Platform
    - Apple Power Macintosh G5
    - Dual 2.5 GHz processors
    - 512 KB L2 cache
    - 1 GB RAM

Dataset	Grid Size (million elements)	MB (Grid Only)	Valid size
Kaweah	1.6	6	56%
Puerto Rico	5.9	24	19%
Sierra Nevada	9.5	38	96%
Hawaii	28.2	112	7%
Cumberlands	67	268	27%
Lower New England	77.8	312	36%
Midwest USA	280	1100	86%
Washington	1066	4264	95%

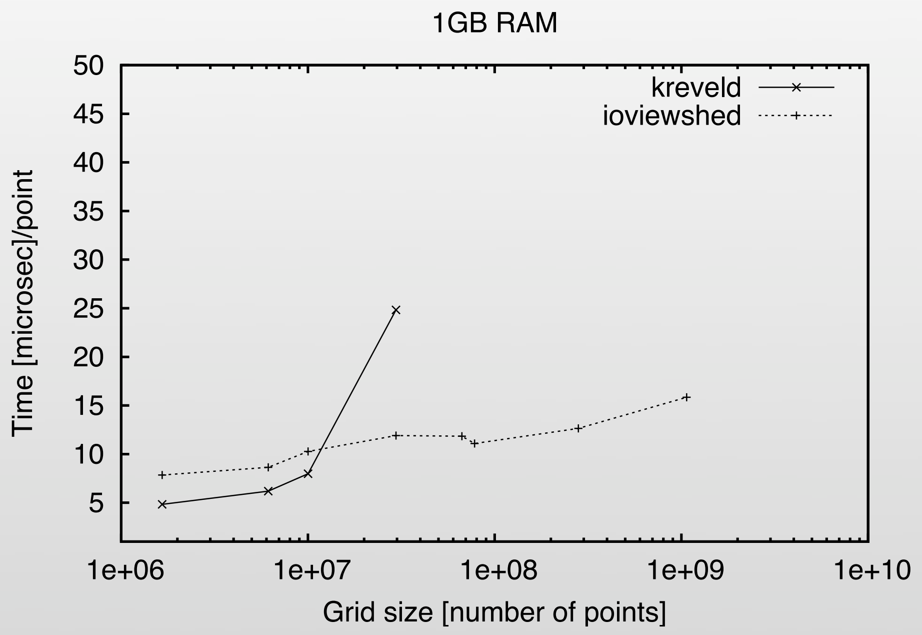
for the experiments we used G5s with 1GB RAM and test grids ranging from 1.6 M cells to 280 M ce

1GB RAM

total time (seconds)



microseconds per grid point



These are the results for 1GB RAM.

x-axis shows grid size log-scale.
Left figure shows total running time, log scale.
Right figure shows running time per grid point (on a regular scale).

we see that GRASS module is very slow, even for small sets.

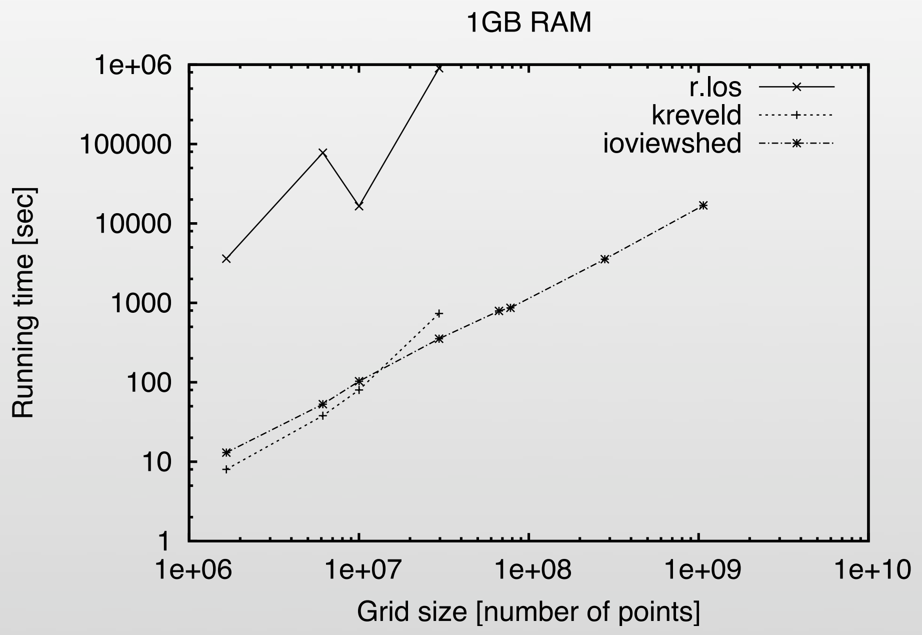
kreveld is fast if n<M , which is the case for the smallest 3 datasets. It starts thrashing onn the 4th smallest, Hawaii. its time per poitn jump to 25 microseconds, and CPU usage drops to 39%. Then on Cumberlands, kreveld cannot complete because of malloc fails.

ioviewshed on the other hand, scales nicely and finishes the 4GB dataset in 4.5 hours. In all experiments, the status structure fits in memory and it does not go into recursion.

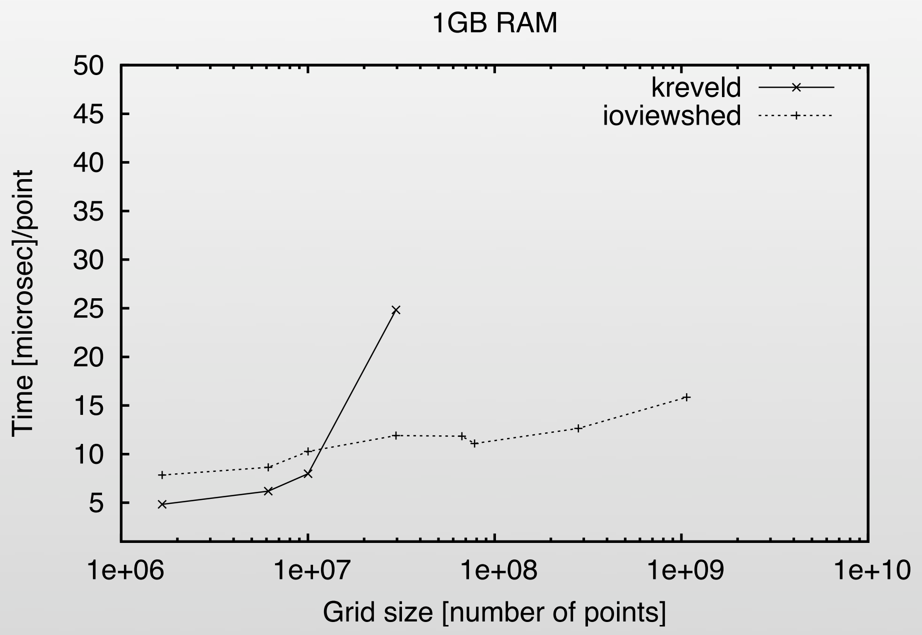
note: ioviewshed is slower than kreveld for small n. Of course, the implementation checks whether n<M and if so, it runs kreveld straihght.

1GB RAM

total time (seconds)



microseconds per grid point



- GRASS
  - program always runs (no malloc() failures) but is very slow
- kreveld
  - starts thrashing on Hawaii (39% CPU, 739 seconds)
  - malloc() fails on Cumberlands
- ioviewshed
  - finishes Washington in 4.5 hours
  - in practice status structure fits in memory, never enters recursion

These are the results for 1GB RAM.

x-axis shows grid size log-scale.
Left figure shows total running time, log scale.
Right figure shows running time per grid point (on a regular scale).

we see that GRASS module is very slow, even for small sets.

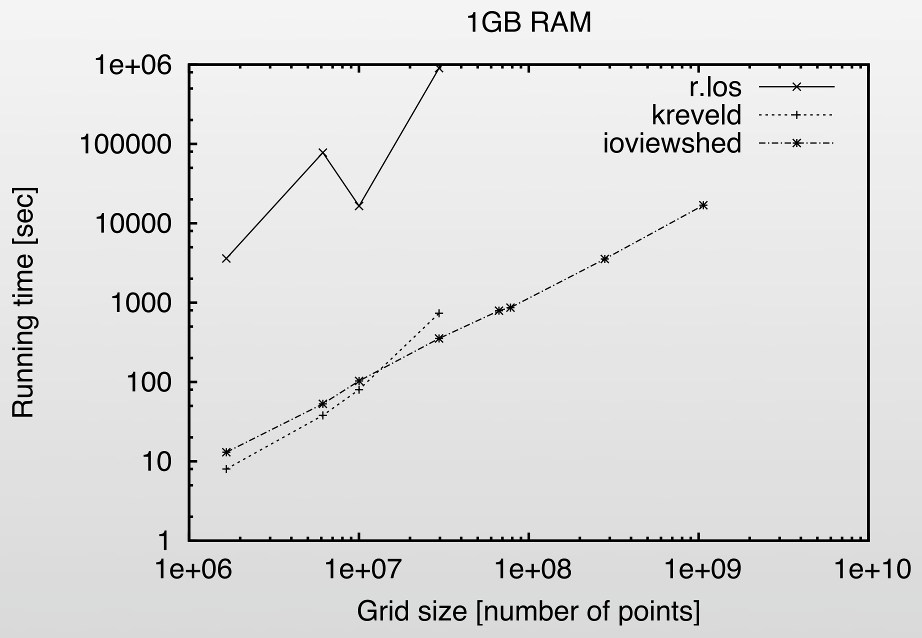
kreveld is fast if n<M , which is the case for the smallest 3 datasets. It starts thrashing onn the 4th smallest, Hawaii. its time per poitn jump to 25 microseconds, and CPU usage drops to 39%. Then on Cumberlands, kreveld cannot complete because of malloc fails.

ioviewshed on the other hand, scales nicely and finishes the 4GB dataset in 4.5 hours. In all experiments, the status structure fits in memory and it does not go into recursion.

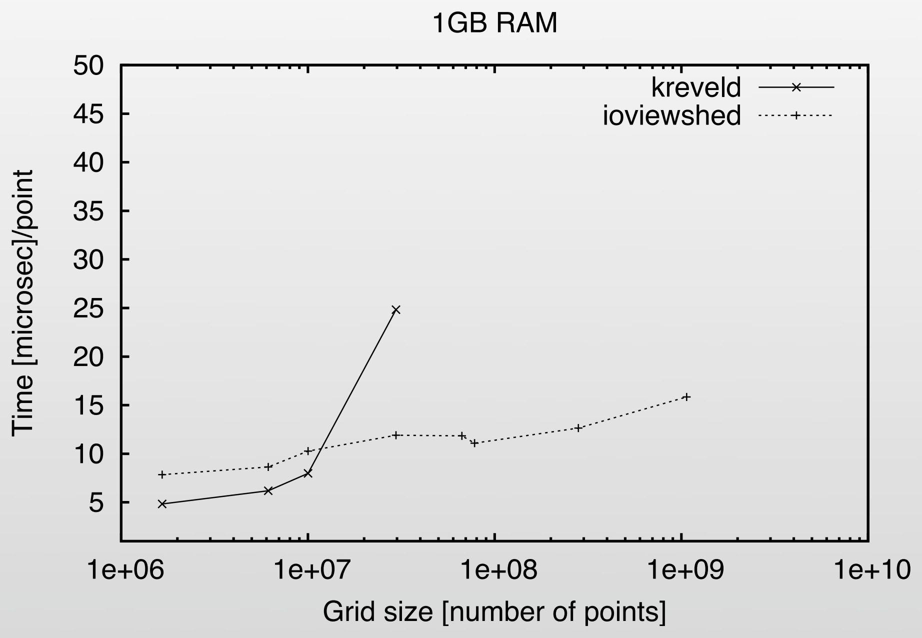
note: ioviewshed is slower than kreveld for small n. Of course, the implementation checks whether n<M and if so, it runs kreveld straihght.

1GB RAM

total time (seconds)



microseconds per grid point



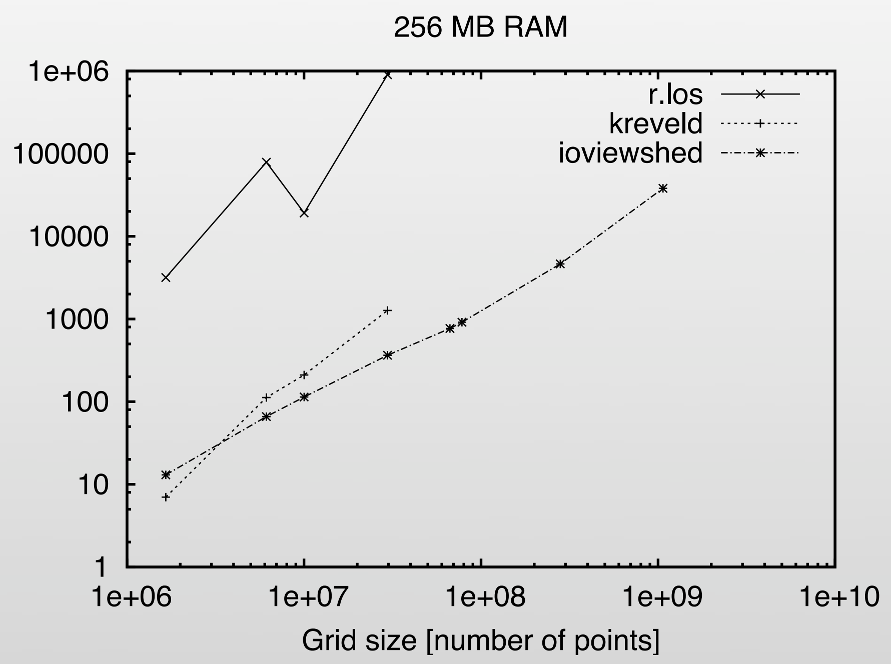
Data set	r.los	kreveld	ioviewshed
Kaweah	2 928	8 (100 %)	13 (84 %)
Puerto Rico	78 778	38 (100 %)	53 (78 %)
Sierra Nevada	16 493	80 (95 %)	102 (67 %)
Hawaii	>1 200 000	736 (39 %)	353 (63 %)
Cumberlands		malloc fails	791 (63 %)
LowerNE			865 (64 %)
Midwest USA			3 546 (64 %)
Washington			16 895 (68 %)

Table 3: Running times (seconds) and CPU-utilization (in parentheses) at 1 GB RAM.

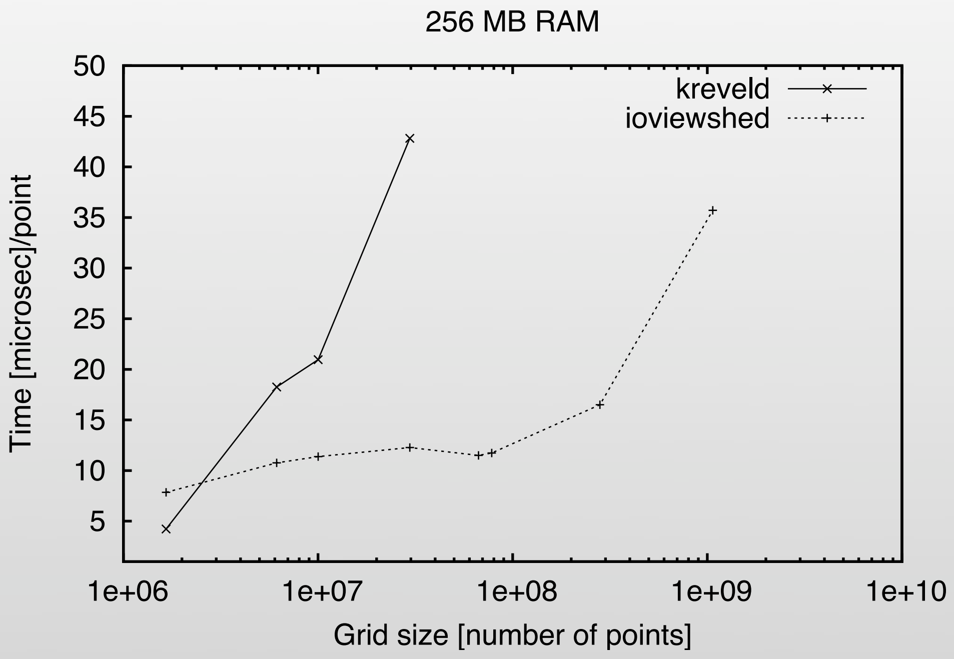
(this shows running times, skip. use it only if there are specific questions)

256MB RAM

total time (seconds)



microseconds per grid point



- kreveld starts thrashing earlier (Puerto Rico, 38% CPU)
- ioviewshed slowdown on Washington dataset
    - due 90% to sorting
    - can be improved using customized I/O sorting [TPIE, STXXL]

(not sure if there will be time for this, but i included it just in case)

We also ran experiments rebooting the machines with 256MB RAM (to simulate the effect of larger datasets).

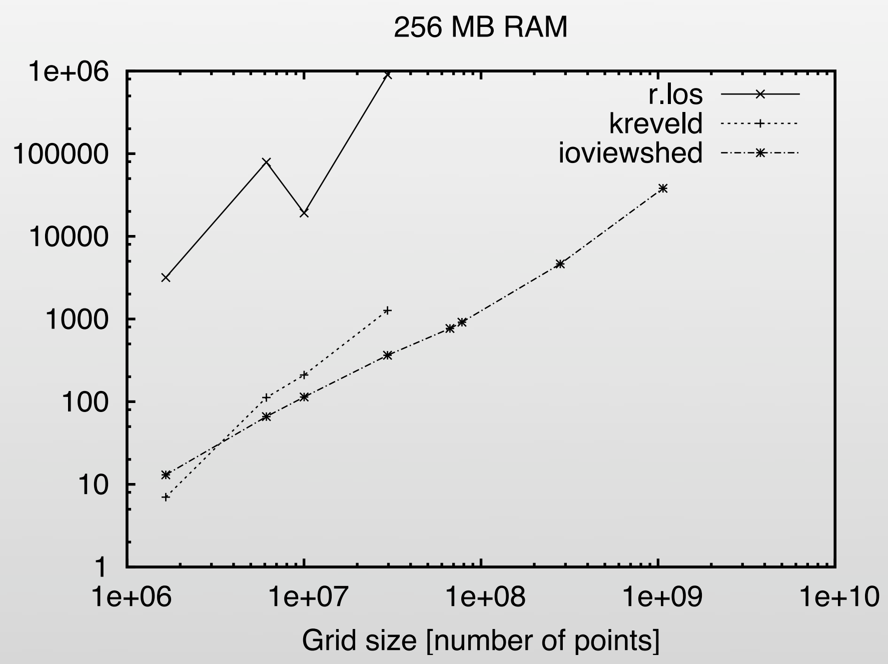
Basically things are similar, just that the crossover point between kreveld and ioviewshed moves to the left. kreveld starts thrashing earlier, on Puerto Rico dataset).

(note: the space used by kreveld for a grid of size n is 44 bytes per grid point. so 44 x 6 million points in puertorico = 256MB)

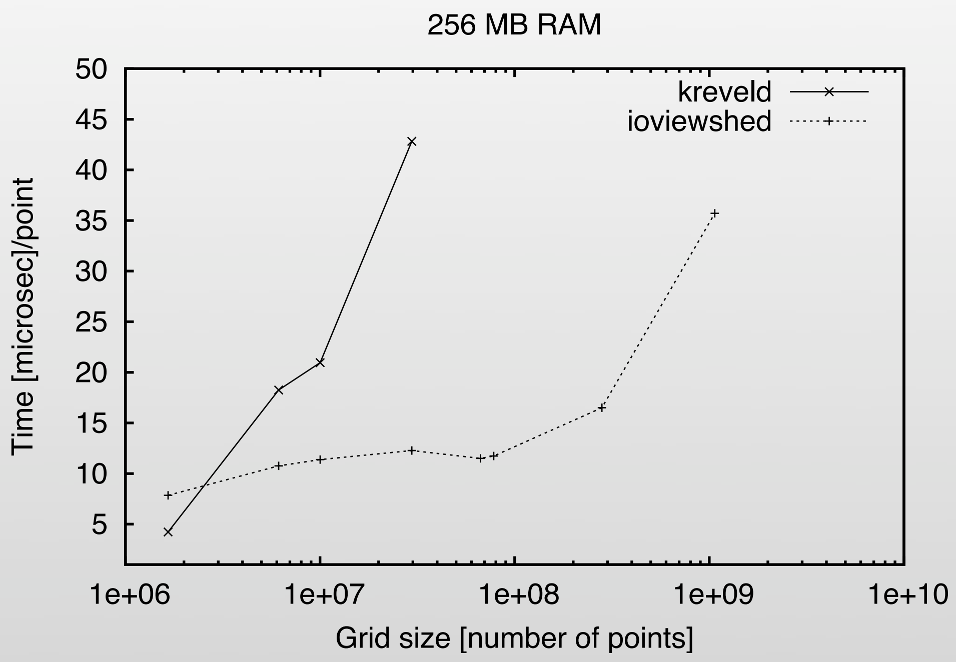
ioviewshed slows down on the largest dataset, and 90% of teh running time goes in sorting. We expect the time will be improved using a customized I/O library.

256MB RAM

total time (seconds)



microseconds per grid point



Data set	`r.los`	`kreveld`	`ioviewshed`
Kaweah	2 984	7 (100 %)	13 (77 %)
Puerto Rico	78 941	112 (38 %)	66 (60 %)
Sierra Nevada	19 140	211 (29 %)	115 (57 %)
Hawaii	>1 200 000	1270 (27 %)	364 (63 %)
Cumberlands		`malloc` fails	768 (62 %)
LowerNE			916 (62 %)
Midwest USA			4 631 (52 %)
Washington			40 734 (30 %)

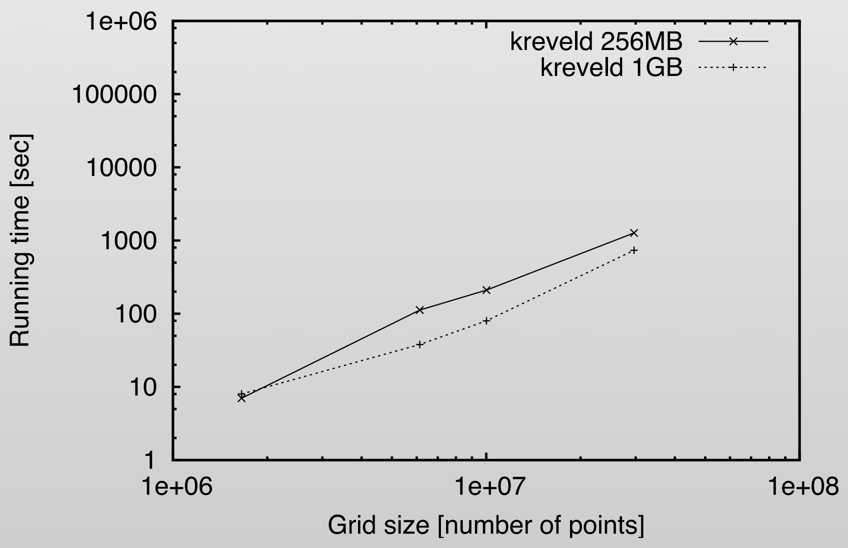
Table 4: Running times (seconds) and CPU-utilization (in parentheses) at 256 MB RAM.

(skip, use only if specific questions)

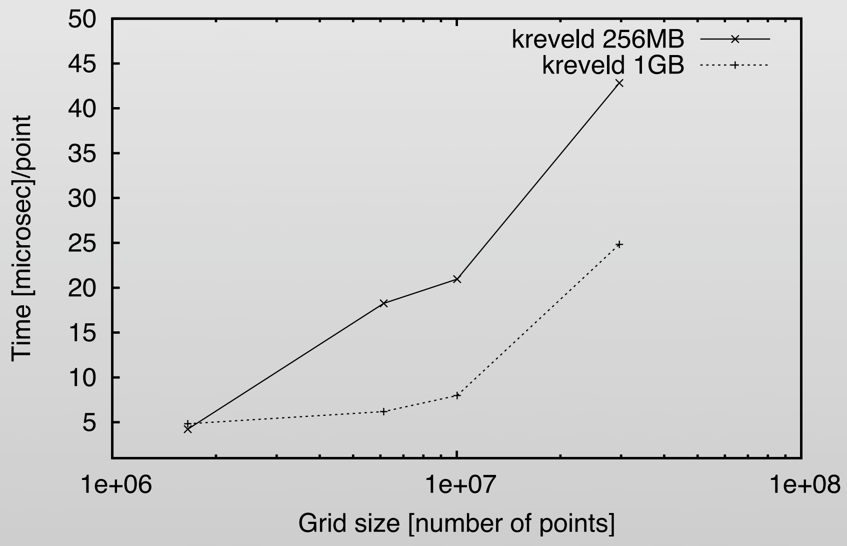
Kaweah	1.6	6	56%
Puerto Rico	5.9	24	19%
Sierra Nevada	9.5	38	96%
Hawaii	28.2	112	7%
Cumberlands	67	268	27%
Lower New England	77.8	312	36%
Midwest	280	1100	86%
Washington	1066	4264	95%

1GB vs. 256MB RAM kreveld

total time (seconds)



microseconds per grid point



- starts thrashing earlier
  - 1GB: Hawai, 39% CPU
  - 256MB: Puerto Rico 38% CPU
    -

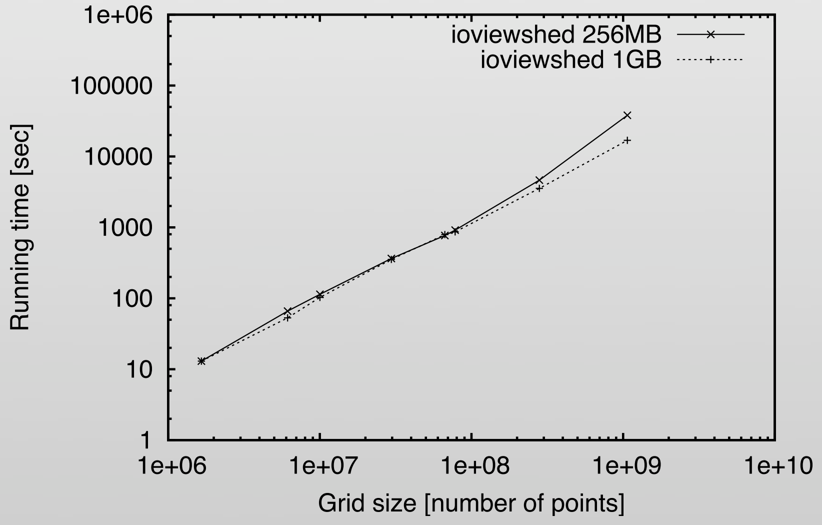
skip?
This shows the effect of RAM (1GB and 256MB RAM) for kreveld.

Notice the slowdown.

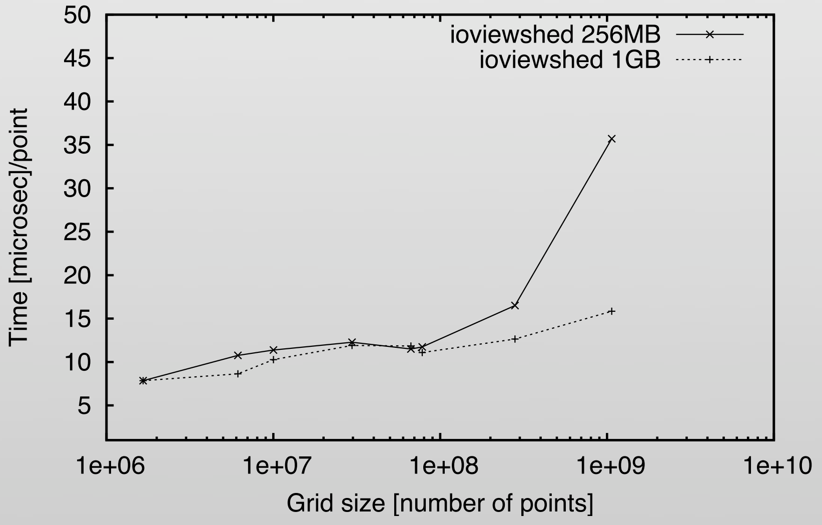
1GB vs. 256MB RAM ioviewshed

Kaweah	1.6	6	56%
Puerto Rico	5.9	24	19%
Sierra Nevada	9.5	38	96%
Hawaii	28.2	112	7%
Cumberlands	67	268	27%
Lower New England	77.8	312	36%
Midwest	280	1100	86%
Washington	1066	4264	95%

total time (seconds)



microseconds per grid point



- slowdown on Washington dataset
- due 90% to sorting
- can be improved using a customized I/O sorting [TPIE, STXXL]

skip?
Same, for ioviewshed.

Conclusion

- ioviewshed
    - theoretically worst-case optimal algorithm
    - in practice status structure fits in memory
        - with extended base case it never enters recursion
    - scalable
        - can process grids that are out of scope with traditional algorithm

Thank you.