Computing Visibility on Terrains in External Memory

Herman Haverkort
TU. Eindhoven
Netherlands

Laura Toma
Bowdoin College
USA

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

\[
\begin{align*}
n = \text{input size} & \quad M = \text{memory size} & \quad B = \text{block size}
\end{align*}
\]

- **I/O-complexity: nb. I/O-operations**

- **Basic I/O bounds**

\[
\begin{align*}
\text{scan}(n) &= \Theta \left( \frac{n}{B} \right) & < & \quad \text{sort}(n) &= \Theta \left( \frac{n}{B} \log_{M/B} \frac{n}{M} \right) & \ll & \quad n
\end{align*}
\]
Terrain data

- Most often: grid terrain
- TIN (triangulated polyhedral terrain)
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.
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]
van Kreveld’s algorithm
van Kreveld’s algorithm
van Kreveld's algorithm
van Kreveld’s algorithm
van Kreveld’s algorithm
van Kreveld’s algorithm
van Kreveld’s algorithm
van Kreveld's algorithm
van Kreveld’s algorithm

- 3n events, $O(\lg n)$ per event $\rightarrow O(n \lg n)$ CPU time
van Kreveld’s algorithm
—in external memory—

van Kreveld’s algorithm
-in external memory-

- Requires 4 structures in memory
  - input elevation grid, output visibility grid
    - stored in row-major order, accessed in rotational order
  - event list
  - active structure
van Kreveld’s algorithm
-in external memory-

- Requires 4 structures in memory
  - input elevation grid, output visibility grid
    - stored in row-major order, accessed in rotational order
  - event list
  - active structure
van Kreveld’s algorithm
-in external memory-

- Requires 4 structures in memory
  - input elevation grid, output visibility grid
    - stored in row-major order, accessed in rotational order
  - event list
  - active structure

- $\Omega(1)$ I/O per cell, $\Omega(n)$ I/Os total

\begin{verbatim}
B B B
B B B
.....
B B B
\end{verbatim}
van Kreveld’s algorithm
-in external memory-

- Requires 4 structures in memory
  - input elevation grid, output visibility grid
    - stored in row-major order, accessed in rotational order
  - event list
  - active structure
- $\Omega(1)$ I/O per cell, $\Omega(n)$ I/Os total

![Graph showing time vs grid size for van Kreveld's algorithm](image)

<table>
<thead>
<tr>
<th>Grid size [number of points]</th>
<th>Time [microsec]/point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e+06</td>
<td>5</td>
</tr>
<tr>
<td>1e+07</td>
<td>10</td>
</tr>
<tr>
<td>1e+08</td>
<td>20</td>
</tr>
</tbody>
</table>

![Graph showing I/O per cell vs grid size for van Kreveld's algorithm](image)
van Kreveld’s algorithm
—in external memory—

- Requires 4 structures in memory
  - input elevation grid, output visibility grid
    - stored in row-major order, accessed in rotational order
  - event list
  - active structure
- \( \Omega(1) \) I/O per cell, \( \Omega(n) \) I/Os total
Our results

\[ n = \text{grid size} \quad M = \text{memory size} \quad \text{size } B = \text{block size} \]

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

- **Experimental evaluation**
  - ioviewshed
  - standard algorithm (Kreveld)
  - visibility algorithm in 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
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
The recursion

- cell $\leftrightarrow$ \{start, end, query\}
- $3n$ events
The recursion

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

If $O(\text{scan}(n))$ per recursion level

$\Rightarrow$ overall $\text{scan}(n) \cdot O(\log_{M/B} n) = O(\text{sort}(n))$
The recursion:
Distributing events to sectors

- query points
- narrow cells: crossing at most one sector boundary
- wide cells: crossing at least two sector boundaries
The recursion:
Distributing events to sectors

- narrow cells
  - cut and insert in both sectors
The recursion:
Distributing events to sectors

- **narrow cells**
  - cut and insert in both sectors

- **wide cells**
  - the visibility of a cell is determined by
    - all narrow cells in its sector that are closer to the viewpoint
    - the highest of all wide cells that span the sector and are closer to the viewpoint
  - **concentric sweep**
    - process wide cells spanning the sector interleaved with query points and narrow cells in the sector
The recursion

- **Input:** event list in concentric order $E_c$ and in radial order $E_r$

- **Radial sweep:** scan $E_r$
  - find sector boundaries

- **Concentric sweep:** scan $E_c$
  - for each sector
    - keep a block of events in memory
    - maintain the current highest wide cell spanning the sector, $H_{s}$
  - if next event in $E_c$ is
    - narrow cell: if it is not occluded by $H_{s}$, insert in the buffer of sector. Otherwise skip it.
    - wide cell: for each sector spanned, update $H_{s}$
    - query point: if it is not occluded by $H_{s}$, insert it in the buffer of sector. Otherwise, mark it as invisible and output it.
  - Recurse on each sector

$O(\text{scan}(n))$ per recursion level $\rightarrow O(\text{sort}(n))$ total
Experimental results
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
  - uses GRASS library for virtual memory management
  - program will always run (no malloc() fails) but ... slow
Experimental results

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Grid Size (million elements)</th>
<th>MB (Grid Only)</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaweah</td>
<td>1.6</td>
<td>7</td>
<td>56%</td>
</tr>
<tr>
<td>Sierra Nevada</td>
<td>9.5</td>
<td>40</td>
<td>96%</td>
</tr>
<tr>
<td>Cumberlands</td>
<td>67</td>
<td>267</td>
<td>27%</td>
</tr>
<tr>
<td>Lower New England</td>
<td>77.8</td>
<td>311</td>
<td>36%</td>
</tr>
<tr>
<td>East Coast USA</td>
<td>246</td>
<td>983</td>
<td>36%</td>
</tr>
<tr>
<td>Midwest USA</td>
<td>280</td>
<td>1122</td>
<td>86%</td>
</tr>
<tr>
<td>Washington</td>
<td>1066</td>
<td>4264</td>
<td>95%</td>
</tr>
</tbody>
</table>
Sierra Nevada, 30m resolution, 40MB
East-Coast USA, 30m resolution, 983 MB
1GB RAM

total time (seconds)

1GB RAM

microseconds per grid point

1GB RAM

Running time [sec]
Valid grid size [number of points]

1GB RAM

Time [microsec]/point
Valid grid size [number of points]
1GB RAM

- **GRASS**
  - program always runs (no malloc() failures) but is very slow
- **kreveld**
  - starts thrashing on Cumberlands (75% CPU)
  - malloc() fails on East-Coast USA
- **ioviewshed**
  - finishes Washington in 3.3 hours
1GB RAM

**total time (seconds)**

**microseconds per grid point**

---

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

<table>
<thead>
<tr>
<th>Data set</th>
<th>r.los</th>
<th>kreveld</th>
<th>ioviewshed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaweah</td>
<td>2928</td>
<td>4 (100%)</td>
<td>6 (88%)</td>
</tr>
<tr>
<td>Sierra Nevada</td>
<td>16493</td>
<td>56 (100%)</td>
<td>68 (72%)</td>
</tr>
<tr>
<td>Cumberlands</td>
<td>&gt;1200000</td>
<td>538 (78%)</td>
<td>196 (70%)</td>
</tr>
<tr>
<td>LowerNE</td>
<td>1226</td>
<td>312 (62%)</td>
<td></td>
</tr>
<tr>
<td>East-Coast USA</td>
<td></td>
<td>894 (65%)</td>
<td></td>
</tr>
<tr>
<td>Horn of Africa</td>
<td></td>
<td>1969 (61%)</td>
<td></td>
</tr>
<tr>
<td>Midwest USA</td>
<td></td>
<td>2319 (63%)</td>
<td></td>
</tr>
<tr>
<td>Washington</td>
<td></td>
<td>12148 (65%)</td>
<td></td>
</tr>
</tbody>
</table>
kreveld starts thrashing earlier (Sierra, 33% CPU)
ioviewshed scales up
256MB RAM

**total time (seconds)**

Running time [sec]

Valid grid size [number of points]

**microseconds per grid point**

Time [microsec]/point

Valid grid size [number of points]

<table>
<thead>
<tr>
<th>Data set</th>
<th>r.los</th>
<th>kreveld</th>
<th>ioviewshed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaweah</td>
<td>3177</td>
<td>6 (94%)</td>
<td>6 (86%)</td>
</tr>
<tr>
<td>Sierra Nevada</td>
<td>19140</td>
<td>288 (33%)</td>
<td>97 (53%)</td>
</tr>
<tr>
<td>Cumberlands</td>
<td>&gt;1200000</td>
<td>932 (52%)</td>
<td>224 (63%)</td>
</tr>
<tr>
<td>LowerNE</td>
<td>1776</td>
<td>370 (54%)</td>
<td>370 (54%)</td>
</tr>
<tr>
<td>East-Coast USA</td>
<td>malloc fails</td>
<td>1254 (49%)</td>
<td>1254 (49%)</td>
</tr>
<tr>
<td>Horn of Africa</td>
<td></td>
<td>2601 (52%)</td>
<td>2601 (52%)</td>
</tr>
<tr>
<td>Midwest USA</td>
<td></td>
<td>3290 (52%)</td>
<td>3290 (52%)</td>
</tr>
<tr>
<td>Washington</td>
<td></td>
<td>18717 (50%)</td>
<td>18717 (50%)</td>
</tr>
</tbody>
</table>

Table IV. Running times (seconds) and CPU-utilization (in parentheses) at 256 MB RAM.
### 1GB vs. 256MB RAM

**kreveld**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Grid Size (million elements)</th>
<th>MB (Grid Only)</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaweah</td>
<td>1.6</td>
<td>7</td>
<td>56%</td>
</tr>
<tr>
<td>Sierra Nevada</td>
<td>9.5</td>
<td>40</td>
<td>96%</td>
</tr>
<tr>
<td>Cumberlands</td>
<td>67</td>
<td>267</td>
<td>27%</td>
</tr>
<tr>
<td>Lower New England</td>
<td>77.8</td>
<td>311</td>
<td>36%</td>
</tr>
<tr>
<td>East Coast USA</td>
<td>246</td>
<td>983</td>
<td>36%</td>
</tr>
<tr>
<td>Midwest USA</td>
<td>280</td>
<td>1122</td>
<td>86%</td>
</tr>
<tr>
<td>Washington</td>
<td>1066</td>
<td>4264</td>
<td>95%</td>
</tr>
</tbody>
</table>

#### Total Time (seconds)

- **kreveld 256MB**
- **kreveld 1GB**

#### Microseconds per Grid Point

- **kreveld 256MB**
- **kreveld 1GB**

---

**Graphs**

- **Total Time (seconds)**
- **Microseconds per Grid Point**
1GB vs. 256MB RAM

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Grid Size (million elements)</th>
<th>MB (Grid Only)</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaweah</td>
<td>1.6</td>
<td>7</td>
<td>56%</td>
</tr>
<tr>
<td>Sierra Nevada</td>
<td>9.5</td>
<td>40</td>
<td>96%</td>
</tr>
<tr>
<td>Cumberlands</td>
<td>67</td>
<td>267</td>
<td>27%</td>
</tr>
<tr>
<td>Lower New England</td>
<td>77.8</td>
<td>311</td>
<td>36%</td>
</tr>
<tr>
<td>Sierra Nevada</td>
<td>246</td>
<td>983</td>
<td>36%</td>
</tr>
<tr>
<td>Cumberlands</td>
<td>280</td>
<td>1122</td>
<td>86%</td>
</tr>
<tr>
<td>Lower New England</td>
<td>1066</td>
<td>4264</td>
<td>95%</td>
</tr>
</tbody>
</table>

- **total time (seconds)**
- **microseconds per grid point**

- Increase due to sorting
  - May be optimized using customized sorting (STXXL)
  - In practice status structure fits in memory, never enters recursion
Conclusion

- I/O-efficient visibility computation
  - Theoretically worst-case optimal algorithm

- Scalable
  - Can process grids that are out of scope with traditional algorithm

- Empirical finding:
  - diagonal of dataset fits in memory
  - extended base case, no recursion necessary

- $O(sort(n))$ I/Os in cache-oblivious model
Thank you.