Computing Visibility on Terrains in External Memory

Herman Haverkort       Laura Toma       Yi Zhuang

TU. Eindhoven  
Netherlands       Bowdoin College  
USA

ALENEX 2007
New Orleans, USA
Problem: visibility map (viewshed) of v
- terrain T
- arbitrary viewpoint v
- the set of points in T visible from v
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)
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

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

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

- **Basic I/O bounds**
  \[
  \text{scan}(n) = \Theta \left( \frac{n}{B} \right) \quad < \quad \text{sort}(n) = \Theta \left( \frac{n}{B} \log_{M/B} \frac{n}{M} \right) \quad \ll \quad n
  \]
**Terrain data**

- **Most often:** grid terrain
- **TIN (triangulated polyhedral terrain)**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>23</td>
<td>25</td>
<td>26</td>
<td>32</td>
<td>46</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>41</td>
<td>46</td>
</tr>
<tr>
<td>24</td>
<td>21</td>
<td>23</td>
<td>31</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>24</td>
<td>27</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>32</td>
<td>22</td>
<td>29</td>
<td>30</td>
<td>35</td>
<td>34</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>33</td>
<td>34</td>
<td>36</td>
<td>37</td>
</tr>
</tbody>
</table>
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-

Requires 4 structures in memory

- input elevation grid, output visibility grid
  - stored in row-major order, read in sweep order
- event list
- status structure
van Kreveld’s algorithm
—in external memory—

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

- Requires 4 structures in memory
  - input elevation grid, output visibility grid
    - stored in row-major order, read in sweep order
  - event list
  - status structure
van Kreveld's algorithm
-in external memory-

- Requires 4 structures in memory
  - input elevation grid, output visibility grid
    - stored in row-major order, read in sweep order
  - event list
  - status structure
- If $n > M$: $O(1)$ I/O per element, $O(n)$ I/Os total
van Kreveld’s algorithm
-in external memory-

- Requires 4 structures in memory
  - input elevation grid, output visibility grid
    - stored in row-major order, read in sweep order
  - event list
  - status structure

- If $n > M$: $O(1)$ I/O per element, $O(n)$ I/Os total
Our results

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

- 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: \( O(\text{sort}(n)) \) I/Os
The recursion cell $\leftrightarrow \{\text{start, end, query}\}$

- 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**
  - 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 from viewpoint
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, \( \text{High}_S \)
  - if next event in \( E_c \) is
    - wide cell: for each sector spanned, update \( \text{High}_S \) for that sector.
    - narrow cell: if it is not occluded by \( \text{High}_S \), insert in the buffer of sector. Otherwise skip it.
    - query point: if it is not occluded by \( \text{High}_S \), 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

- **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
Experimental results

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

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

**total time (seconds)**

![Graph showing total time (seconds) vs. grid size for different algorithms.]

**microseconds per grid point**

![Graph showing microseconds per grid point vs. grid size for different algorithms.]

<table>
<thead>
<tr>
<th>Grid size [number of points]</th>
<th>Running time [sec]</th>
<th>Time [microsec]/point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1GB RAM</td>
<td></td>
</tr>
<tr>
<td>r.los</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kreveld</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioviewshed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Table 3: Running times (seconds) and CPU-utilization (in parentheses) at 1 GB RAM.
• 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]
256MB 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>2984</td>
<td>7 (100%)</td>
<td>13 (77%)</td>
</tr>
<tr>
<td>Puerto Rico</td>
<td>78941</td>
<td>112 (38%)</td>
<td>66 (60%)</td>
</tr>
<tr>
<td>Sierra Nevada</td>
<td>19140</td>
<td>211 (29%)</td>
<td>115 (57%)</td>
</tr>
<tr>
<td>Hawaii</td>
<td>&gt;1200000</td>
<td>1270 (27%)</td>
<td>364 (63%)</td>
</tr>
<tr>
<td>Cumberlands</td>
<td></td>
<td>malloc fails</td>
<td>768 (62%)</td>
</tr>
<tr>
<td>LowerNE</td>
<td></td>
<td></td>
<td>916 (62%)</td>
</tr>
<tr>
<td>Midwest USA</td>
<td></td>
<td></td>
<td>4631 (52%)</td>
</tr>
<tr>
<td>Washington</td>
<td></td>
<td></td>
<td>40734 (30%)</td>
</tr>
</tbody>
</table>

Table 4: Running times (seconds) and CPU-utilization (in parentheses) at 256 MB RAM.
1GB vs. 256MB RAM

Kreveld

<table>
<thead>
<tr>
<th>Location</th>
<th>Total Time (s)</th>
<th>Microseconds/Point</th>
<th>CPU Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaweah</td>
<td>1.6</td>
<td>5</td>
<td>56%</td>
</tr>
<tr>
<td>Puerto Rico</td>
<td>5.9</td>
<td>24</td>
<td>19%</td>
</tr>
<tr>
<td>Sierra Nevada</td>
<td>9.5</td>
<td>38</td>
<td>96%</td>
</tr>
<tr>
<td>Hawaii</td>
<td>28.2</td>
<td>112</td>
<td>7%</td>
</tr>
<tr>
<td>Cumberlands</td>
<td>67</td>
<td>268</td>
<td>27%</td>
</tr>
<tr>
<td>Lower New England</td>
<td>77.8</td>
<td>312</td>
<td>36%</td>
</tr>
<tr>
<td>Midwest</td>
<td>280</td>
<td>1100</td>
<td>86%</td>
</tr>
<tr>
<td>Washington</td>
<td>1066</td>
<td>4264</td>
<td>95%</td>
</tr>
</tbody>
</table>

- Starts thrashing earlier
  - 1GB: Hawai, 39% CPU
  - 256MB: Puerto Rico 38% CPU
1GB vs. 256MB RAM

ioviewshed

total time (seconds)

<table>
<thead>
<tr>
<th>Region</th>
<th>Running time (seconds)</th>
<th>Total time (microseconds)</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaweah</td>
<td>1.6</td>
<td>6</td>
<td>56%</td>
</tr>
<tr>
<td>Puerto Rico</td>
<td>5.9</td>
<td>24</td>
<td>19%</td>
</tr>
<tr>
<td>Sierra Nevada</td>
<td>9.5</td>
<td>38</td>
<td>96%</td>
</tr>
<tr>
<td>Hawaii</td>
<td>28.2</td>
<td>112</td>
<td>7%</td>
</tr>
<tr>
<td>Cumberlands</td>
<td>67</td>
<td>268</td>
<td>27%</td>
</tr>
<tr>
<td>Lower New England</td>
<td>77.8</td>
<td>312</td>
<td>36%</td>
</tr>
<tr>
<td>Midwest</td>
<td>280</td>
<td>1100</td>
<td>86%</td>
</tr>
<tr>
<td>Washington</td>
<td>1066</td>
<td>4264</td>
<td>95%</td>
</tr>
</tbody>
</table>

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

- I/O-efficient visibility computation
  - 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.