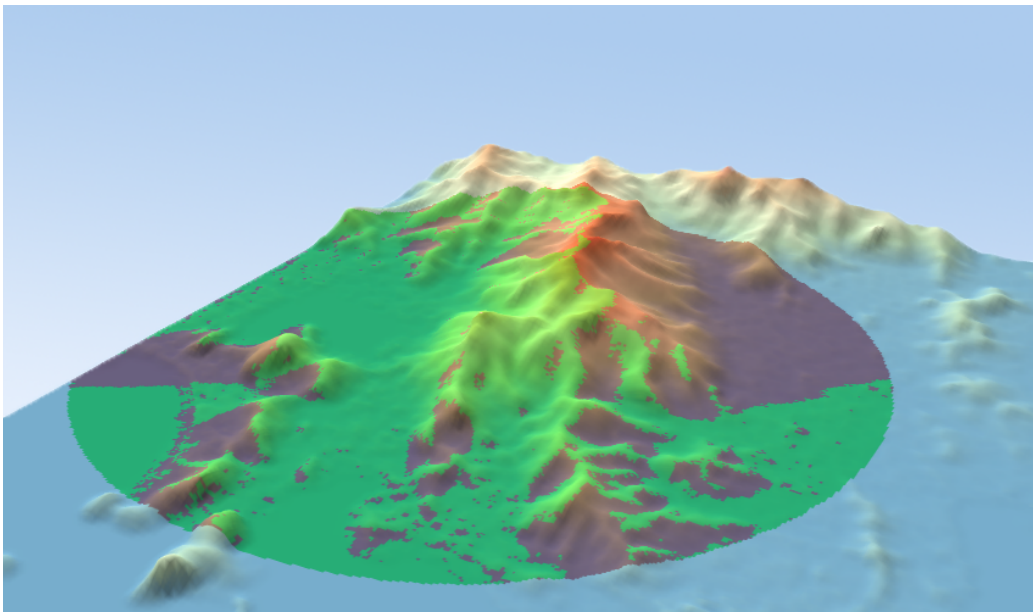
Delft University of Technology

Digital Terrain Modelling
GEO1015

Assignment 2: Viewshed Computation



Michiel de Jong (4376978)

December 17, 2020

TU Delft Delft University of Technology

General Strategy for Viewshed Computation

This section will describe step by step the what was done to compute the viewshed analysis. After each step, the relevant parts of the step are illustrated through either pseudocode or a diagram.

1. **Initialize output array and multiple viewpoint handling.**
   To start everything off, we need to initialize the output array that will be written to file. To make everything easier and faster, I have chosen take an game/explore map approach. This entails that from the outset, everything will be considered invisible, except the viewpoint, until you have seen it.

   To enable the code to handle multiple viewpoints, only the initialisation of the height (input) grid, the visibility (output) grid, and the writing of the file are done outside a loop that iterates over the viewpoints supplied by the .json file.

Code block 1 Viewpoint looping

```
input = DEM
output = raster with shape(input) and values = 3        ▷ 3 value for outside horizon
parameters = .json file
for viewpoint = 1, 2, ... do
    Compute viewshed analysis around viewpoint
end for
Write to file using supplied code
```

2. **Construct circle based on max viewing distance**
   To shorten computation times, I have chosen to initiate a circle with the maximum viewing distance as its radius and the viewpoint as its center, and use the cells describing the circumference of that circle as endpoints in Line-of-Sight queries. This is faster, because I only have to consider pixels inside the circles, because all other points are already set to the invisibility value of 3.

   The radius is extrapolated through the index of the raster, thus choosing the pixel where the max distance ends, and then using the max distance in "index space" for the construction of the circle. Because the circle is a type of geometry that is not supported by GeoJSON, I had to mathematically construct the circle with the following algorithm.

Code block 2 Circle construction

```
r = index[viewpoint_x + maxdistance] − index[viewpoint]
center = index[viewpoint]
for 0 < Θ ≤ 360 do                                      ▷ To cover all cells between VP and edge
    x_index[pointoncircle] = r · sin Θ + x_index[viewpoint]
    y_index[pointoncircle] = r · cos Θ + y_index[viewpoint]
    Add tuple(x, y) to list of cells that describe circle circumference.
end for
Remove duplicate tuples from list                       ▷ Because of increment value
```

3. **Rasterize line with rasterio**
   The following step is to construct the lines on which the LoS queries will be performed. This is done using a Rasterio module that burns shapes into a raster. The lines are instantiated for each cell in the list of cells that describe the circumference of the max viewing distance circle. Most of this code was already provided, but I will describe it shortly: a GeoJSON LineString is created between the viewpoint(xy) and circle cell (xy). This line is input into the rasterize function, which returns a Boolean array of $shape = input$ with True for the cells which represent the line. This array needs to be converted into a list of cells that will be used to compute the LoS. This is done by determining the lines' position in which quadrant of the circle, and thus sorting it either according to the lines' x or y coordinates.

Code block 3 Line rasterization and sorting

```
input = viewpoint, circlepoint
for cell, touched by line[vp, cp] do
    True
end for
for cell == True do
    add cell to list of cells
end for
if line = quadrant_upperright then
    sort: x ascending & y ascending
else if line = quadrant_upperleft then
    sort: x descending & y ascending
else if line = quadrant_lowerright then
    sort: x ascending & y descending
else if line = quadrant_lowerleft then
    sort: x descending & y descending
end if
```

4. **Line-of-Sight query**
   Now we come at the core of the analysis. We arrive at this point with a list of pixels that describes a line between the viewpoint and a point on the circumference created by the circle about the viewpoint with the maximum viewing distance as its radius. The line of sight query is performed using the tangent method. An initial tangent is created between the viewpoint and the first point next to the viewpoint on the line. This tangent is defined by the following equation: $t_{cur} = slope * \Delta x + h_{viewpoint}$,
   where $slope = \frac{\Delta h}{\sqrt{\Delta x^2 + \Delta y^2}}$ and $\Delta x = \sqrt{\Delta x^2 + \Delta y^2}$.
   Then for every pixel on the line, the value of t_{cur} is evaluated, and if the value of t_{cur} is equal or lower than the elevation of this pixel, then the pixel is visible, and the tangent is updated with the parameters of this pixel, and the next pixel is evaluated. If the value of t_{cur} is not lower or equal to the elevation of the pixel, the pixel is invisible, and the tangent is not updated.

Code block 4 Line of sight query

```
t_cur = slope * d(viewpoint, pixel_1) + h_viewpoint
slope = (h_pixel_1 - x_viewpoint) / d(viewpoint, pixel_1)
for pixel in line of sight do
    if t_cur <= h_pixel then
        assign value [1] to pixel
        slope = (h_pixel - x_viewpoint) / d(viewpoint, pixel)
        t_cur = slope * d(viewpoint, pixel)x + h_viewpoint
    else
        assign value [0] to pixel
    end if
end for
```

In the figure below, there is a visual representation of this procedure. This is a profile of the Tasmania dataset, sliced from (515703.86454456096, 5263772.525086236) to (517705.77522456093, 5263772.525086236).

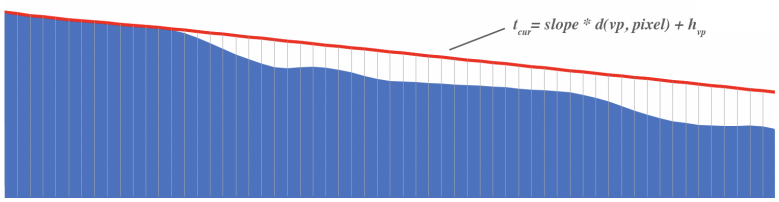


Figure 1: Profile with tangent

Assumptions made and their consequences

This section will describe several assumptions, shortcuts and other decisions made in the process that have consequences for the output of the analysis.

1. *Calculation of radius*
   The radius of the circle is calculated in *index space*, thus there is an uncertainty introduced to the viewshed horizon. However, since this is linked to the general resolution of the raster, the effect of this shortcut is generally not high.

1: **horizon(x,y)** $= index(x_{viewpoint} + maxdistance, y_{viewpoint}$
2: **radius** $= horizon(x, y) - viewpoint(x, y)$

---

2. *Circle construction & initial conditions*
   To make sure I only had to deal with the cells that define the *boundary* of the horizon, I decided to construct my circle as described above. This saves me from looping over the entire array, but has to be supported by giving all pixels a value of 3 (outside horizon) from the outset. However, for a discrete computer, a mathematical definition of the circle is not enough, therefore I had to discretize the range, and settled on an increment value of:

$$\frac{1}{\text{length of the array row}}$$

   This ensures enough "coverage". However, this results in a high number of duplicates for a low maximum viewing distance. This is a trade off that has to be considered. For the Tasmania dataset, the initial statement produces 181800 points on the horizon, but only 480 are needed. For the Cristo dataset, the initial statement produces 224640 points on the horizon, but only 1336 are needed. This could definitely be improved by linking in to a certain discretization factor needed for a specific resolution, because for large dataset this could eventually cost speed instead of save it.

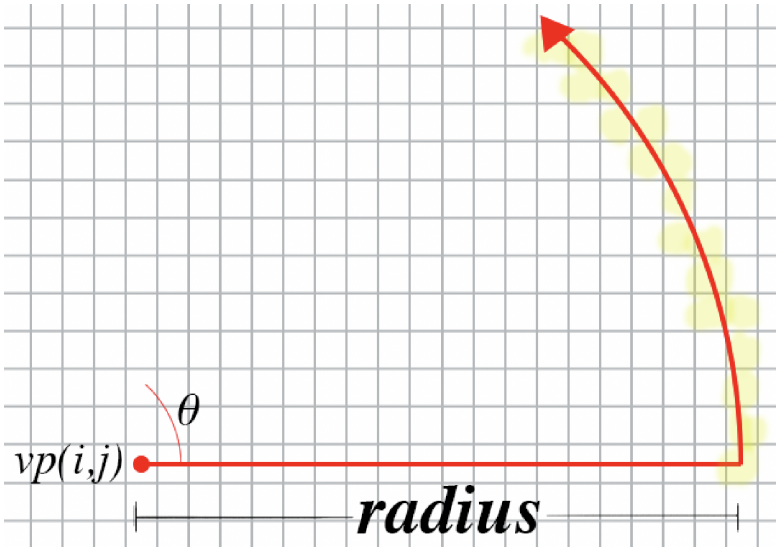


Figure 2: Circle construction

3. *Line rasterization with Rasterio*
   Most of the code to rasterize the viewlines was already present when the code was pulled from GitLab, however, I tweaked some of the presets. Namely, the rasterize function has a parameter *all touched*, which can be true or false. This determines whether *all* pixels that the line touches are being rasterized, or just the pixels whose centers interact with the geometry. This has effect on the viewlines, which can be seen in the figure below. However, due to the fact that when all pixels touched by the line also includes elevations that are not in fact in the line of sight, I have chosen to use the stricter definition, with only pixels whose center is within the polygon or that are selected by Bresenham's line algorithm used. This also has a benefit for speed, because per viewline, less pixels need to be evaluated.

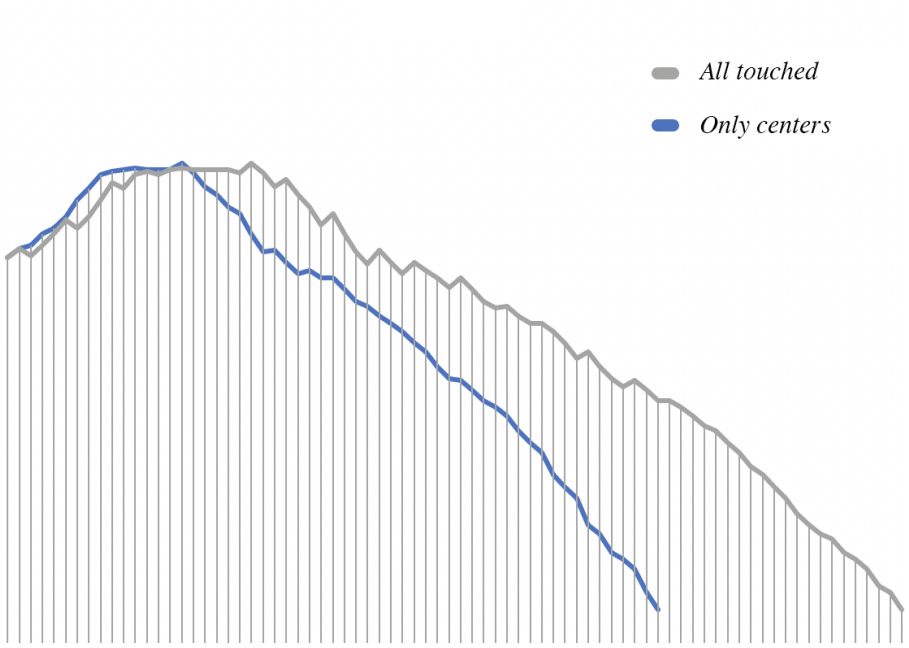


Figure 3: Comparison of profiles with different parameters

4. *Multiple visibility's*
   Another thing I had to take into account was that, especially near the center, when most of the viewlines share their pixels. To speed this all up, only pixels that have not already been "seen" by the algorithm are included for the line of sight query. Because if a pixel can be seen in one of the lines of sight, it can be seen in all the lines of sight (lines of sight do not go around the corner).

1: **if** $value_{pixel} \neq 1$ and $value_{pixel} \neq 2$ **then**
2: include pixel with LoS query
3: **end if**

This way, less pixels will be passed for LoS calculation by the algorithm, which will decrease running time. Furthermore, this approach also helps with multiple viewpoints. Otherwise the second viewpoints would just overwrite what the first viewpoint has calculated, which is not what is required, which is: if one DTM cell is visible only from one viewpoint, then it is visible.

Results, benchmarking and validation

To validate the results, I have done multiple comparisons with the "Profile Tool" plugin in QGIS, which show an accurate result for the viewshed analysis.

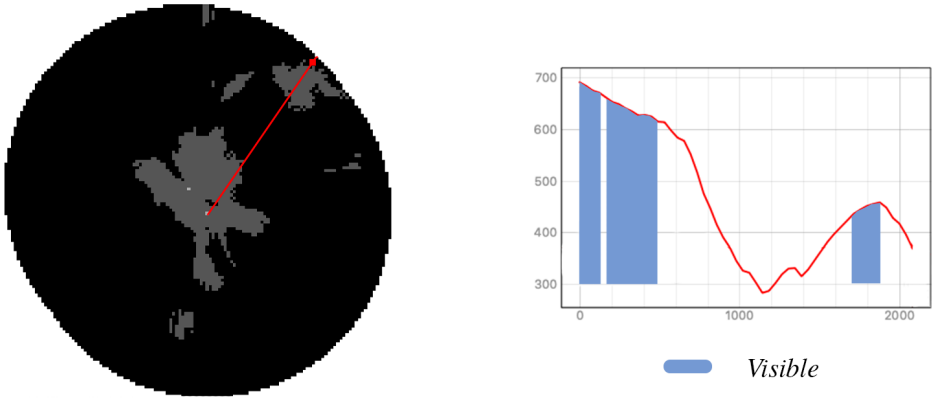


Figure 4: Comparison of viewshed analysis and profile extracted from DEM.

The figure above shows a intuitively correct result. Of course, the GeoTIFF's will be supplied with the report. Secondly, all these measures to speed up things seem to have worked, for the write times are not high. For the Tasmania dataset:

```
Viewshed file written to 'out-tasmania.tif'
--- 2.745 seconds ---
```

And for the Cristo dataset:

```
Viewshed file written to 'out-cristo.tif'
--- 7.179 seconds ---
```

Lastly, all is documented on: https://github.com/dumigil/viewshed_computation