# Calculating horizon altitudes with raster pyramids

## Background

In August, there will be an eclipse in Spain. The catch is that it'll be low above the horizon (only about 5°) so if I want to find the best place to observe it, I need to find a place where the apparent horizon, like hills and trees (mostly hills), does not get in the way.

However, this is not readily obvious from a topographic map. One gets an idea to look at hilltops but how do you know if a hilltop is good enough? Is this hilltop better than that hilltop? How do you quickly find a good hilltop that's reasonably accessible by car?

I set out to get an elevation map of (parts of) Spain, and for each pixel, calculate the altitude of the apparent horizon when looking from that point in the 285° azimuth, which is approximately the azimuth of the Sun during the eclipse.

The basic principle is simple. If you're looking in a certain direction and you have a sequence of equally spaced blocks with certain heights, you can just calculate the apparent altitude of each block, and take the highest one (Figure 1). That's your horizon altitude.

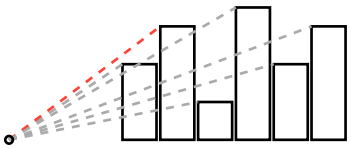

Figure 1: One-dimensional version of the problem

**Terminology:** I use the word "elevation" for the absolute height above the sea level. I use the word "altitude" for the apparent angle above the horizontal.

The problem is that the images are *big*. A typical size of a 30m-per-pixel GeoTIFF would be about 10k×10k pixels. See, you have to cover not only the area that you are interested in. You have to also cover those areas whose mountains could affect the apparent horizon in the area you're interested in. If a mountain can go about 2.5 km high, and you're worried about a 1° error in horizon altitude, you need to look at least about 150 km away. This has two consequences: it makes your raster big, and each line of sight will have to check many pixels.

150 km divided by 30 m per pixel gives us 5000 pixels (potentially down to only 3.5k if you're looking exactly diagonally but let's keep it simple). Hence if you want to process the entire picture, you need to process 10k × 10k × 5k = 500 billion pixels. Even for current hardware, that's a lot of pixels.

Nevertheless, that's what I did, because simplicity is king, and it takes about an hour to process the picture. Problem solved.

## Can we do better?

It turns out that a lot of times, the naïve algorithm traverses the line of sight, then it hits a mountain range, and then it still continues checking Every. Single. Pixel. Until the end of the line, even though the valley behind the mountain range is both (1) farther and (2) less elevated, so it cannot contribute to the apparent horizon. We should skip that.

To address this, I came up with an algorithm that uses raster pyramids. One is shown in Figure 2. A raster pyramid is a layered stack of rasters, where layer number 0 corresponds to the original raster, and going up by one step decreases the resolution by a certain factor, which I chose to be 2.

The key property of my pyramid is that the value of each cell is equal to the maximum of the values of the four cells underneath. This lets the algorithm quickly determine the maximal elevation of wide swaths of land, and if the maximum is low enough, the algorithm can skip processing those cells entirely.

Building the pyramid takes a bit of preprocessing time, but the tradeoff is certainly worth it.

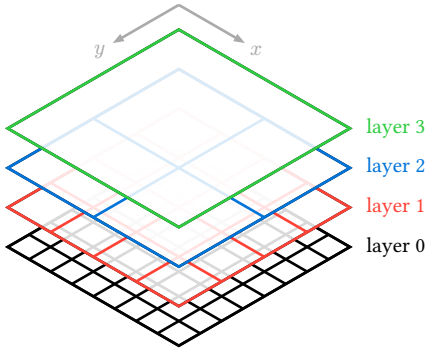


Figure 2: A raster pyramid

## The algorithm

Let's have a look at computing the apparent horizon altitude for a single pixel representing an observer situated in the corresponding place on the ground.

We start with variables $x$ and $y$ initialised to the pixel coordinates of the observer, and the thresholds to negative infinity.

```
// highest altitude seen so far
let mut max_tan_altitude = f32::NEG_INFINITY;
// highest elevation seen so far
let mut threshold = f32::NEG_INFINITY;

let (x_start, y_start) = <observer coordinates>;
let mut x = x_start;
while x > 0 {
```

To keep things simple, let's use the fact that the azimuth will be within 45° from due west. This allows us to iterate over the $x$-coordinates, and calculate the $y$ coordinate without skipping pixels. If your azimuth does not satisfy this, I'm afraid you'll have to rotate your raster until it does. I'm not complicating my algorithm just to fit *your* azimuth. Fortunately, rotation by multiples of 90 degrees will always be sufficient.

At this point in the while loop, $x$ is either the initial $x$-coordinate or the last-inspected $x$-coordinate. We don't want to inspect that point again in either case, so we decrement the on-the-ground $x$ coordinate and calculate the corresponding $y$ coordinate.

```
x -= 1;
let y = y_start - ((x_start - x) * DY / DX);
```

`DX/DY` is the rational number representing the tangent of the deviation of the azimuth from the west. In our case, azimuth 285° deviates 15° north from due west, so I set `DX = 268` and `DY = 1000`.

However, before inspecting these new coordinates, we first try to skip however much land we can. We initialise our pyramid coordinates at the ground level.

```
let mut x_pyramid = x;
let mut y_pyramid = y;
let mut level = 0;
```

Now we try to go as high in the layers as we can. That corresponds to finding the largest super-cell containing the current cell that does not exceed the elevation `threshold`. We divide the coordinates by 2 in every step because every layer has half the resolution (Figure 3).

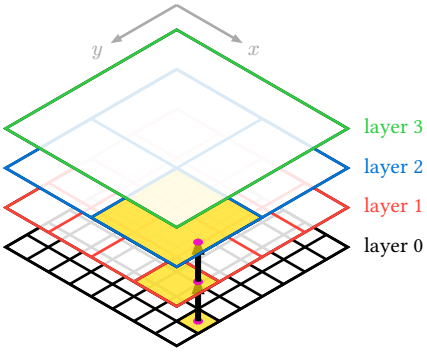


Figure 3: Going up

```
while level < pyramid.len()-1 {
    let x_candidate = x_pyramid / 2;
    let y_candidate = y_pyramid / 2;
    let candidate = (x_candidate, y_candidate);

    if pyramid[level+1][candidate] <= threshold {
        level += 1;
        x_pyramid = x_candidate;
        y_pyramid = y_candidate;
    } else {
        break;
    }
}
```

We've found the largest cell whose maximal elevation does not exceed the `threshold`, and therefore it is safe to skip entirely. Since we're only moving ~westward, skipping the (big) cell simply corresponds to decrementing `x_pyramid`.

```
if x_pyramid == 0 {
  // no more cells, break out of the entire loop
  break;
}
x_pyramid -= 1;
```

Figure 4 visualises this step.

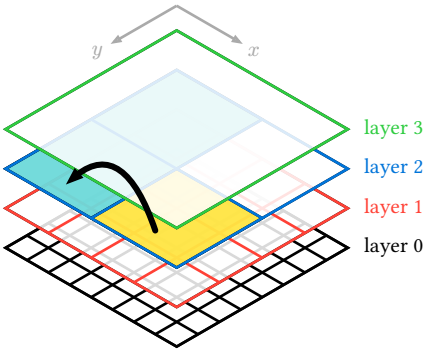


Figure 4: Advancing sideways at a higher level. We don't know the target $y$ coordinate, which is indicated by highlighting the entire column in teal.

We don't know the $y$ coordinate of our new (super-) cell at this level but we don't need it right now. We'll recalculate it again once we reach the ground layer zero.

This is also the reason that we decremented `x_pyramid` only once. We could do that once because we knew which cell we were *exiting*, and that it was safe to skip that entire cell. But since we don't know our new $y$ coordinate, and therefore we don't know which cell we've entered, we don't know that cell's maximum elevation, and we don't know whether it's safe to skip that one, too. We need to go back to the ground level 0 to find out.

What will the ground $x$ coordinate be? Well, given that in every line of sight, we traverse pixels ~westward, as we descend through the layers, the first pixel to inspect will be in the rightmost column. So in every step, we add 1 to the doubled $x$ coordinate to get to the rightmost column belonging to the given cell.

```
while level > 0 {
    x_pyramid *= 2;
    x_pyramid += 1;
    level -= 1;
}
```

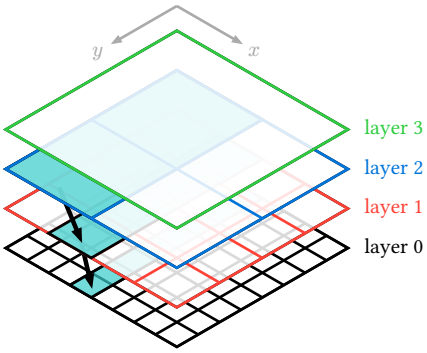


Figure 5: Descending back to ground level 0, conservatively taking the rightmost subcolumn in every step. Like in Figure 4, not knowing the exact $y$ coordinate is illustrated by highlighting the entire column of cells.

This places us back on the ground in the rightmost column of the big cell. Let's recalculate the $y$ coordinate as well.

```
x = x_pyramid;
let y = y_start as i32 - ((x_start - x) as i32 *
params.dy / params.dx);
```

And this is the place whose elevation we need to check next! We have possibly skipped many pixels. It's also possible that we skipped zero pixels. But in any case, we've moved at least one pixel to the left in total because we decremented $x$ in the first step of the loop.

```
let dx = x_start - x;
let dy = y_start - y;
let distance = METRES_PER_PX * \
  ((dx*dx + dy*dy) as f32).sqrt();
if distance > MAX_DISTANCE_M { break; }
let dh = pyramid[0][(y, x)] - elevation_start;
let tan_altitude = dh / distance;

// update the thresholds
max_tan_altitude = \
  max_tan_altitude.max(tan_altitude);
threshold = max_tan_altitude * distance \
  + elevation_start;
```

You might think that we could simply set `threshold = pyramid[0][(y, x)]` but don't forget that `max_tan_altitude` can actually be a lot higher. We derive the threshold from the maximum altitude we've seen so far, not from whether the currently inspected spot will obscure the terrain behind it.

Therefore `threshold` can never decrease in the process, and we skip more and more land – as we should.

```
}

max_tan_altitude.atan().to_degrees()
```

And that's the answer!

## How fast is it?

This algorithm shortens the one-hour job to process all pixels of the GeoTIFF to about 45 seconds on my machine. That's about ~80× speedup.

## Alternatives

Currently, if one of the four sub-cells' elevation exceeds the threshold, you can't continue rising in the layer hierarchy, even if the

excessive elevation is in a completely different direction than where you're travelling.

Since the azimuth in my program is the same across the entire raster, we could probably rotate the raster to align its scanlines with the line of sight, and then build a binary maximum tree over each scanline independently. The advantage is that neighbouring scanlines' elevations would not interfere with the current scanline, and we could skip more pixels each time.

The downside is that you'd have to build a separate index for every scanline, instead of reusing the same 2D index built once for all pixels. You'd also have to rotate the bitmap by an arbitrary angle (once), which I expect would be more expensive than 90-degree rotations. Not sure how this overhead would compare with the gains from skipping more efficiently. Also, this is not usable if you want to be more precise and trace different azimuths from different places.