The skyline problem

31 August 2014 programming

You are given a set of n rectangles in no particular order. They have varying widths and heights, but their bottom edges are collinear, so that they look like buildings on a skyline. For each rectangle, you're given the x position of the left edge, the x position of the right edge, and the height. Your task is to draw an outline around the set of rectangles so that you can see what the skyline would look like when silhouetted at night.

How shall we proceed? If you're drawing a blank, it's always good to get a really awful solution on the table right away so that you have something to think about and improve upon.

The first thing I thought of was to construct a 1-dimensional heightmap. The idea is to create an array of height values and write each rectangle onto it. Without worrying about the details of mapping rectangle coordinates to pixel array indicies, the code will look something like this:

```
for each rectangle r:
    for each heightmap cell c starting at r.left and ending at r.right:
        c gets the max of r.height and the previous value of c
```

00888888880444777779999999999444444666666666666666660

OK, so this works as a first attempt at a solution. What, specifically, is wrong with it?

You can see from the animation that the skyline constructed from the heightmap isn't quite correct. The edges of the rectangles don't line up perfectly with the array cells, so there is a small amount of error in the shape of the skyline. In fact, it is easily shown that the only way to guarantee zero error is to use a heightmap with the same resolution as the final rendered image. This means that the running time of your algorithm depends not only on the number of given rectangles, but also on the resolution of the output image.

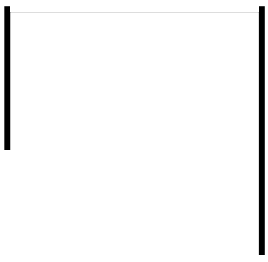
Of course, unless you're using a vector display, it's inevitable that at some point you will have code looping on the order of the resolution of the output image, just to draw the line segments one pixel at a time. I'm inclined to not worry about this cost. If you're writing code in Perl, for example, your concern is to do as little as possible in Perl and offload as much work as possible to the drawing library, which is likely compiled and heavily optimized. Only when the line-drawing code isn't much faster than your application logic does it start to make sense to use a raster approach like a heightmap.

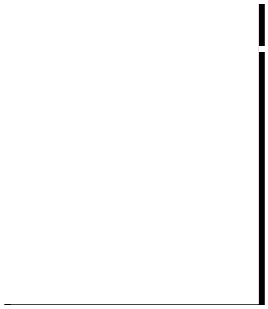
What now? If the chief weakness of the heightmap approach is the sheer number of points to deal with in application code, maybe we can reduce the number of points in play. Now that we think about it, the skyline is made up of horizontal line segments, interrupted in only a few places. In fact, the only time the skyline can change its y position is at the left or right side of a rectangle. It's clear now that if we find the height of the skyline at each of these "critical points" on the x axis then we will have completely determined the shape of the skyline. At each critical point, you just go up or down to the new height and draw a line segment to the right until you reach the next critical point.

So how do we find the true height of the skyline at each of these critical points? Well, we already have this heightmap approach, so let's try doing something similar. This time, instead of printing the rectangles onto a heightmap array with an entry for each pixel, let's print the rectangles onto an array with an entry for each critical point! This will eliminate the problem of dealing with too many points, because we're now dealing with the minimum number of points necessary to determine the skyline.

Here's a first stab at what the code would look like:

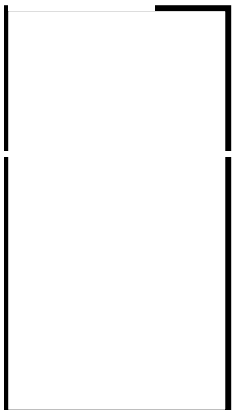
```
for each rectangle r:
    for each critical point c:
        if c.x >= r.left && c.x < r.right:
            c.y gets the max of r.height and the previous value of c.y
```





Looks good! So now we have a working $\mathcal{O}(n^2)$ solution to exactly the problem we were trying to solve, with no error. Can we achieve a better running time? It occurs to us that we don't really need to look at *every* critical point when printing a rectangle, but rather only those critical points below the rectangle in question.

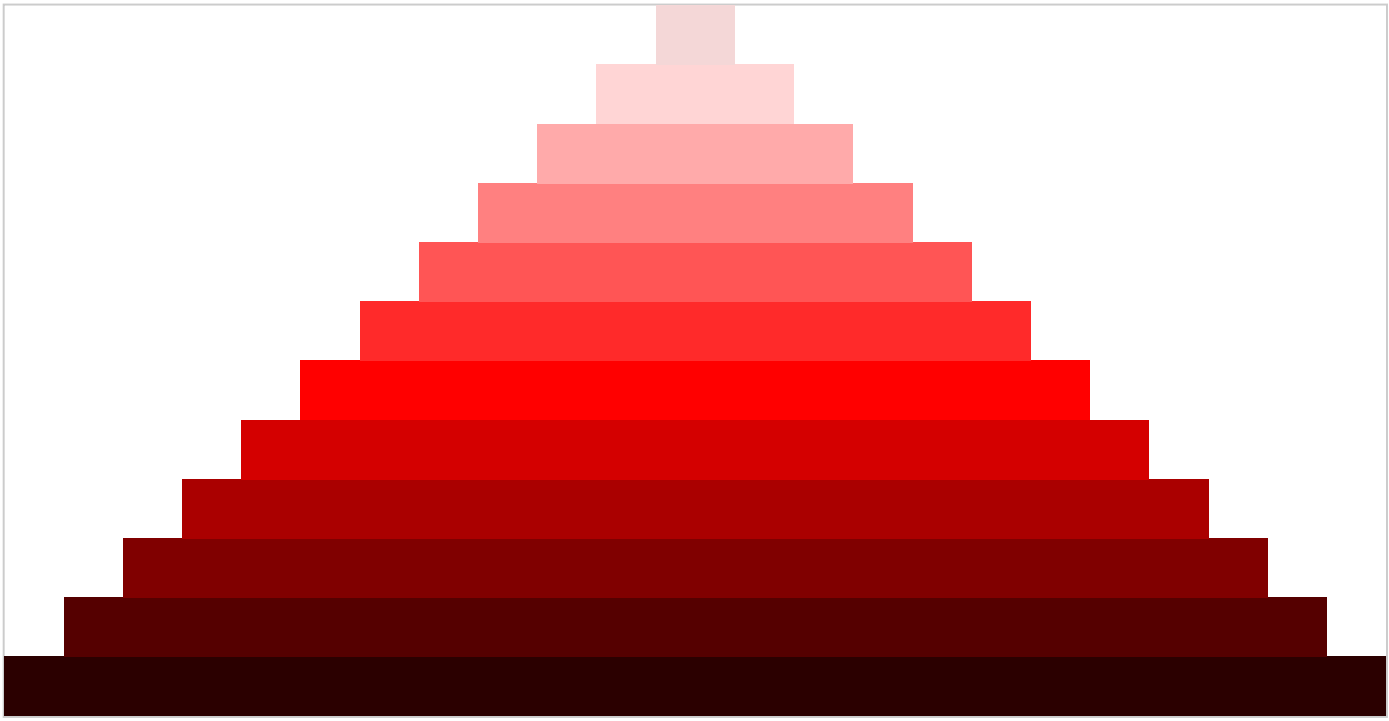
```
for each rectangle r:
    for each critical point c below r (except the one at r.right):
        c.y gets the max of r.height and the previous value of c.y
```



This optimization depends, of course, on being able to efficiently find which critical points are subtended by each rectangle. This is easily done by sorting the critical points. For example, if we want to find the critical points subtended by the magenta rectangle, we start at the left side of the magenta rectangle and scan to the right, accumulating critical points until we reach the right side.

0 1 2 3 4 5 6 7 8 9

Unfortunately, this isn't an asymptotic improvement in the worst case. It's still $\mathcal{O}(n^2)$ given something like the following configuration:



At this point perhaps no ideas jump out at you about how to improve the algorithm's performance further. Let's try perturbing the solution we have in order to see what might present itself. What if, instead of looking at each critical point for each rectangle, we look at each rectangle for each critical point? This is the same code as before, with the loops switched:

```
for each critical point c:
    for each rectangle r:
        if c.x >= r.left && c.x < r.right:
            c.y gets the max of r.height and the previous value of c.y
```

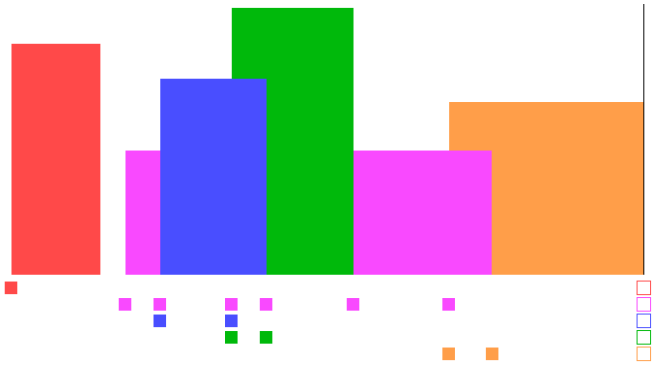
Again, we don't really need to consider all rectangles, only the ones above the critical point in question:

```
for each critical point c:
    for each rectangle r above c (not including the right edge of rectangles):
        c.y gets the max of r.height and the previous value of c.y
```

So, given a critical point, how do we efficiently find all of the rectangles above it? This requires a different strategy than before. Before we turned the problem inside out, we needed to find all of the critical points between the left and right sides of the given rectangle. Now, we need to find all of the rectangles with a left edge to the left of the given critical point and a right edge to the right of the given critical point.

What if we begin at the critical point and go left looking for left edges, and also go right looking for right edges, and then intersect the two sets of rectangles? That would work, but, again, it's $\mathcal{O}(n^2)$ in total to do this for every critical point.

A better approach is to simply scan across the skyline's sorted critical points from left to right, keeping track of an *active set* of rectangles as you go. When you reach a critical point, the active set is updated and then the critical point gets assigned a copy of the current active set of rectangles. By the end of the pass, each critical point will know about all of the rectangles above it.



Now that we're able to scan through the critical points and consider only the "active" set of rectangles at each critical point, an interesting opportunity presents itself. Our current solution can be written as:

```
for each critical point c
    c.y gets the height of the tallest rectangle over c
```

This is no longer obviously $\mathcal{O}(n^2)$. If we can somehow calculate the height of the tallest rectangle over c in faster than $\mathcal{O}(n)$ time, we have beaten our $\mathcal{O}(n^2)$ algorithm. Fortunately, we know about a data structure which can keep track of an active set of integer-keyed objects and return the highest one in $\mathcal{O}(\log n)$ time: a heap.

Our final solution, then, in $\mathcal{O}(n \log n)$ time, is as follows. First, sort the critical points. Then scan across the critical points from left to right. When we encounter the left edge of a rectangle, we add that rectangle to the heap with its height as the key. When we encounter the right edge of a rectangle, we remove that rectangle from the heap. (This requires keeping external pointers into the heap.) Finally, any time we encounter a critical point, after updating the heap we set the height of that critical point to the value peeked from the top of the heap.

Related Posts

- 24 Jul 2018 » Who creates money?
- 16 Jul 2016 » Developer interview questions
- 05 Sep 2014 » Covariance and contravariance rules in Java
