Notebook2

loli

February 13, 2018

Contents

1 Edge Detecotr **1**

2 General Notes on performance **6**

1 Edge Detecotr

1. Sobel
   (a) Describing Sobel
   - This filter works rather similar to linear blur filter from notebook1, as you are just convoluting a filter over an image.
   - I actually stated what the matrix was for the Sobel filter in Notebook1, but I shall restate it here and go into why this matrix makes sense for an edge detector.
   - let $S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{pmatrix}$ and let $S_x = \begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix}$
     - Lets consider any pixel at (k,l) with the S_y and S_x filter.
       - the S_y filter will remove the contribution of the pixels on the same row as the (k,l) pixel, and S_x will do the same to the vertical pixels as S_y
       - Now the interesting part are on the top row of S_y (left column of S_x) and bottom row of S_y (right column of S_x). as if you look at these two, if the color is the same in the neighborhood of (k,l) then the value should be 0, as the top will cancel out the bottom (or in the S_x case the left would cancel out the right)
       - This would only leave pixels in which there is a difference between the top and bottom for S_y and a difference between the left and right for S_x
2. Implementing Sobel

- This implementation actually fell out of the discussion session, as they suggested (since my solution was so general) that I change my Gaussian kernel into the Sobel one, below is how I ended up defining the kernel.

```
sobelEdgeX :: Num a ⇒ Stencil DIM2 a
sobelEdgeY :: Num a ⇒ Stencil DIM2 a
sobelEdgeX = [stencil2| -1 0 1
                        -2 0 2
                        -1 0 1|]

sobelEdgeY = [stencil2| -1 -2 -1
                         0  0  0
                         1  2  1|]
```

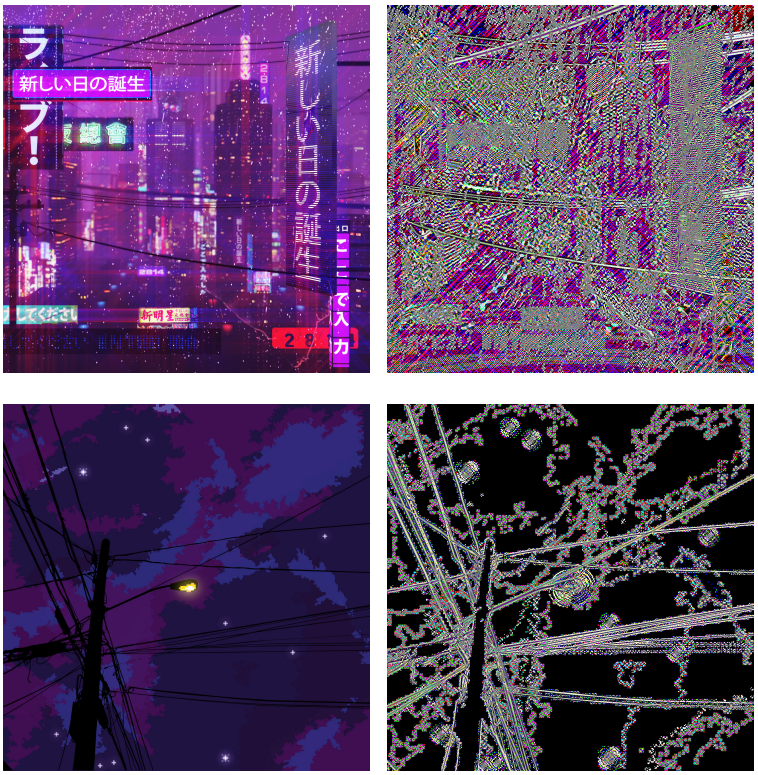
- Now that we have the kernel up, lets run them and compose them over the image (the GHC fuses these two kernel passthroughs into 1).

```
sobelX :: (Source r b, Num b) ⇒ Array r DIM2 b → Array PC5 DIM2 b
sobelY :: (Source r b, Num b) ⇒ Array r DIM2 b → Array PC5 DIM2 b
sobelX = mapStencil2 BoundClamp sobelEdgeX
sobelY = mapStencil2 BoundClamp sobelEdgeY


sobel :: (Source r b, Num b) ⇒ Array r DIM2 b → Array D DIM2 b
sobel = R.delay . sobelX . sobelY
```

- I rather like running these on colors, even though for edge detection it doesn't matter, so I ended up generalizing my code for blurCol (which allows me to blur colored images) so that I can pass it a kernel to filter the image with.

```
with2d f = flip reshape . g . fromList . fmap f . slices <*> R.extent
  where g (RGBA a b c d) = interleave4 a b c d
        g (RGB a b c)    = interleave3 a b c
        g (Grey a)       = a

edgeCol :: (Fractional e, Source r e) ⇒ Array r DIM3 e → Array D DIM3 e
edgeCol = with2d sobel
```

  - here with2d's logic is already kind of explained with the `blurCol` logic in notebook1, basically it just makes 2d slices of the RGBA bands of an image and applies our filter f (this case sobel).
    * for more detail see notebook1
- With all this being done we can actually use sobel and see what images we get (note I do filter pixels later, but we'll get there).
  - I ended up trying it on two images in the current state.
  - The top left image is the same image used in part1.
  - if you were to look at the full image. "Color-save-proper-colors.png", you'll notice there is a lot of noise in this image so the image below won't be surprising.



  - The image on the top right is labeled "Blur-into-Sobo-color.png", as I ran my Gaussian blur before the Sobel edge detector.
  - Even though it looks rather cool, it really doesn't give much useful information (I can see some edges in there!).
  - Another image I ran this on was "Colors-wires.png" which can be seen on the bottom left.
  - The right shows the output, which does look like edge detection does take place on a not so noisy image.
- Motivated by the lack of good edges in the noisy image, I decided to remove all pixels below a certain value as seen below.

```
-- type signatures simplified for pdf
filterPixelsS :: Array r DIM3 b → b → Array D DIM3 b
filterPixelsS arr min = R.traverse arr id (passThresh min averaged)
  where
    (Z :. _ :. _ :. k) = R.extent arr
    averaged           = R.sumS $ R.map (/ fromIntegral k) arr

filterPixelsP :: Array r DIM3 b → b → m (Array D DIM3 b)
filterPixelsP arr min = (R.traverse arr id . passThresh min) <$> averaged
```

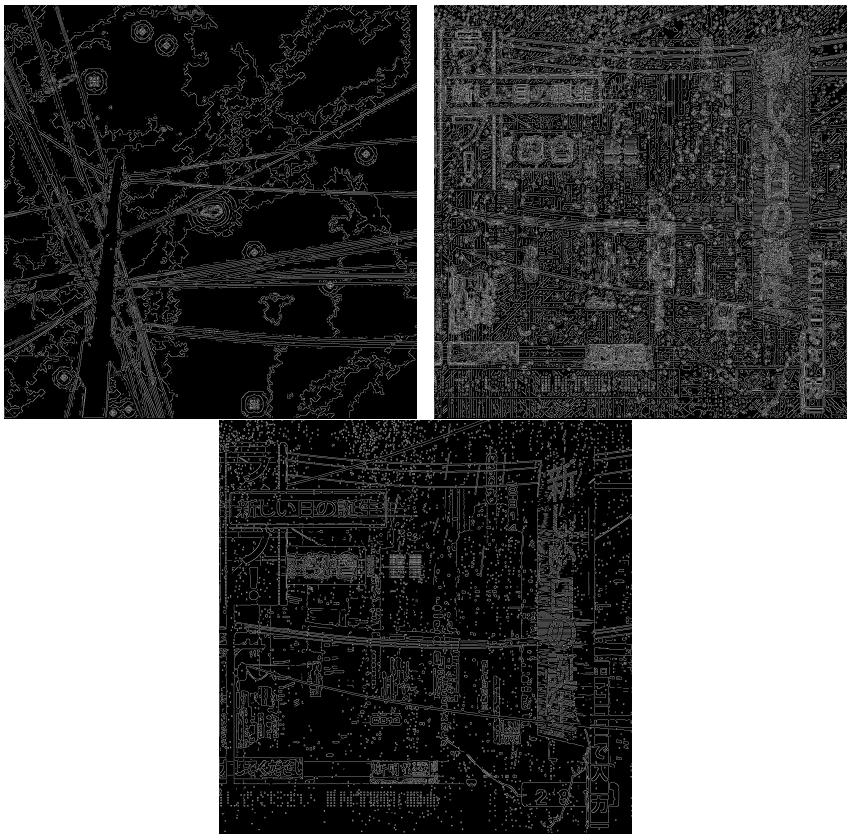
```
  where
    (Z :. _ :. _ :. k) = R.extent arr
    averaged           = R.sumP $ R.map (/ fromIntegral k) arr

-- helper for filterPixelP and S
passThresh min avg f sh@(Z :. i :. j :. _) | avg ! ix2 i j >= min = f sh
                                           | otherwise            = 0
```

  - Here we really traverse the old image and compare it with the average value at that point, and if it's below our threshold we just say it's 0. Also note that there is `filterPixelsS` and `filterpixelsP` because I noticed that all my cores weren't being used when the S version was being run
    * I would show "colores-blur-then-sobel-100", "colores-blur-then-sobel-210", and "Color-wires-blur-then-sobel-100-min.png", but sadly they don't show up too well on the pdf, but it was rather interesting to see what details went away when the min value was changed.

Canny

1. Describing Canny
   - The canny edge detector can be broken into a few discrete steps that I've sourced from Wikipedia
     (a) Find the intensity gradients of the image
     (b) Apply non-maximum suppression to get rid of spurious response to edge detection
     (c) Apply double threshold to determine potential edges
     (d) Finalize the detection of edge by suppressing all other edges that are weak and not connected to strong edges
2. Implementing Canny
   - There is a package for Haskell, called `friday` that has a cannyEdge detector built in, so I ended up using that.

     ```
     cannyEdge :: Int -> Int32 -> Int32 -> Image PixelRGB8 -> I.Grey
     cannyEdge radS min up = canny radS min up . toGrey . toFridayRGB
     ```
   - I ended up using my Gaussian blur to preprocess the image, as the library does not do that for me (edge detectors are quite sensitive to noise, and blurring helps with that).
   - the Libraries version only works for Grey images sadly, however it did a better job than the Sobel detector at edge detector on the 2 images I compared it with.
     – The top two were run with radius 5, while the bottom one was run with radius 3.
     – For a really noise image at 1500x1500 size, that a smaller radius is preferable.



    * Further tests with a 3872x2592 image of trees also confirms this as at radius two, the edge detection was rather good, but at radius 5 I would need to filter out a lot more (this could just be caused with brighter values sneaking its way in!).

2 General Notes on performance

- In general I'm rather proud of the performance. The Sobel filter with all colors running take less than 40 seconds to run on a 1500x1500 image.
- Also the code provided hammers all 8 of my cores without stopping (except when I run `filterPixelS`, but I don't ever run this)
  - The best part about this, is the only word I speak of parallel code is `filterPixelP` and the 2 `computeUnboxedP` in the main function.
- The speed of this allows me to quickly test out what happens if I say only use the SobelX but not SobelY