Technical University of Denmark

02132 Computer Systems

Assignment 01
Cell detection and counting algorithm in C.

Rath, Tommy Jonas
s204438
Tommy Rath

Laforce, Erik Aske
s204505
Erik

Course lecturer
Luca Pezzarossa

4. october 2020

Work distribution

Code

Gray scaling and binary threshold were combined into one function that was discussed together and implementet together.

The work for the algorithms loop was divided so Tommy took framing and Erik took erosion.

Functions relating to creating images from the 2d array we worked with were done by Erik and Tommy made batch files to A: input all the diffrent samples and B: test a bunch of threshholds and framesizes on the first sample from each dificulity in order to decide what the optimal threshhold and framesize would be.

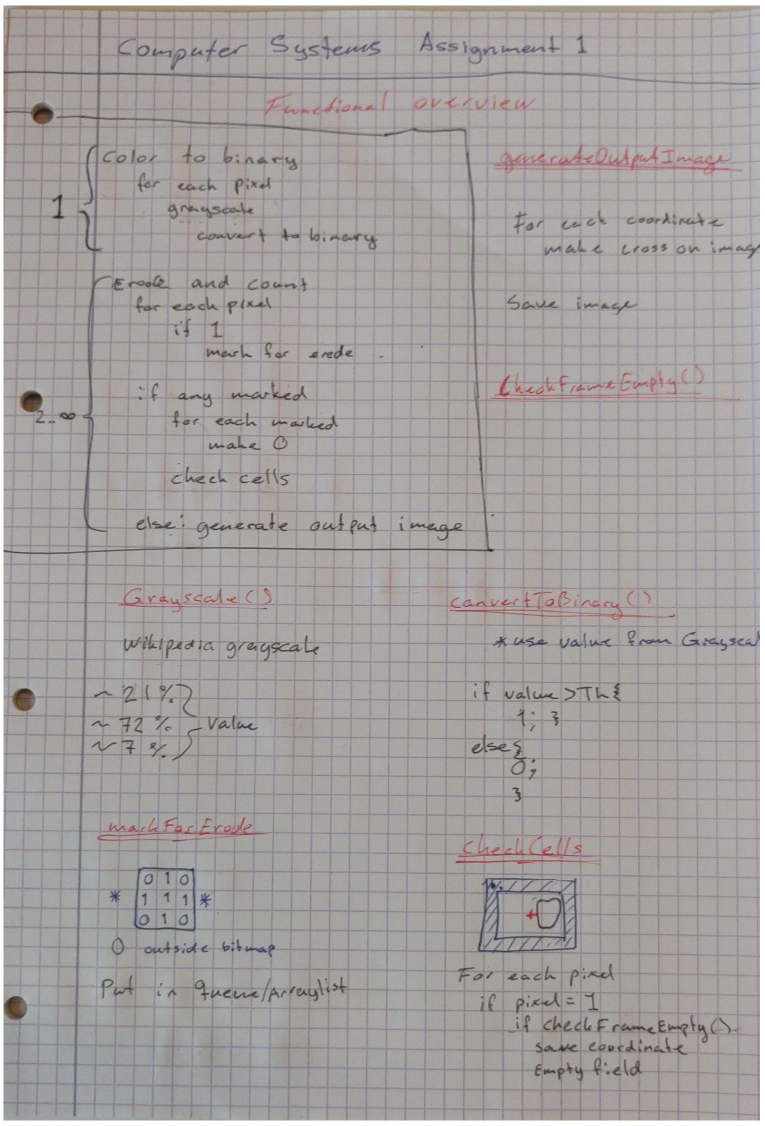
Functional tests to see if the code worked as it should were designed by Erik.

Optimizations were done throughout the implementation and afterwards and all ideas for improvements were discussed together so little knowledge stands on who exactly did what Optimizations.

Report

In regards to the report, the "Work distribution" section was done together as we discussed what parts of the code we each had done.
The "design" section: Tommy and Erik
The "implementation" section: Erik
The "Optimization and enhancements" section: Tommy and Erik
The "Test and analysis" section: Erik

Design



Computer Systems Assignment 1

Functional overview

```
1    Color to binary
       for each pixel
         grayscale
           convert to binary

2..∞ Erode and count
       for each pixel
         if 1
           mark for erode

       if any marked
         for each marked
           make 0

         check cells

       else: generate output image
```

generateOutputImage

```
For each coordinate
  make cross on imag
Save image
```

CheckFrameEmpty()

Grayscale()

Wikipedia grayscale

```
~ 21 %
~ 72 %   Value
~ 7 %
```

convertToBinary()

* use value from Grayscal

```
if value > Th {
    1; }
else {
    0;
    }
```

markForErode

0	1	0
1	1	1
0	1	0

0 outside bitmap

Put in queue/Arraylist

CheckCells

```
For each pixel
  if pixel = 1
    if checkFrameEmpty()
      save coordinate
      Empty field
```

Figure 1: Original plan for code

The design was originally made as shown on Figure 1. Each part of the algorithm was made into a function except for grayscaling and binary threshhold which was made into one function instead of two. Certain parts of the code have also been given their own functions such as checking the pattern for erosion being moved to a function called checkNeighbor().

Implementation

GrayScaling and threshold

The two functions are made into one to save time. The three color channels each have their own modifier which originally was taken from the wiki site for grayscaling but later they have been changed to 0.3333 since how acurate it looks to the human eye doesnt matter

Erosion

The erosion works by first looping over each pixel and marking them if they dont match the pattern and then afterwards going over all pixels again and removing the marked. This is to avoid any removal of pixels affecting the checks of future pixels.

Framing

The frame does not work by moving around the screen and checking if something is inside and the frame is empty. Rather it goes over every pixel and checks if it is 1 and if it is then it checks if the frame with that pixel in the middle is empty. This removes the need to check if something is within the frame. Since it only checks once it knows something is in the middle. It also removes any time benefits from bigger frames and the option to skip an area you know is empty, but the time saved seems worth it.(There are no tests to confirm this, but both systems could be set up and tested against eachother.)

Output Image

The creation of the output image as of now is in-effecient space- and time-wise. This is because the entirety of the input image gets copied over to an output image and plus signs are then placed on top of the output image. Instead plus signs could just be added to the input and saved. This comes from previous versions where debugging images were used to show the progress of the code and has been left in for easier debugging if needed.

Running the code

Two files are made to be compiled with the code. One is runner.c which compiled with the rest of the code creates an excecutable that counts cells and prints the coordinates. The other is tester.c which when compiled makes an excecutable that tests the functionality of certain functions within main.c.

Optimizations and enhancements

2d array for binary bmp values

One of the bigger optimizations that was done was saving the binary image as a 2d array, so that we can simply read the color of a pixel from an array, which is a lot faster than reading the bitmap each time we want to check a single pixel. The tradeoff however, is that all the binary values contained in the array will take up some space en the computers RAM.

Avoiding division

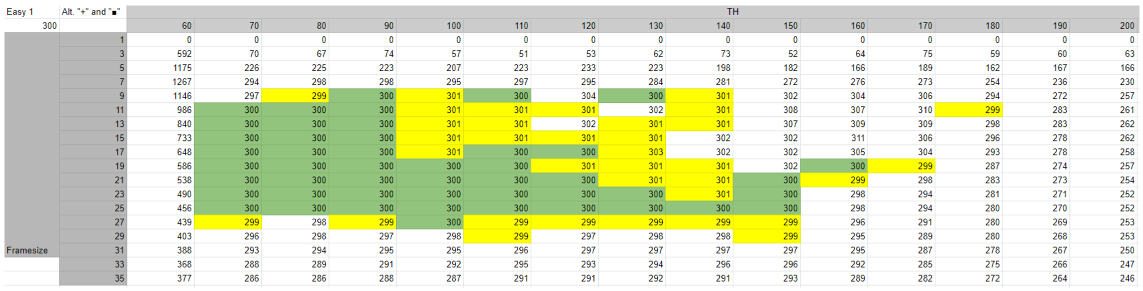
In the process of grayscaling an easy method to avoid division is multiplying each color channel with 0.3333 instead of dividing by 3. This of course will only add up to 0.9999 of the color, but so little loss means very little and if it was a problem more 3's could be added.

Alternating "■" and "+" patterns:

We have decided to go with a 3x3 grid for the erosion pattern since larger grids seemed like they would create too fast erosion. To optimise in a way that doesnt resize the erosion weve looked at diffrent patterns strengths and weeknesses. The plus pattern is a very basic pattern that slowly erodes the picture. It does have a flaw though. Two cells placed askew from eachother could accidentally be missed due to the erosion making a diagonally angled line out of them that is larger than the framesize.
This is where the "■" shape comes in. By filling the erosion pattern is a bit more agressive and since the corners are 1's it prioritizes removing 1's askew from 0's. This means it can handle the problems the "+" pattern has. This one does also posses a flaw though. Since its a little more agressive two cells directly next to eachother could get removed before they get seperated.
For this reason weve opted for switching between the two patterns which has given improved results on the sample HARD1.

Optimal threshhold and framesize with batch scripts

In order to find out what the optimal threshhold and framesize would be, manually counted the number of pixels in respectively EASY1, MEDIUM1, HARD1 and IMPOSSIBLE1, bitmaps, after which we created a batch testscript, that executed the program with every combination of threshhold (from 60 to 230, incrementing with 10) and framesize (from 1 to 35 incrementing with 2), while putting the values into a table. Afterwards we would compare the table-values to the munually counted ones.
We saw that many of the values that were equal to the actual cell count in "EASY1" were very close to eachother in the table. elsewhere in the table we saw similar results, but upon closer Examination of the output images, it seemed, that some of them was a result of some cells not being found, and others being detected multiple times, meaning that the mistakes cancelled eachother.

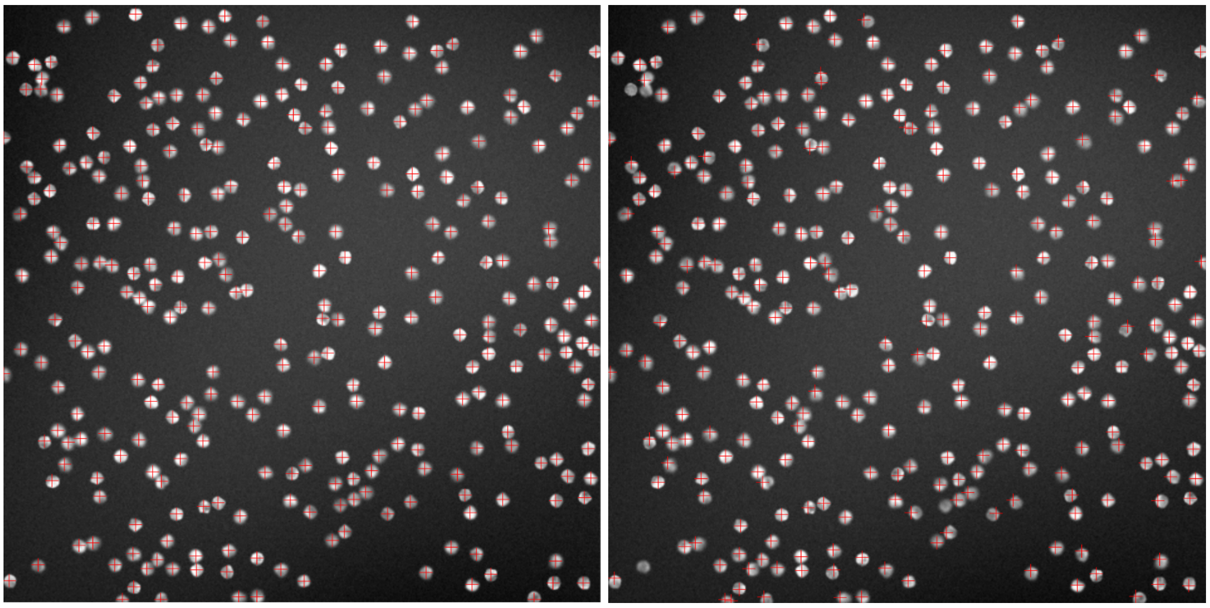


Easy 1	Alt. "+" and "■"	TH														
300		60	70	80	90	100	110	120	130	140	150	160	170	180	190	200
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	3	592	70	67	74	57	51	53	62	73	52	64	75	59	60	63
	5	1175	226	225	223	207	223	233	223	198	182	166	189	162	167	166
	7	1267	294	298	298	295	297	295	284	281	272	276	273	254	236	230
	9	1146	297	299	300	301	300	304	300	301	302	304	306	294	272	257
	11	986	300	300	300	301	301	301	302	301	308	307	310	299	283	261
	13	840	300	300	300	301	301	302	301	301	307	309	309	298	283	262
	15	733	300	300	300	301	301	301	301	302	302	311	306	296	278	262
	17	648	300	300	300	301	300	300	303	302	302	305	304	293	278	258
	19	586	300	300	300	300	300	301	301	301	302	300	299	287	274	257
	21	538	300	300	300	300	300	300	301	301	300	299	298	283	273	254
	23	490	300	300	300	300	300	300	300	301	300	298	294	281	271	252
	25	456	300	300	300	300	300	300	300	300	300	298	294	280	270	252
	27	439	299	298	299	300	299	299	299	299	299	296	291	280	269	253
	29	403	296	298	297	298	299	297	298	298	299	295	289	280	268	253
Framesize	31	388	293	294	295	295	296	297	297	297	297	295	287	278	267	250
	33	368	288	289	291	292	295	293	294	296	296	292	285	275	266	247
	35	377	286	286	288	287	291	291	292	291	293	289	282	272	264	246

Figure 2: Table of cellcounts for "EASY1"

The rest of the tables can be seen via this link

In the table it is clear, that many different combinations of framesize and cellcount resulted in the return of the desired cellcount, but when looking at the pictures below **(Figure:3)**, it is clear that not all correct cellcounts mean that the cells were countet correctly



(a) All cells detected correctly (Threshhold 90, framesize 11)

(b) Found the correct amount of cells, but some were countet multiple times and some were missed (threshhold 160 framesize 19)

Figure 3: Correctly countet cells versus Random correct cellcount

The same is true for more of the values in the outer edges of the green areas in the table (green means correct cellcount). This tells us, that the optimal settings for threshhold and framesize must be somewhere inbetween other examples of correct results. this table is only an example using one of the easiest samples, and the optimal inputs may vary depending on the difficulty of the input bitmap.

Test and analysis

Functional tests

Functional tests can be found in tester.c where a bunch of really simple tests for functionality has been made. Its tests are very vague though since many of the things could change such as frame size erosion patterns and such.

Time test

Whenever one runs the code it opens up timeTest.txt and starts writing the time diffrent functions take. With the optimizations weve gotten both erosion and frame time down under 10 ms and a total time under 1s. The reason it still takes so much time is a result of the 300ms it takes to read the input image and the almost 300ms it takes to copy into the output image, add red plus' and write the image.

Space analasys

Currently three big arrays take up space. This could be reduced to 2 by using the input array as the output, but as mentioned earlier it existed for debugging purposes.