02132 Assignment 1 report

Software implementation of a cell detection and counting algorithm in C

05-10-2025

Group 62

Anton Brinch - S246072
Asger Ditlev Otte - S245803
Emil Jensen - S240617

Work distribution

Explain here who has done what, for both implementation and report.

We have made a conscious effort to share the workload equally. By working side by side, we were able to identify problems together and discuss possible solutions. At the same time, we also assigned each other smaller, individual tasks when appropriate. This approach proved very effective as it prevented all three of us from working on simple problems that could easily have been solved by just one person.

Design

Explain here what the design process was. Explain how you structured your code (e.g., divide functionality into functions, decide the functions prototypes, etc.). Explain how you decide to represent and store data (e.g., what representation, what buffers to use, etc.). Motivate the design decision you made.

The overall design of the project we chose was a modular approach, where each major image-processing task was implemented as separate functions. The purpose of this was to make the program easier to maintain, debug and expand while keeping the main program clear and readable.

Our code is divided into two main modules:

1. Main.c which controls the program flow and handles the image input/out also the relevant processing functions.
2. Functions.c contains the implementations for all image-processing functions.

This approach makes it possible to reuse and test each function independently, which was in our case a very efficient design pattern during the development for our embedded system.

The Main functionality is divided into the following functions:

1. Invert() inverts image colors (used for testing and debugging).
2. convert_to_greyscale() converts the RGB image into a single grayscale channel
3. Binary_threshold() converts the grayscale image into a binary image using threshold value
4. Basic_erosion() performs erosion using a cross structuring element to separate connected cells

5. detect_spots() scans the eroded image for white clusters representing cells, registers their coordinates and prevents double counting by blacking out detected spots.
6. generete_output_image() overlays a visual Teemo figure on each detected cells coordinates and saves the final BMP output.

The cell coordinates are stored in a fixed-size array.

```
int coords[MAX_COORDINATES][2];
```

Each entry holds the (x,y) coordinates of a detected cell. The upper limit prevents memory overflow and provides a bound for memory usage.

The main.c file defines a simple and linear pipeline of executions:

1. Load image
2. convert to grayscale
3. Apply binary threshold (Which is otsu method in our case)
4. Repeatedly erode the binary image until no further changes occur
5. Detect cells and store their coordinates
6. Overlay detection markers on the original image
7. Save the output and print CPU timing and further statistics.

As you can read, this modular design ensures clarity where each function only handles exactly one task, Reusability where we can change and/or reuse any step without changing other parts of the code, It's easy to maintain and debug due to it's easy to isolate and fix issues. Our program is efficient due to the simple loops and static memory with our embedded system, and it's easy to further scalability which we experienced by adding otsu method instead of binary threshold during the later stages of the development.

Implementation

Briefly discuss the implementation in C of your design. Explain how you have exploited the C language in the context of embedded systems to implement the algorithm. You can include some code snippets if these are relevant to explain certain aspects of the implementation.

The implementation follows the structure described in the design section. But to dive deeper in the code we can talk a bit about the core routines of function.c

With our Greyscal conversion we took the average of the RGB values using integer arithmetics and wrote it into all three color channels to keep the output compatible with the BMP format.

```
output_image[x][y][c] = ((r + g + b) * 85) >> 8;
```

This avoids floating point division and makes the loop faster.

After otsus method has determined the global threshold, each pixel is converted to black or white using a very simple comparison. All three RGB channels are set at once using the standard function memset, which is faster than assigning each channel separately:

```
unsigned char post_threshold_value = (input_image[i][j][0] > threshold)
? 255 : 0;

memset(output_image[i][j], post_threshold_value, BMP_CHANNELS);
```

Otsus method calculates the optimal global threshold based on the variances between foreground and background.

Erosion is implemented using a 3x3 cross structuring (Center and four neighbors). A temporary snapshot of the image i created with memcpy before each pass to avoid reading and writing to the same data simultaneously. The algorithm checks each pixels four direct neighbours and erases the pixel if any of them are black:

```
memcpy(temp_image, output_image, sizeof(temp_image));
for (int x = 1; x < BMP_WIDTH - 1; x++) {
    // we go from 1 to width-1 to avoid borders
    for (int y = 1; y < BMP_HEIGTH - 1; y++) {
        if (temp_image[x][y][2] == 255) {
            //if the center is white we continue
            if (temp_image[x - 1][y][2] == 255
                && temp_image[x + 1][y][2] == 255
                && temp_image[x][y - 1][2] == 255
                && temp_image[x][y + 1][2] == 255

                ) {
                // keep white (do nothing to output)
            } else {
                memset(output_image[x][y], 0, BMP_CHANNELS);
```

The outermost borders are set to black (code not shown) in order to allow our exclusion zone to rest alongside the border without half-cells causing issues. Each erasion pass has been timed which we will talk more about later.

After each erosion detect_spots is called which scans the picture for white pixels. If one is found it checks whether the exclusion zone is free of white pixels (cells), if so, the pixels coordinates are recorded, the whole detection zone (12x12) is exterminated, and the detection zone moves to the next pixel. Coordinates are stored in two fixed size arrays with overflow detection.

Lastly generate output image copies the original RGB image to the output and overlays red crosses on top of detected cell coordinates

```
/*Kopier originalt billede*/
memcpy(output_image, input_image, sizeof(unsigned char) * BMP_WIDTH * BMP_HEIGTH * BMP_CHANNELS);
```

Within the context of embedded systems, limited memory is quite an issue. We tried to keep this in mind and therefore implemented memory allocation and temporary buffers in our morphological_closing function to store temporary pixel values.

```
unsigned char *temp = malloc(BMP_WIDTH * BMP_HEIGTH);
if (temp == NULL) {
    fprintf(stderr, "Memory allocation for morphological closing failed\n");
    return;
}
```

We declare a pointer for temp to the allocated memory given by malloc. We then check if the memory allocation process was successful. If not, we throw a stand error.

Optimizations and enhancements

Explain here the optimizations and enhancements you have implemented in order to improve cell detection rate, execution time, memory use, and/or other algorithm characteristics you considered relevant. Explain what was the motivation (thinking-process) behind the optimizations and enhancements you implemented.

We've already covered the majority of the optimations during the design and implementation but to sum it up quick we have

1. The use of **bit manipulation for arithmetic** (r+g+b)*85 >> 8 avoids division and floating point operations, which makes a loop faster.
2. Using **memset** to set all three color channels simultaneously which improves performance and CPU time compared to per channel loop assignment.
3. Adaptive thresholding with **otsu** instead of using a fixed threshold, improves cell detection accuracy under varying light and contrast conditions.
4. Before each erosion pass the current image is duplicated with **memcpy** to ensure consistent neighbor comparisons.
    - Pro: Guarantees correct behavior and easy debugging
    - Con: Slightly higher memory use, but simple and reliable
5. **Border handling** by setting the first and last rows to black to black in each pass prevents undefined edge behavior and keeping erosion results consistent.
6. The black ring surrounding the **detection window** ensures that two adjacent cells are not counted as one. This reduces false positives when cells are clustered.
7. Each major step (Erosion and detection) is wrapped with **clock()** timers to measure runtime and identity performance bottlenecks, which helps evaluate efficiency improvements and verify erosion and thresholding are the dominant stages.
8. We also implement morphological closing to fix the issue of high threshold causing holes in cells.
9. Our last optimization was a new algorithm, distance transform. This algorithm creates a distance map of distance of white cells to black cells. From there it finds the local maxima of the distance map. These maxima correspond to individual cells. We originally got the idea from asking chatgpt, and again used gpt to understand the logic behind it.

Test and analysis

Report here the results from the test and analysis you have carried out according to the assignment instructions. You need to at least address the following: functionality tests, execution time analysis, memory use analysis.

For each optimization/enhancement you implement, you need to perform tests to prove its validity. If you have implemented optimization/enhancements which do not give the expected benefits, describe why it does not work.

Remember to discuss the results from the test and analysis you have carried out, do not just present them, but explain and argue their meaning.

Functional tests

During the development of our embedded system a different variety of tests were performed. With functions like Grayscal conversion a simple visual inspection could verify the color intensity preserved brightness with no unwanted noise. The use of integer arithmetic provided results visually identical to a standard floating point average.

The otsu method was compared to a manually selected fixed threshold. Otsu's adaptive approach consistently provided a threshold between 100-120 depending on the image, matching our expectations from when we were juggling around different values manually ourselves to find the best fit. This confirmed that the histogram based calculation was functioning correctly.

The binary image and erosion correctly isolated cell regions from the background. The erosion step gradually reduced cell size while removing small noise clusters as intended. After several iterations no remaining white pixels were present except the eroded cell centers.

The exclusion ring logic successfully prevented multiple detections of the same cell in clusters. Detected coordinates matched the visible cell centers when overlaid with a red cross in the output image. No false positives were observed at image borders due to explicit black border handling.

And the generated output image correctly overlaid crosses on top of the detected cell centers confirming coordinate translation between detection and visualisation. Overall the full pipeline behaved as expected with each stage producing interpretable results and the final output clearly indicating all detected cells.

Execution time analysis

Execution time was measured using standard clock() function around the two computationally intensive routines (Basic erosion and spot detection).
Each following iteration represents one complete erosion followed by a detection step.

Iteration	Detecting Spots (s)	Basic Erosion (s)
1	0.001926	0.003799
2	0.001905	0.003396
3	0.001876	0.003482
4	0.001936	0.003523

5	0.001611	0.002879
6	0.001333	0.002459
7	0.001249	0.002413
8	0.001361	0.002635
9	0.001131	0.002333
10	0.001045	0.002173
11	0.001094	0.002095
12	0.001049	0.002316
13	0.001081	0.002236
14	0.000983	0.002111

Tabel 1 Time analysis on each iteration for detecting spots and erosion

Summed time for all iterations:

- Detecting spots: 0.01768s
- Basic erosion: 0.03975s
  Total: 0.05743s

The measured total runtime with the clock function:
Time used: 0.121633s

The total measured runtime (0.121633s) is roughly twice the sum of the total iterations. This is expected since several other stages and operations are not part of the iteration timing like File I/O operations, Grayscale conversion, memory copying, console printing and final output rendering. These contribute the remaining 0.06 seconds bringing full runtime to 0.12s.

Performance improvements

Initially before optimization, the total execution time was approx 0.2 seconds for the same image. Through a sequence of targeted improvements runtime was reduced to 0.121 seconds representing a 40% speed up without sacrificing detection accuracy.

Optimization step	Approx. Runtime After Change (s)
Baseline implementation	**0.200**
Integer arithmetic	**0.178**
Use of memset in thresholding/erosion	**0.158**
Introduction of memcpy	**0.135**
Streamlined console output & loop structure	**0.126**
Minor code clean-up & constant reuse	**0.121**

It's important to know that these timing values are indicative rather than absolute since each new execution results in slightly different measured values due to variations in system load caching effects and background processes on the host machine. The times have been captured on a macbook pro m3.

The reduction demonstrates how cumulative micro optimizations even if individually small can lead to significant speed up.

Memory use analysis

Our implementation uses static memory allocation for all image buffers and coordinate arrays providing predictable and mostly constant memory usage.

Our input RGB image (the original) is 950 * 950 * 3 which equals 2.707.500 bytes. The same goes for our output image. However, our binary_image and temp_image are only 950*950 which equals 902.500 bytes. Originally we had 3 3D arrays, but we decided during our optimizations to go for 2 3D and 2 2D arrays. We went from 8122500 bytes to 7220000, a difference of 902500 bytes or 11.11% saved memory.. Furthermore we have a saved coordinate array with the all the detected cells and its coordinates which have been set to max 310 cells with 2 ints (x,y) and each int costs 4 bytes which equals 310 * 2 * 4 = 2.480 bytes. This gives us a total of around 7.22mb of fixed memory usage.

Using statically allocated arrays eliminates the need for dynamic memory (malloc) ensuring constant memory consumption. Although the temporary erosion buffer costs more image memory it was essential to ensure correctness when reading and writing to the same image data during each iteration. A similar situation occurred in the morphological closing function, where we had to use a temp array. Our solution to this was to implement dynamic memory since the array would only be used to perform one set of calculations. Since the memory is freed, we don't include it in our total. The total 7.22mb is well within low end computers capacity.

Detection accuracy and precision

To assess the accuracy of the cell detection algorithm the average for each difficulty has been analysed taken from all the tests provided in the project.

Difficulty	Average Detected Cells	Theoretical (Expected)	Deviation (%)
Easy	302.6	300	**+0.87%**
Medium	285.7	300	**−4.8%**
Hard	282.2	300	**−5.9%**
Impossible	272.0	300	**−9.3%**

Accuracy improvements

During development several refinements were introduced to improve detection rate. We've made a table below showing the improvements on the medium difficulty. Duly note that distance transform & local Maxima was an idea given by generative AI however written by us.

Version / Change Introduced	Description	Average Detected Cells (Medium)
Baseline	Fixed threshold = 90 (no Otsu, no exclusion ring)	231
Border Handling Fix	Blackened border-frame prevents false positives along edges	252

Otsu's Adaptive Threshold	Automatically selects threshold from histogram	280
Morphological closing	Filled all cells with white to stop erosion caused by holes due to high threshold. *Note* Lower cell count due to no cells being counted more than once.	257
Distance Transform, Local Maxima & edge detections	Distance transform is a more advanced algorithm for finding cells. It calculates how far each white pixel is from the nearest black pixel. This creates a 2D array, where the cell centers have the highest value. This algorithm doesn't erode like basic erosion.	285.7

Many other steps have been taken however not anything that made enough of a difference to necessarily show or document down. The results indicate that the algorithm maintains near perfect precision for simple images and remains somewhat consistent for more complex images (using Distance transform for medium to impossible. As image difficulty increases, the loss in accuracy follows the expected trend due to overlapping regions and erosion side effects.

References

- "Otsu's Method." www.youtube.com, www.youtube.com/watch?v=jUUkMaNuHP8.
- Jian Wei Tay. "Morphological Opening and Closing." YouTube, 6 Oct. 2020, www.youtube.com/watch?v=E_vU1Wd7Ks8. Accessed 4 Oct. 2025.
- Jian Wei Tay. "The Distance Transform." YouTube, 11 Oct. 2020, www.youtube.com/watch?v=oxWfLTQoC5A. Accessed 4 Oct. 2025.
- Geeks for geeks. "memcpy() in C" https://www.geeksforgeeks.org/cpp/memcpy-in-cc/
- Geeks for geeks "memset() in C with examples" https://www.geeksforgeeks.org/c/memset-c-example/