Coloreduce

PELLERIN Guillaume 296935

December 2018

1 Analysis

The goal of Coloreduce is to reduce the number of colors in a picture, depending on the user inputs. To solve this, I begun by subdividing the problem into 4 parts.

First, we need to get all the inputs. I first thought about using ofstream and ifstream to parse data, char by char, which I implemented in C++. I tried to create general functions to reuse them as much as possible, to get data, remove spaces and put it in arrays. I realized after having my program working, that I needed to use cin, so I modified this at the end. So now it works as follow: For each data, we store it in a variable, check if it respect what we want and then cout it if it is needed for the picture. For example, nbF and nbR are stored as simple integers, while colorsUsed and colorsThreshold are stored in C-style arrays of Pointers. To do this, we use "cin" and check if the value respect what we want. For example, if nbR is not between 2 and 255, we call the function error_nbR with nbR as argument and then return(1) from main to stop the program. To store the two arrays of pointers, we allocate memory with "new" for integers or pointers to Array of integers. All of these pointers are deleted with deletePointers function at the end of the program.

After this, there is the thresholding, which process the image to reduce the number of color. In order to lower complexity, I merged thresholding and parsing the picture. We pass the empty picture to the function which compute for each pixel $\sqrt{R^2 + G^2 + B^2}/(\sqrt{3} * MAX)$ and set the appropriate color in a 2D map. This map is an Array of pointers of Array of short defined in the main function.

Then, we filter the picture, depending on how much time the user wants. To do this, we first create an array for each possible color, initialized at 0, and an empty temporary array which will have the new values of the map. Then, for each pixel of the map, except the one on the border, we check how much time we see on the 8 closer pixels the different colors. If the max of the Array of occurrences is equal or more than 6, the current pixel get this color, else it get black. Then we reset the table. At the end, we copy the temporary array the the map.
When every filtering are done, if it has been done more than once, we had black at the border of the picture.

At the end, we just send back the picture, using the map and the colorTable to cout R G B of each pixel.

2 Pseudo-Code of filtering

```
input  : Integer xSize
         Integer ySize
         Short nbR
         Array of array of short map[xSize][ySize]
Integer maxColorNb, maxColorValue = 0
Array of integers testColor[nbR + 1] = 0
for xPos ← 1 to xSize do
    for yPos ← 1 to ySize do
        for i ← 0 to nbR do
            testColor[i] = 0
        end
        for i ← 1 to 1 do
            for j ← 1 to 1 do
                if not (i == 0 and j == 0) then
                    testColor[map[i + xPos][j + yPos]] += 1
                end
            end
        end
        maxColorNb, maxColorValue = 0
        for i ← 0 to nbR do
            if testColor[i] > maxColorNb then
                maxColorNb = testColor[i]
                maxColorValue = i
            end
        end
        if maxColorNb >= 6 then
            tmpMap[xPos][yPos] = maxColorValue
        end
        else tmpMap[xPos][yPos] = 0
    end
end
for xPos ← 1 to xSize do
    for yPos ← 1 to ySize do
        State map[i][j] = tmpMap[i][j]
    end
end
```

3 Complexity of filtering

The complexity is in $o(xSize*ySize*nbR)$. Indeed, the function can be divided in two part. The first one, to create the new map, will be done $xSize*ySize$ time. In it, we have the assignment of testColor which is done each time nbR time, the counting taking 9 instruction, the searching for max value taking 2nbR instructions and the final test which is constant. With this, we have $xSize * ySize(3nbR + 10)$ instructions. The second one to write back the new map is only in $o(xSize * ySize)$. It is why the complexity is $o(xSize * ySize * nbR)$.