Graded exercise: A Qt-based image viewer

Documentation : ImageProcessor

The following document is a documentation of a how the graded exercise "A QT-based Image Viewer" of the pratical c++ course in SoSe 2016 was solved. It contains the basic structure of the developed program, the used classes and algorithms as well an explanation of the develeped algorithm for "detecting detecting dark objects within noise."

Andre Bode - 358254

Inhaltsverzeichnis

1 Part 1 : Loading, Saving, Analyzing and Modifying Images

The following chapter descripes the design of the system-structure with its classes and the concepts of the used algorithms.

1.1 System - Structure

The system is divided into three main parts :

1. the **G**raphical **U**ser **I**nterface
2. the Image-Class
3. the SignalProcessor-Class

All three parts together make up the program "ImageProcessor". In the following sections the three main parts are described more precisely.

1.1.1 Graphical User Interface

The graphical user interface, which one can see in figure 9 is divide into four main parts.

Abbildung 1: the graphical user interface

The first part in the very left is the main menu, from here one can load an save images, and open the settings of different analyzing, filtering and modifying algorithms. E.g. by pushing the "Smooth"-Button within the main-menu the settings for smoothing in the very right upper corner appear. In the center of the graphical user interface one can the second part see two images, with some controls between them. The upper image is the image which is used for

all analyzing, filtering, modifying or simply said for every processing the program can do on images. In the lower image one can see the a copy of the upper image which is shown there for

1. having a reference how the image looked before processing on it.
2. making it easy to load the original back to the upper image, without the often annoying file-dialog.

Between the two images one can see a progress bar, which shows the progress within CPU-intensive processing on the images. Next to the progress bar one can see three magnifiers. The first two are for zooming into and out of the image. If you press the third magnifiers the image is stretched to the size which is currently provided by the GUI. If the image is stretched the zooming is not allowed. Next two the magnifier-buttons one can see three arrow buttons. The first is for copying the upper image into the lower. The second is for copying the lower image into the upper. And the is for swapping the images.
The already mentioned settings are displayed in the third part, the upper right corner of the GUI.
The task of analyzing the images and auto-detecting dark objects within the noisy images produce results. These results are displayed within the fourth part, which can be found in right lower corner of the graphical user interface. The source-code of the graphical user face can be found within the files :

- main.cpp
- mainwindow.h
- mainwindow.cpp

The file main.cpp is kept as short as possible.
It only starts the graphical user interface and has no other use in this project. The files mainwindow.h and mainwindow.cpp do not contain any complex algorithms or any complex code. Their only purpose is to control the graphical user interface by making controls visible, unvisible, enable or disable them or to start methods from the Image-class.

1.1.2 Image - Class

The Image-Class is a wrapper class which firstly is an interface between the graphical user interface and the QImage-Objects (in which the images are stored internally), secondly it is an interface between which provides methods for the graphical user interface to easily execute analyzation and modification algorithms from the SignalProcessor-class and thirdly it is an interface for transform the data-representation within the QImage-class into the vector-representation within the SignalProcessor-class and as well the same way back form SignalProcessor to QImage-class.

1.1.2.1 GUI <–> QImage

Within the graphical user interface the images are encapsulated in QLabel-objects which provide methods for displaying, zooming, scrolling and related purposes.
The QImage-class provides methods for loading, saving and manipulating images.
In order to connect the previous described graphical user interface the Image-Class provides functions such as

- loadImage()

- saveImage()
- convertToGrayScale()
- stretchImageToLabel()
- scaleImage()

The image class is an interface between the graphical user interface which provides buttons and other controls to start and parametrize algorithms, which process on the image.

1.1.2.2 GUI <–> SignalProcessor

Two avoid complex code within the graphical user interface the Image-class provide some methods such as

- smooth()
  Smooth() reduces noise within the image, using a two-dimensional sliding window, which iterates through all pixels and sets their colour to the average-color within the sliding window.
- gradient()
  The gradient-method derives the image, it is possible to derive horizontally, vertically, and to calculate magnitude- and direction-derivatives.
- invert()
  Inverts all colors of the image. This is often useful for displaying derivatives.
- threshold()
  The calculation of the threshold can be done either in "normal" mode or in "binary" mode. Within the binary mode all values which are smaller one value and all values which are bigger are set to another specific value. In the normal mode only the the values, which are smaller than the threshold are set to a new specific value and the values which are bigger are not touched. This makes it possible to stepwise extract the colors one wants to keep.
- staticIncrease()
  StaticIncrease() statically increases or decreases all values by a given number. Simply said this is an addition.
- getMinMax()
  The getMinMax-method provides information about either the minimum or maximum color value within the image.
- getAverage()
  The getAverage-method calculates the average-color-value within the image.
- getStandardDeviation()
  This method provides the information of the standard deviation of the color-values within the image.
- calcHistogram()
  The calcHistogram-methods creates a tab-seperated text-file which contains frequencies of every color within the image.

These functions have the purpose to provide a not complex and easy to use interface for the graphical user interface to use the the analyzation and modification-methods of the SignalProcessor-class.

1.1.2.3 QImage <–> SignalProcessor

To load color-data from the QImage to the SignalProcessor-class and the same way back the functions

- qImageToSignalProcessors()
- signalProcessorsToQImage()

were developed.
Inside the QImage object the color-data is stored in so called "scanlines". Every scanline describes a horizontal line of pixels.
The function qImageToSignalProcessors iterates all scanlines and loads every value into the one-dimensional integer vector inside the SignalProcessor-class. To avoid failures because of the different ranges of integer-values and valid image values (0 to 255) the function SignalProcessor::cutOffToRange provides the possibility to cut every calculated value within a modification-algorithm back to a given range. This range can be set within the constructor of SignalProcessor or by using the functions

- SignalProcessor::setMinMaxValue()
- SignalProcessor::getMinValue()
- SignalProcessor::getMaxValue
- SignalProcessor::setUseCutOffToRange()
- SignalProcessor::getUseCutOffToRange()

To load data back from the SignalProcessor-class to QImage the function signalProcessorsToQImage was developed.
It iterates all values of the one-dimensinal integer vector within the SignalProcessor-Class and associates them back to their position within the scanlines of QImage.

As there are some task in this graded exercise which need the image to get interpret as a two-dimensional data-structure, the the functions qImageToSignalProcessors() and signalProcessorsToQImage() furthermore provide the possibility to load the colors within QImage into a vector of SignalProcessors and back from this vector to QImage. These vectors can either contain the horizontal lines of pixels or the vertical columns of the pixels. For loading the colors into a vector of SignalProcessors and for determining whether the vector contains color-lines or color-columns one can use some additional parameters.

1.1.3 SignalProcessor - Class

The SignalProcessor class very abstractly describes a signal of integer values. This has the purpose to make the class usable for many different data-processing applications.
The class provides basic mathematical operators as well as standard-constructor, copy-constructor and equality, non-equality and assignment-operators for making the class canonical.
Furthermore the class provides algorithms to retrieving information about the

- minimum value
- maximum value
- average of all values
- standard-deviation of all values

In addition to this it implements basci data-processing algorithms like

- add and subtract scalar values and other signals
- multiply with scalar values or other signals
- divide by scalar values or other signals
- threshold and binary threshold
- moving-average - noise filtering
- calculting normal and absolute gradients
- invert values by subtracting them from the maximal possible value

To avoid boiler-plate-code the analyzation and modification algorithms are all encapsulated into the two functions

- analyzeSignalProcessor() and
- modifySignalProcessor

The type of analyzation or modification ist defined by the parameters. To keep a simple parameter list, which fits well for every modification-purpose a vector of integer values is used. If only one parameter is used, it is stored within index 0 of the vector, if more parameters are used they are stored at the index 1 to the number of parameters.
If a whole signal is used the values of the second signal are stored within the vector.
In addition to analyzing the signal, the SignalProcessor-class offers the possibility to calculate a frequency distribution, as usually visualized in histograms and output this frequency distribution into a text file.

2 Part 2 : estimate image size

The development of the estimation-algorithm was mainly based in using the in Part 1 developed functions and testing which of them offer the best possibilities to extract dark regions from noisy regions within an image.
A first approach was to detect edges within the image. After some unsuccessfull tries with a combination of different derivatives and smoothing it turned out that the standard-deviation, which in all regions of the image is nearly the same makes it really hard to detect edges. And even though after detecting an edge there is still the problem that has to count the number of pixel between the detected edges. What sounds very simple for only two edges within a line of colors can be really tricky for when there were several edges detected, becaus one never knows whether the edge belongs to the region which is to detect or not.

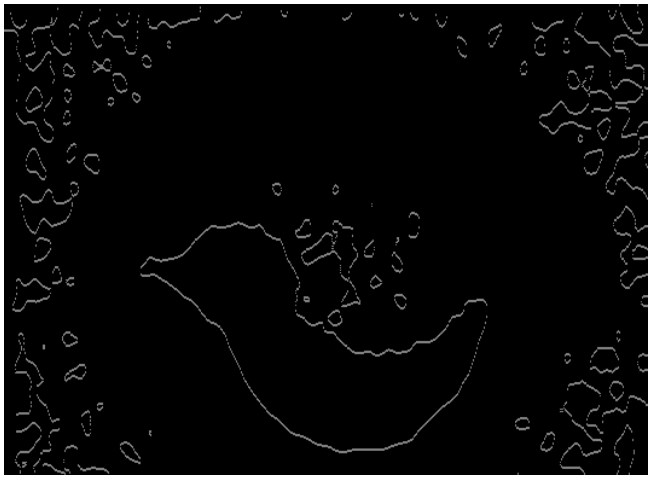

Abbildung 2: bad results for detecting edges

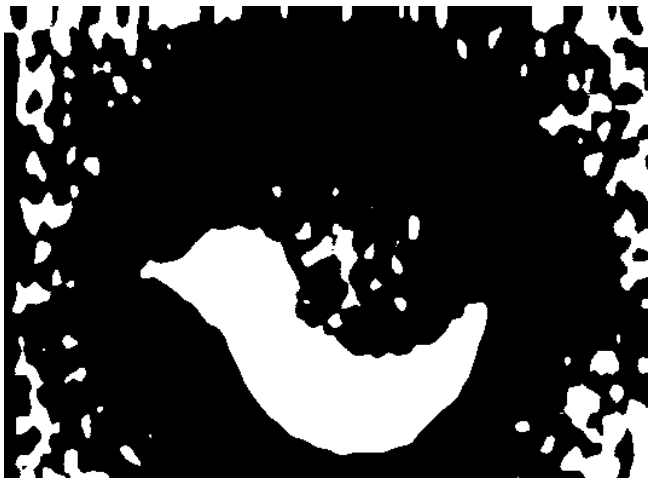

Abbildung 3: bad results for detecting edges

Recording to this, the research into edge detection was stopped and the picture itself and parts of it wer analyzed to figure out common properties.

For this the several regions were determined :
Region 0 : noise

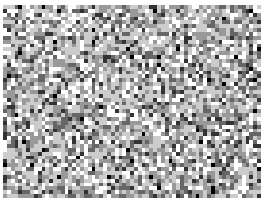

Abbildung 4: noisy region

Region 1 : dark noise

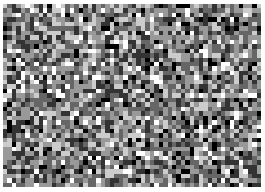

Abbildung 5: dark noisy region

Region 2 : duck

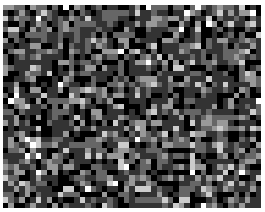

Abbildung 6: duck

Region 3 : a border of the duck

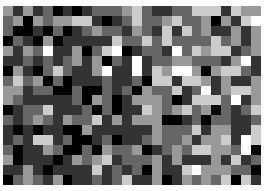

Abbildung 7: a border of the duck

By analyzing these four regions it was noticed that minimum, maximum values were everywhere exactly the same. The standard-deviation was only a little diffent in the regions of the duck, but very common with the standard-deviations of the regions of dark noise. But surprisingly the average values were much more different than the other properties.

The new aim was now to detect the dark zones by its average values.
To increase the differences between the average-values of the images several filters were tested. It turned out that mainly the smooth filter supported increasing of the differences of the average values.
After this research a very simple algorithm was developed that simply executed the smooth algorithm with decreasing the size of the sliding window every fourth step by factor 0.5 for three times. The best initial size of the sliding window was discovered at a tenth of the image-sizes. Bigger sliding windows blurred the colors to much and produced really bad results. To small sliding windows did not increased the differences of the average-values enough. The result was, that to many pixels were detected.

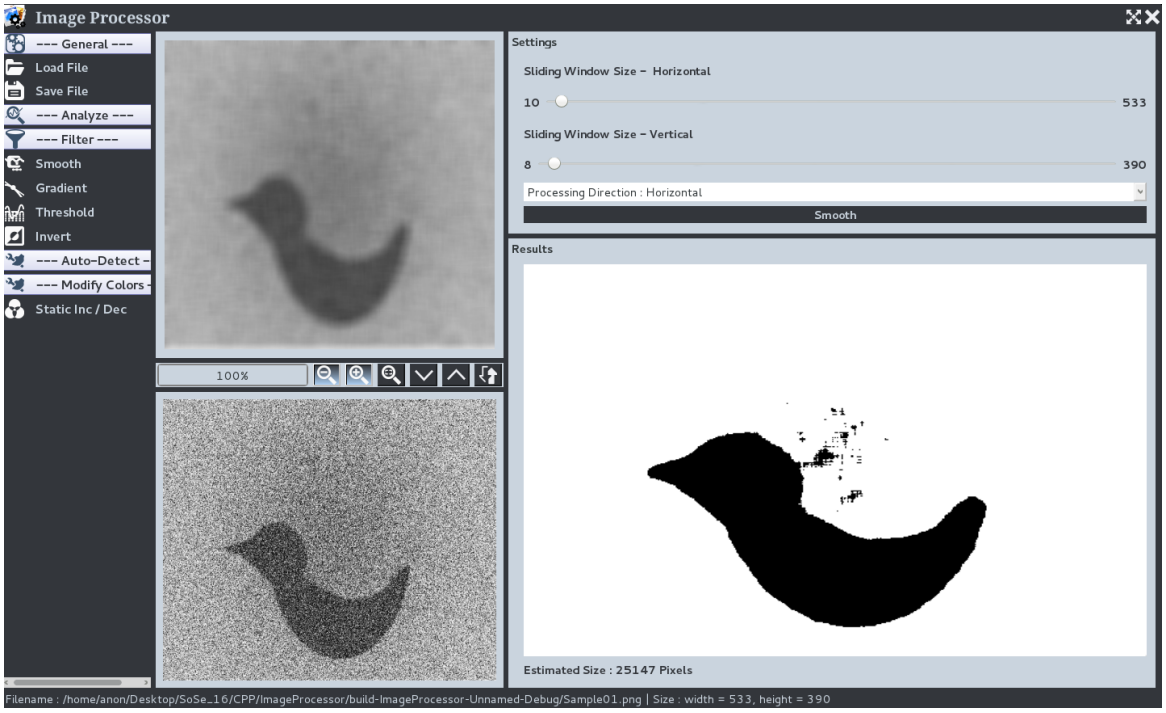

Abbildung 8: a detection after a little number of smooth iteraions with big sliding window

After stepwisely reducing the noise of the image and increasing the difference between the average-values it was discovered that a simple dynamic threshold algorithm which turned every value which was smaller than the the average of the image - minus the standard deviation of the image was turned to black and was counted as "detected pixel" and every value, which was bigger than the average minus the standard deviation was turned to white and ignored as "not detected pixel".

Abbildung 9: a detection after a many number of smooth iterations with first a big and then a small sliding window