LIBRARY FOR ORGANIZATION OF IMAGE RECOGNITION SYSTEMS

BY

OLEKSANDR SHYROKOV

BS in ECE, Odessa State Polytechnic University, 2000

THESIS

Submitted to the University of New Hampshire

in Partial Fulfillment of

the Requirements of the Degree of

Master of Science

in

Electrical Engineering

September, 2002

This thesis has been examined and approved.

Thesis director, Richard A. Messner, Ph.D.
Associate Professor of Electrical and Computer Engineering

L. Gordon Kraft, III, Ph.D.
Professor of Electrical and Computer Engineering

W. Thomas Miller, III, Ph.D.
Professor of Electrical and Computer Engineering

Andrew L. Kun, Ph. D.
Assistant Professor of Electrical and Computer Engineering

Date

DEDICATION

I would like to dedicate this thesis to my mother Elvira and sister Alina for their love and support.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr. Richard Messner, for his time and effort and for making the resources of the Synthetic Vision and Pattern Analysis Laboratory (SVPAL) available for my research.

Many thanks to my academic advisor, Dr. Andrzej Rucinski, for the support and help.

A special thanks to Dr. Russell Carr of the University of New Hampshire for the use of his microscope in the process of image capture. And Dr. Elise Sullivan from Microbiology Department for cooperation and help provided for the development of the test applications.

I would also like to thank Dr. Kun, Dr. Miller and Dr. Kraft for serving on my defense committee and reviewing my thesis.

Much appreciation goes to all my friends who supported me during the development of my thesis.

TABLE OF CONTENTS

DEDICATION ..... **iii**

ACKNOWLEDGEMENTS ..... **iv**

LIST OF FIGURES ..... **viii**

ABSTRACT ..... **ix**

CHAPTER 1 PROBLEM STATEMENT **1**

1.1 WHAT IS A RECOGNITION SYSTEM? 1
1.2 WHO NEEDS RECOGNITION SYSTEMS? 2
1.3 WHAT IS CURRENTLY DONE? 3
1.4 PRICE INFORMATION 4
1.5 WHAT CAN BE DONE? 5

CHAPTER 2 APPROACH **7**

2.1 STARTING POINT 7
2.2 PLUG-INS 8
2.2 ACQUSITION 9
2.3 SEGMENTATION 10
2.4 DISPLAYING 10
2.5 TRANSFORMATIONS 11
2.6 INTERFACING 11
2.6.1 Acquisition 12
2.6.2 Segmentation 13
2.6.3 Features 13
2.6.4 Classification 13
2.6.5 Displaying 13
2.7 MODES OF OPERATION 14
2.8 ABSTRACTING 14

CHAPTER 3 IMPLEMENTATION EXAMPLE **16**

3.1 MAIN APPLICATION 16
3.2 SYSTEM ORGANIZATION 17
3.3 NORMAL AND RESEARCH MODES 19
3.4 AMORS USER INTERFACE 19
3.5 GENERAL PURPOSE PLUG-INS 23

CHAPTER 4 AUTOMATIC HUMAN BRAIN CELL RECOGNITION **24**

4.1 PROBLEM STATEMENT 24
4.2 IMAGE SEGMENTATION 26
4.3 POLAR TRANSFORM 28
4.4 FEATURE PLUG-INS 29
4.5 FUZZY LOGIC 30
4.6 PERFORMANCE 31
4.7 3D ACQUISITION 32
4.8 SOM DATABASE 33
4.8.1 Self-organized maps 34
4.8.2 Learning 34

4.8.3 Example 36
4.8.4 Implementation 39
4.8.5 Results 40

CHAPTER 5 COUNTING OF BACTERIA 43

5.1 PROBLEM STATEMENT 43
5.2 PREVIOUS WORK 44
5.3 ACQUISITION AND DISPLAYING 44
5.4 METHODS 45
5.5 RESULTS 46
5.5.1 Experiment 1 47
5.5.2 Experiment 2 48
5.5.3 Experiment 3 48
5.5.4 Experiment 4 49
5.6 CONCLUSIONS 49

CHAPTER 6 DISCUSSION 51

APPENDIX A INTERFACE LIBRARY DESCRIPTION 54

A.1 INTERFACES.LIB LIBRARY 54
A.2 INTERFACE OBJECTS AND FUNCTIONS 54
A.3 CLASSES DESCRIPTION 55
A.3.1 CSJBuffer 55
A.3.2 CSJRGBAColor 60
A.3.3 CSJImage 62
A.3.4 CSJImageList 67
A.3.5 CSJRect 68
A.3.6 CSJObjectInfo 70
A.3.7 CSJObjectInfoList 72
A.3.8 CSJSpecimen 73
A.3.9 CSJIniFile 77

APPENDIX B INTERFACE FUNCTION DESCRIPTION 81

B.1 NORMAL MODE 81
B.2 RESEARCH MODE 82
B.3 PROCESSING ROUTINES 83
B.4 UTILITY ROUTINES 84

APPENDIX C PLUG-IN INTERFACING 86

C.1 COMMON FUNCTIONS FOR ALL PLUG-INS 86
C.2 ACQUISITION PLUG-IN 87
C.3 DISPLAY PLUG-IN 88
C.4 OBJECT SEPARATION PLUG-IN 89
C.5 FEATURE EXTRACTION PLUG-IN 89
C.6 DATA BASE PLUG-IN 90

APPENDIX D DESIGNED PLUG-IN DESCRIPTION 91

D.1 DUMMYACQ PLUG-IN 91
D.2 FROMFILEACQ PLUG-IN 92
D.3 SIMPLEDISPLAY PLUG-IN 92
D.4 AREA AND CENTER PLUG-INS 97
D.5 MINAREA AND MAXAREA PLUG-INS 97
D.6 INIDB PLUG-IN 98
D.7 SOMDB PLUG-IN 101
D.7.1 Parameters 101

D.7.2 Format of input/output files 102
D.7.3 Purpose of each program 103
D.7.4 Setup dialog box for SOMDB 104
D.8 BLOB SEPARATION 105
D.9 SIMPLE REJECTER PLUG-IN 105
D.10 MORPHOLOGY OP PLUG-IN 105
D.11 AREADIVISION PLUG-IN 106
D.12 POLAR PLUG-IN 107
D.13 SAVE LOADOBJ PLUG-IN 107
LIST OF REFERENCES 108

LIST OF FIGURES

FIGURE 1.1 EXAMPLE OF THE BLOCK DIAGRAM OF THE SYSTEM ORGANIZATION 6
FIGURE 2.1 DIAGRAM OF RECOGNITION PROCESS FOR HUMAN BRAIN CELL RECOGNITION 8
FIGURE 2.2 MODULE CONNECTION 12
FIGURE 3.1 MAIN APPLICATION ORGANIZATION 17
FIGURE 3.2 BLOCK DIAGRAM OF PLUG-IN CONNECTIONS 18
FIGURE 3.3 BLOCK DIAGRAM FOR NORMAL MODE 19
FIGURE 3.4 BLOCK DIAGRAM FOR RESEARCH MODE 19
FIGURE 3.5 MAIN WINDOW OF AMORS 20
FIGURE 3.6 MAIN WINDOW DURING THE EXECUTION 21
FIGURE 3.7 CONFIGURATION DIALOG BOX 21
FIGURE 4.1 EXAMPLES OF CELLS (A) - MICROGILA AND ENDOTHELIAL; (B) - NEURON, ASTROCYTE AND OLIGODENDROCYTE 25
FIGURE 4.2 (A) ORIGINAL GRAYSCALE IMAGE; (B) TRANSFORMED IMAGE 27
FIGURE 4.3 REGION-GROWING ALGORITHM 28
FIGURE 4.4. POLAR TRANSFORM MAPPING DIAGRAM 29
FIGURE 4.5 THE CONFIGURATION OF THE APPLICATION FOR AMORS. 32
FIGURE 4.6 ILLUSTRATION OF THE MAXIMUM DISTANCE 36
FIGURE 4.7 THE DISTRIBUTION OF INPUT SAMPLES 36
FIGURE 4.8 (A) MAP WITH THE SIZE 5X4 (B) RANDOMLY INITIALIZED MAP IN THE SAMPLE SPACE 37
FIGURE 4.9 THE MAP AFTER TRAINING 38
FIGURE 4.10 REPRESENTATION OF THE MAP 38
FIGURE 4.11 LABELED MAP REPRESENTATION 39
FIGURE 4.12 TRAINING PARAMETERS FOR THE SOM 40
FIGURE 4.13 SOM WITH SIZE 12 BY 8 USING 5 FEATURES 40
FIGURE 4.14 SOM WITH SIZE 20 BY 16 USING 5 FEATURES 41
FIGURE 4.15 SOM WITH SIZE 12 BY 8 USING AREA, DISTINCTNESS AND ENTROPY AS FEATURES 42
FIGURE 4.16 SOM WITH SIZE 12 BY 8 USING AREA, ROUNDNESS AND MAXIMUM RADIUS AS FEATURES 42
FIGURE 5.1 INDUSTRIAL EXAMPLE 44
FIGURE 5.2 PURE CULTURE (INVERTED) 44
FIGURE 5.3 DIFFERENT KIND OF BACTERIA IN THE PURE CULTURE (INVERTED) 45
FIGURE 5.4 THE CONFIGURATION OF THE PLUG-INS FOR AMORS TO COUNT BACTERIA 46
FIGURE 5.5 IMAGE TAKEN FROM MICROSCOPE (INVERTED) 47
FIGURE 5.6 INPUT IMAGE (INVERTED) 47
FIGURE 5.7 RESULT (INVERTED) 47
FIGURE 5.8 IMAGE TAKEN FROM THE MICROSCOPE (INVERTED) 48
FIGURE 5.9 INPUT IMAGE (INVERTED) 48
FIGURE 5.10 RESULT (INVERTED) 48
FIGURE 5.11 IMAGE WITH APPLIED THRESHOLD (INVERTED) 48
FIGURE 5.12 INPUT IMAGE (INVERTED) 48
FIGURE 5.13 IMAGE WITH APPLIED THRESHOLD (INVERTED) 48
FIGURE 5.14 INPUT IMAGE (INVERTED) 49
FIGURE 5.15 RESULT (INVERTED) 49
FIGURE D.1 SETUP DIALOG BOX OF DUMMYACQ PLUG-IN 92
FIGURE D.2 MAIN WINDOW OF SIMPLEDISPLAY 93
FIGURE D.3 INFORMATION DIALOG BOX 95
FIGURE D.4 CHOOSE HISTOGRAM DIALOG BOX 95
FIGURE D.5 HISTOGRAM WINDOW 96
FIGURE D.6 ORIGINAL IMAGE (A) AND IMAGE WITH SELECTION APPLIED (B) 97
FIGURE D.7 MEMBERSHIP FUNCTIONS 99
FIGURE D.8 SOMDB SETUP DIALOG BOX 104
FIGURE D.9 MORPHOLOGYOP SETUP DIALOG BOX 106

ABSTRACT

LIBRARY FOR ORGANIZATION OF IMAGE RECOGNITION SYSTEMS

by

Olexander Shyrokov

University of New Hampshire, September, 2002

In many recognition problems it is possible to divide the recognition task into a collection of separate processes. This thesis develops a method that simplifies the creation of automatic recognition systems. Formal separation of different algorithm steps was performed and an interface for communicating between each step was developed. In order to evaluate the robustness of the proposed method, the algorithms developed by Mr. Tony Pawlak (M.S. UNH ECE 1998) in his thesis work on automatic human brain cell recognition were implemented. The original human brain cell recognition research was done using Matlab software [12]. After the original recognition process was ported into the proposed standard, improvements were made to demonstrate how the designed system could be extended and modified. To demonstrate how a "new" recognition task could be implemented a different recognition problem was chosen: the counting of bacteria in a pure culture. Documented examples show that the proposed method and standards simplify the organization, execution, and maintenance of the recognition procedures.

Developed in this thesis is a ready to use library for various algorithms with examples and documentation as well as a full discussion on how researchers can develop their own modules for inclusion. Complete documentation on standards and interfacing is included in appendices.

CHAPTER 1

PROBLEM STATEMENT

The following chapter states what a recognition system is, and who needs it. There is a big demand for automatic systems, to make a decision without human supervision. It is possible to create such systems and they are used in industry. It is possible to create a platform that will simplify the development of the recognition systems not only for industry purposes, but also for the research purposes. The chapter discusses how recognition systems are done and how process of their development can be simplified.

1.1 What is a Recognition System?

Merriam-Webster's Collegiate Dictionary [1] describes the noun "recognition" as "the action of recognizing; the state of being recognized; special notice or attention". The process of recognition implies the process of identification! In science and industry there has always been a need for procedures that require identification of various objects and/or actions. After identification is performed an action corresponding to the particular case should be performed. This action could be counting, recording, or providing a control signal. The simplest (and in many cases most robust) solution for such identification was to use a human, to observe the situation and make the proper decision. But there are some disadvantages, such as human error, cost of human employment, and limit of the speed with which the work can be accomplished. The next logical step is to make this process automatic (i.e., design a machine, which is able to make the decision). This raises many questions regarding how to make a machine decide the what, where, and how in the decision making chain. These questions have resulted in the steep rise of research in the field of pattern recognition. Any system that can recognize something and make some decision based upon that recognition is called a Recognition System.

1.2 Who Needs Recognition Systems?

Any field of industry or science where repetitive tasks are performed potentially could benefit from a machine designed to perform those tasks. Consider an example from industry. During the production of light bulbs, each bulb should be inspected to assure that it does not have any cracks in the glass and that the shape of the glass envelope is correct. The company that produces these light bulbs could employ people to check each bulb manually. This is not very efficient since a human can work with only a small number of bulbs per hour. Furthermore a human is subject to fatigue and could potentially make errors. To alleviate these problems the job might be designed as an automatic system with a camera and a computer. The view of a bulb is acquired through the camera, digitized and then fed to a digital computer where certain algorithms are coded and executed in order to detect whether the bulb should be rejected or not. Because of the variability and dimensionality of the problem this may not be an easy system to design and could be high in cost. The desirability of such a system would be that once it is designed and built the system could work for a long time without additional expenses. The cost savings would be from not having to employ as many humans in the loop. Of course the system still needs to be maintained by a specialist, but if such a system replaces a number of people who were doing this job manually, it is cheaper to use this system with one engineer. Most industries today, which are using large numbers of production lines, use recognition systems in their workflow. Extensive research must be done, in order to make these systems better. Experiments must be performed and data must be analyzed in order to make the proper choices in the system design.

Industrial settings are not the only place where recognition is necessary. In many areas of science, scientists must perform many repetitive recognition tasks. In my work, I have found that many science applications often require observations that are very time consuming and tedious. An automatic method for these observations would save the scientists (and graduate students) much time. A good example would be the problem in microbiology science, where the counting of bacteria is required. At the moment, this is done in the following manner in the UNH College of Life Sciences and Agriculture (COLSA): A human is in the loop, who is taking samples and counting bacteria manually while looking into the microscope. From

my conversation with people involved in this kind of research, they could spend hours and hours looking into the microscope. It is observed that this causes potential health problems (eyes, spine and so on). An example is given in Chapter 5 that shows what I did to simplify their job and how this work helps to solve these problems.

1.3 What is Currently Done?

Because of the high demand for automatic recognition systems, companies have been formed to provide both special hardware and software to address recognition problems. These companies produce special hardware and/or design systems for specific jobs. An example of such a company is Datacube, Inc. (www.datacube.com). This company is mainly targeting industrial and medical recognition problems. Other companies such as Zeiss (www.zeiss.com) provide recognition tools for simplification of scientific data acquisition. Now let us return to our example of counting bacteria for a microbiology problem and see what is typically done to automate these problems. Zeiss sells microscopes coupled to digital cameras that allow scientists to use the high-resolution display of a computer to help them make their decision. Zeiss even made a step forward providing some software tools to adjust parameters of acquired pictures (contrast, color balance and so on). But still, since the software is not capable of implementing the desired task, scientists count bacteria manually. Counting of bacteria is one of the most time-consuming jobs in the Microbiology Department. Having a tool that would do the whole process automatically (or at least partially automatic) would be a big help for microbiology researchers.

What I have just described is a specific problem for microbiology. To explore a more complex recognition problem and understand what type of algorithms are used, I refer to the thesis of Mr. Pawlak. His thesis is entitled, “Automatic human brain recognition system” [2] and deals with brain cell recognition for research on Huntington's Disease. In his research Mr. Pawlak makes significant use of the Matlab software package [12]. The data images were acquired in BMP (BitMaP) format using Datacube hardware and software. These images were converted into TIFF (Tagged Image File Format) format using a program for image converting (Adobe® Photoshop® [13] or similar). Then Matlab was utilized to write the routines necessary to perform the algorithms used in his research. Matlab is a very powerful and flexible system for analysis and experimentation but is not very user friendly for specific tasks. At present, this is the way most

research is done with Matlab. If some good method or algorithm is designed during one's research, it is almost impossible to use the code without rewriting it for the new task. This is because notations and structures are different in the original and a new task. One way to make parts of different research compatible is to use a standardized approach. This work describes a method of system organization that would help to solve this problem.

1.4 Price information

As was mentioned in the section above, there are some products that simplify and take care of some aspects of the monotonous work for scientists. However, the next question after availability of the product is the cost of the product. In many instances the final product consists of both hardware (camera, controllers, computer, acquisition boards, etc.) and software (applications, development environments, etc.). It rarely happens that the software part is absent. To perform a certain task we need a certain quality of the hardware. It is very hard to argue with this fact. If you want to count bacteria automatically you need a microscope with enough magnification to see these bacteria. The magnification should be large enough to see the details of the bacteria if you want to be able to separate two kinds of bacteria. Then the resolution of the camera is defined by the size of the details one wants to capture. As we can see, one cannot hide from expenses on the hardware part of the system. But what is it about the software part? Companies, producing special hardware, often develop specific software packages (Datacube, Zeiss), which can often cost more than the hardware.

When research is done for industrial purposes, the company interested in the outcome of that research will pay for the research and equipment. However, at universities it is not always the case that monies will be available for equipment and software that could help in various research areas. In many cases, universities will resort to doing many things manually. Even when appropriate hardware is available, software can provide some additional functionality, but it is usually more efficient to design something specifically suited to the particular research task or application. This may sound obvious for people versed in Computer Science (CS) or Electrical Engineering (EE), however people who need such help are often not from such departments. In my case, I interacted with the Microbiology Department. They were not aware that their task of counting bacteria could be easily done via computer. Once aware of such systems

and techniques, they have two choices: invite someone to design the system for them or learn on their own how to design it. Lack of the experience and time issues, causes the first approach to be preferred in most of the cases.

Once the decision to invite someone to design a recognition system is made, the question that comes up next is how much does such a system cost. Often companies will take on the task of designing a system with most of the money being spent on hardware and little being spent on the software side of things. It is then logical to have a university put systems together so that all aspects of the problem can be taken into account and keep costs down. Students are typically involved in the projects, but when a student sees what is a proper approach to the problem, he/she may realize that too much time is required to design the system, even if the methods to perform recognition are quite simple. The student must take care of image acquisition, processing and displaying. A good approach is to use a group of students, but then someone has to manage the project and students will tend to come and go with time possibly making continuity difficult. The point is that there is no simple, and common way to solve this problem at the moment.

1.5 What can be done?

A look at different types of imaging systems gives us some insight into how such systems are designed. But maybe there is a way to simplify the development of the recognition systems. What is this way? The answer is simple: STANDARDS. Standard means a predefined way of doing things, which makes it possible to connect parts from different projects seamlessly. In the case when an old standard cannot provide high enough performance, the old standard is extended or a new one is proposed.

What is ultimately needed is a system, which is: inexpensive, flexible, and extendable with reusable components. In this thesis I am proposing a methodology of organization for recognition systems that simplifies the design, implementation, and execution of various recognition research tasks. The development of a philosophy for the standardization of a "plug and play" library is done which allows for easy implementation and extension to existing systems.

The proposed approach separates the recognition task into five major steps: Acquisition, Object Separation, Feature Extraction, Classification and Displaying as shown in Figure 1.1

The main source for acquisition is the camera (or cameras), but it could be any device presenting two-dimensional data. During acquisition the data is introduced into the system. This acquisition can be a one-image or a continuous sequence of video frames. Regions of interest are detected using an Object Separation feature. Individual features of these regions are then extracted by Feature Extraction functions. Under certain conditions subdivision into more regions of interest is necessary and would be performed using Feature Extraction functions. Once regions and their features have been defined, classification can be performed in the Classification step. The final result is shown during the Displaying step.

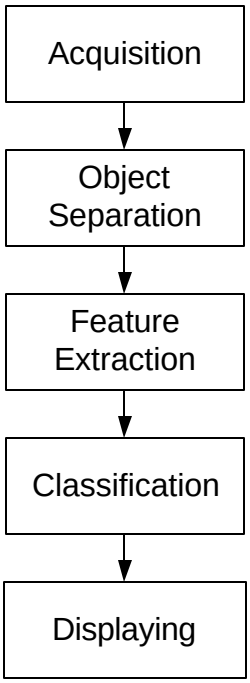


Figure 1.1 Example of the block diagram of the system organization.

The following Chapter describes the approach used to develop the proposed standard. The problems encountered and the trade-offs considered are discussed. Chapter 3 provides an overview of an example implemented specifically for this thesis. Chapter 4 shows the research performed by Mr. Pawlak [2] implemented using the proposed standard. In Chapter 5 a specific research problem of counting bacteria is developed and shown. Finally, specifications of the proposed standard are described in Chapter 6.

CHAPTER 2

APPROACH

Described in this chapter are common steps required for many recognition tasks including the way this separation into tasks is found. Each step is described indicating how it is connected with the rest of the system. Interfacing between the steps is discussed.

2.1 Starting point

The work of Mr. Pawlak [2] was a starting point for this thesis. Doctors performing research on Huntington's disease, study slices of brain's tissue to find out how many cells of a particular kind are in this tissue. To date this is done manually (see Chapter 4 for additional details). Mr. Pawlak developed a method for automating this process by computer. He acquired images of the samples and implemented his methods in Matlab. These methods were designed to recognize a certain number of cells and then indicate their type and location in the image. Doctors checked the results and they found accuracy of the methods sufficient to be helpful in their work.

This thesis began as an implementation of Mr. Pawlak's research, adding a user friendly Graphical User Interface (GUI) and some improved methods of the recognition in order to provide the doctors with a useful tool for their research in Huntington's disease. Figure 2.1 shows how Mr. Pawlak organized the recognition process. It shows blocks, which are independently functional as stand-alone units.

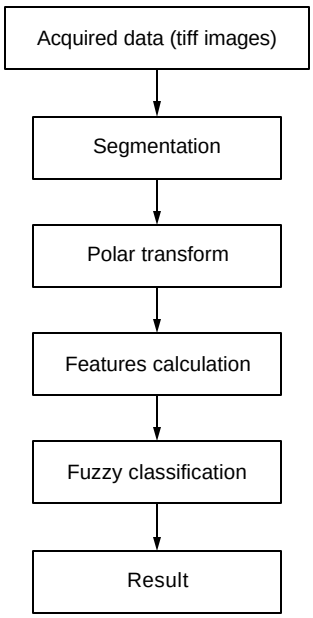


Figure 2.1 Diagram of recognition process for human brain cell recognition

It was clear that this kind of application would help doctors to perform research on Huntington's disease, but also could be used for other similar recognition tasks. Doctors working on other disease need to recognize other types of cells. This often means that some new methods could be required to recognize the new types of cells. In order to make this adaptation simple, the designed system should be easily extendable, in the sense that one can have additional methods along with old ones.

2.2 Plug-Ins

One way to design an extendable system is to use the plug-in concept. A plug-in is a module that encapsulates certain functionality (see section 3.1 for details). A user indicates to a main program of the system what plug-ins he/she wants to use. This method allows new functionality to be added into the program without rebuilding the program itself. The program uses plug-ins by means of a predefined interface that was developed for this thesis. The main program loads a required plug-in into memory and finds the location of the required functions. These functions are then called according to their order of execution and purpose. When the program finishes using the plug-in, plug-in is unloaded from the memory. It was decided to use the plug-in concept so different methods could be implemented and used when they are required without changing the main program. This requires that an interface must be developed. The

interface includes the definition of the function types with the required parameters. The plug-ins require information from the main program and this information are passed as arguments to the function.

Often times change of the classification method is required, in order to classify a new cell. Classification is incorporated into the "Fuzzy classification" block (Figure 2.1). One can easily extend the system if this block is implemented as a plug-in. It is also possible that addition of new features may help solve the problem in a better way. The "Feature calculation" block is responsible for available features. Extension of this block can also be done using the plug-in concept described above.

2.2 Acqusition

In Mr. Pawlak's research, all images were acquired using the hardware board MaxPci designed by Datacube. The input to this board was an RS-170A analog composite video signal from a standard RS-170A camera. The camera was mounted to an Olympus BH-2 microscope and the Datacube Max Vision digital image processing system was used for acquisition. The Max Vision system is equipped with an 8-bit analog-digital converter producing 256 intensity levels. An Olympus A100PL 1.30oil objective (type of objective that uses oil as an immersion media) was used. The camera was placed in the monocular path of the microscope using a special eyepiece with a magnification of 0.3. Images were saved in BMP format by the capturing software. BMP images were converted into TIFF format using Photoshop [13].

This method of obtaining data is good when the research datum is small. When non-technical people (such as doctors) are using the system, they must know how to capture an image, how to save it, and how to load the appropriate programs. A good method is to acquire images directly from the camera into the program. However, it is possible that higher resolution of acquired images might improve the performance of recognition. At such a point a new camera with higher resolution can be purchased. In order to support this, the program must often be rebuilt to support another camera. Furthermore if someone at another university would like to continue the research, they could have a different camera. In this case the program must be modified to work with the new camera.

Analyzing the situation one can see that a good solution would be to have an acquisition module be dedicated to just the acquisition of images. Once such a module is built for a particular camera, everyone

with such a camera could use it in their research. If a different camera is required one just needs to write the acquisition module for this camera and the rest of the system need not necessarily be changed.

In Mr. Pawlak's thesis, he proposed a way to improve his results by using images of the same region of the sample captured with different focus distance [2]. Automatic xyz translation stages make this possible. A control program is required in order to send the proper control signals to the xyz stage. Such a stage is very expensive and therefore is not widely available. With the plug-in methodology, various system configurations can be implemented, depending on the available equipment.

2.3 Segmentation

To find cells in the image Mr. Pawlak used a Region Growing algorithm (described in Chapter 4). Of course there are other methods and algorithms that could be used for this task. For his work he picked this method and it provided good results, but it would be interesting to test other methods as well. If segmentation is performed in a separate module, then anyone could test the system with different methods of segmentation and pick the best-suited one. Segmentation methods also depend on the type of images acquired by the camera. For example, in the case of color images, different segmentation methods are often required than for grey scale images. All this indicates that segmentation should be done in a separate module.

2.4 Displaying

To display the results of processing, the application program must be able to represent the data in a convenient form for users to interpret. How the data is represented depends on the type of processing performed and the data visualization desired. One of the tasks required in the analysis of human brain cells is to count how many cells of a particular type are in the view and what their locations are. This can be automated, however, it is also required to provide the ability for doctors to visualize this data to check for the correctness of the performance. This implies that that they should be able to see the captured image and the labels assigned to the cells.

Once doctors are confident with the results of the program, they could process many data samples without human intervention to gain statistical data. Thus, doctors do not need (or want) to check each

sample processed, but rather store the statistical result or have the printout of the data. The number of choices implies that displaying should be implemented as a plug-in. This would give maximum flexibility to the users.

2.5 Transformations

The one block left undefined in Figure 2.1 is the block labeled Polar Transform. I decided to design previously defined blocks, as plug-ins to the main application, which just connects them together, transporting appropriate data. However, should Polar Transform block be moved to another module, or maybe it can be treated as one of the defined blocks?

Let us look at the purpose of this block. It was designed to provide the method of classification with additional features that could not be extracted directly from the images. Look a little bit closer to the way features are extracted. Segmentation gives areas of interest and then features are extracted for each region of interest. There are two ways to perform this operation: 1) Pick the area of interest and extract all features; 2) Pick a feature and extract it for all areas of interest. I suggested that Polar Transformation should be treated as a usual feature that would save the transformed information. And this works properly for both cases, described above, i.e. features are extracted correctly.

2.6 Interfacing

It is obvious that all plug-ins, which could be built by different people, should have a common interface and use common data structures. One way to do this is to design a library, which would be used by all developers. The next question is, what information each of the plug-ins require for a correct performance? This information would show what has to be included into the library. Let us go step by step trying to understand what each type of the module does. We will define input and output data for each module adding more structures or changing them as we go along. The connection of two modules is shown in Figure 2.2

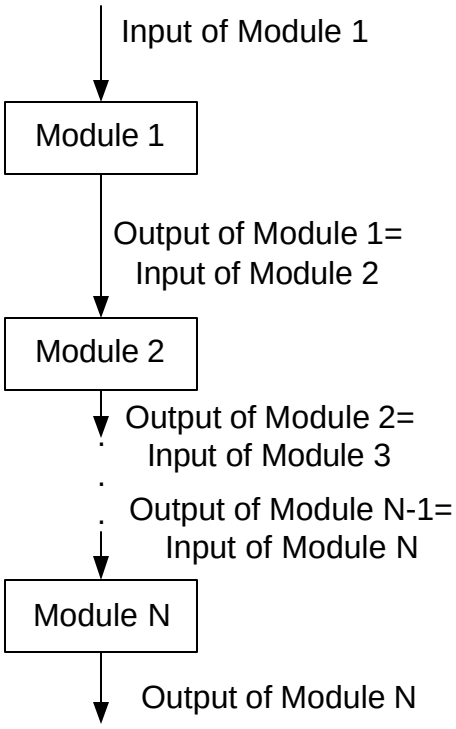


Figure 2.2 Module connection

2.6.1 Acquisition

It looks like the input data for the acquisition module should be a command to start the acquisition and some parameters for the acquisition itself (i.e. source of the acquisition if there is more then one camera; camera setup). This data could be represented by a set of variables with values. Which means that the first thing to have is a list of variable names with their values. The name of a variable is a set of ASCII (American Standard Code for Information Interchange) characters (i.e. a string), while the value could be a number or a string. Any number can be represented as a sequence of ASCII characters, so we need a list of pairs of strings: value – name.

The acquisition module has to acquire data and store it in the memory. We are working with the image data, which is a two-dimensional array of numbers or simply: an image. The images could have different size and different color depth. But the next question is if only one image is always acquired. The processing could require data from a previous image or, as in the case with cell recognition, images captured with different focal distances. It means that in general, acquisition should return a list of images.

2.6.2 Segmentation

Input data for this module is the output data of the acquisition module, i.e. the list of images. And of course, the segmentation could have its own parameters given by the list of variables as it was given for the acquisition module. The result of segmentation is a set of regions of interest. Each region of interest may be characterized by its bounding rectangle. Thus, the output of this module could be a list of bounding rectangles. It was decided to use this description for the region of interest.

2.6.3 Features

The output of the segmentation module is an input for the feature extraction module. We know that we could have a number of features and I decided that each module designed to extract a particular feature should go through the list of the bounding rectangles and extract information. We know that the bounding rectangles define the location of the region of interest in the acquired images. It means that feature extraction methods should be able to access the acquired images as well.

The output of the feature extraction method is a feature value assigned to a particular region of interest. It could be different for each region and there could be more then one value. To make it easier, let us describe the region of interest, not only with a bounding rectangle, but, also with a list of variables. Sometimes a transformation of the original image for the region of interest is required, so other plug-ins could extract some features. Let us add the transformation image to the description of the region of interest.

2.6.4 Classification

The classification is performed for each region of interest based on the values found in the list of variables for each feature. Also classification should have some prior information, we can call it a database, because it should have information about required features and some decision boundaries for all features. Output data is a decision that labels the regions of interest. The label could be added to the list of variables for each region.

2.6.5 Displaying

This module should have access to all data acquired and calculated during the performance. Depending on the requirements not all of the data needs to be displayed, but for the purpose of verifying and debugging all data should be accessible.

The output of this module depends on the task specification. It could be interactive displaying (as in my case, the program shows what were the decisions and allows browsing through the found region of interests) or just a notification about the decisions made during classification.

2.7 Modes of operation

The designed system should help a scientist to perform some time-consuming operations. Usually only the result of the whole procedure is required (like data collection or statistical distributions). It means that it is desired to have a system that performs the operations during some period of time without human interaction and produces the ready-to-use result. This mode of operation would be useful for testing purposes, when the system has to be checked for consistency of the performance.

On the other hand, if the plug-ins have parameters, it is not always obvious what settings would result in the best performance. In this case, there is a need to have an ability to go step by step through the operation of the whole system adjusting parameters of each part.

Analyzing these statements I came up with two modes of operation: Normal Mode and Research Mode. When in Research Mode of operation a user is able to set up each module and observe the effect produced by this module with those parameters. If the user is satisfied, these parameters should be used to make the recognition, otherwise the user should be able to change parameters and see the result of the change. It is also useful to see how different parts of the system work.

Once all parameters are set, the user should be able supply different data as input and see what the results are. This is the Normal Mode of operation.

2.8 Abstracting

The proposed sequence of the steps and their interfacing was designed just for implementing the methods proposed by Mr. Pawlak. But comparing this design with the designs mentioned in the textbook for pattern recognition by R. Duda, P. Hart and D. Stork [3], I found out that the proposed design could be used for many other recognition tasks. If there is a need to perform the same operations on each frame, then continues acquisition could be done. The only thing to add is a loop, so the acquisition is called again after

the displaying. It will work if the processing has sufficient speed otherwise some data could be lost. But it is a question of implementation and available hardware.

If the data structures from one loop of the acquisition are used for the next iteration, then a feedback is possible and a decision made during the previous loop can affect the procedure of the next one. I decided to create one data structure, which has all the required information for the system. The address of this structure is given to all plug-ins. The plug-in takes information that is required for its performance only. But in this way it could have access to any information, which makes the system very flexible. The same structure is used for the next acquisition step and control parameters could be changed in the previous step.

To test the described philosophy, the designed library was also used to solve the problem with bacteria counting (Chapter 5). Such points as porting into other operating systems and automatic memory management were considered during the design.

CHAPTER 3

IMPLEMENTATION EXAMPLE

A complete system should be created in order to test the conclusions of Chapter 2. The following chapter describes how the system can be constructed in order to support the ideas of plug-ins and use of a standardized library. I ported the automatic human brain cell recognition application and designed a bacteria counting application using the proposed philosophy to test the library. Cells and bacteria can be called micro objects because of their small size, and I named the system: Automatic Micro Object Recognition System (AMORS).

Different applications for this system differ in a configuration file that stores the parameters of the system.

3.1 Main application

In order to have an easy-to-use application, the application should have a user-friendly interface. Microsoft Windows 2000 and Visual C++ 6.0 were used to create and test the application. The Standard Template Library included into the ANSI (American National Standards Institute) standard for C/C++ was used in the design of the standardized library. That results in platform independence, so the library can be used under any operating system. The designed library was used to build the main program and plug-ins for the system. Plug-ins were implemented as DLLs (Dynamic-Link Library). Figure 3.1 shows the block diagram of the main application.

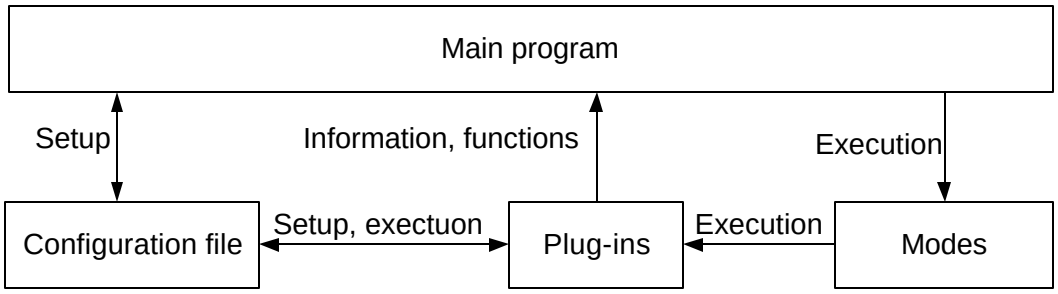


Figure 3.1 Main application organization

The main program is used to create (or modify) a configuration file with information about the plug-ins and their parameters to be executed. The main program also takes responsibility of loading plug-ins into memory and releasing them. In this way the library, which contains interface objects is free of platform dependent routines. The main program gives a user the ability to execute the system in Normal or Research Mode (see section 3.3), and stop the system during the performance.

3.2 System organization

Figure 3.2 shows the internals of the system. Explanations for each block are as follows.

Acquisition module – this block is responsible for acquiring data from the source (microscope) and converting it to the format described by the interface definitions. This module interfaces hardware with the rest of the system;

Camera manager – obtains the image data from the camera and transmits it to the acquisition manager;

XYZ stage manager – controls the position of the specimen under the microscope (depending on the equipment Z will be changed or the focus of the microscope will be changed in order to get other objects into focus);

Acquisition manager – controls the *Camera manager* and *XYZ stage manager* to obtain a set of frames. Once the frames are acquired, the manager transfers them to the *Processing module*;

Processing module – this block incorporates all routines required to perform transformation and classification of the objects;

Object separation– finds and separates different objects in the frames using the *Object separation plug-ins*;

Object separation plug-in – methods to separate the objects within frames;

Feature extraction – extracts features from each object using *Feature extraction plug-ins*;

Feature extraction plug-ins – methods for transformations and detection of the different features;

Classification – compare features for each object with *Database* to make a decision about the object label;

Database – contains the information about objects names and features that allows connecting the label and a particular object in the frame;

Display manager – displays the result of the analysis and a user can set some parameters using this manager, for the system to use them during the next iteration;

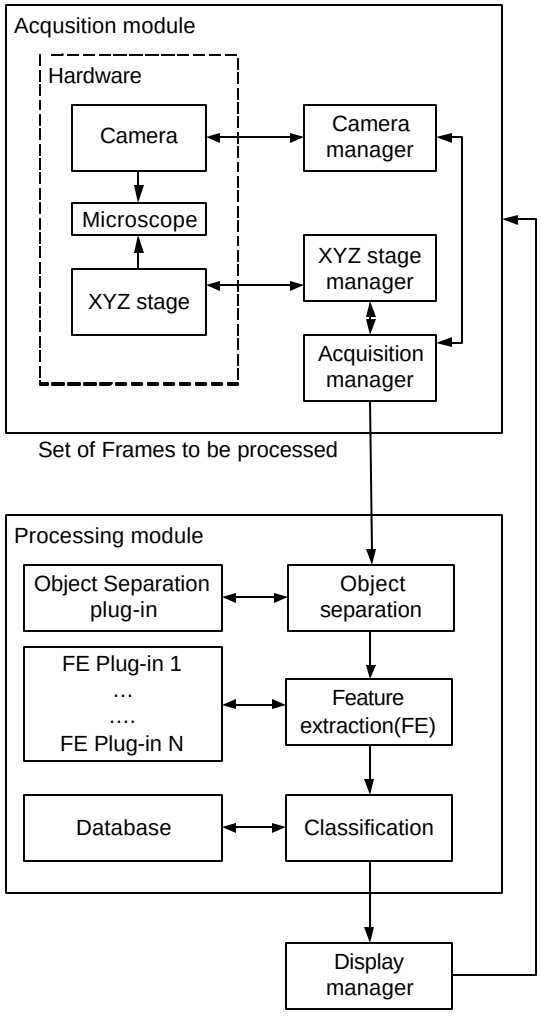


Figure 3.2 Block diagram of plug-in connections

3.3 Normal and Research Modes

When in Normal Mode (Figure 3.3), the system acquires data using the acquisition plug-in that composes data into frames. Each frame then undergoes a processing using the set of plug-ins defined by the current configuration. The result is directed to the display module where, depending on the requirements of the task, the program will wait for user interaction or continue with repeating the whole process, until some criteria are met.

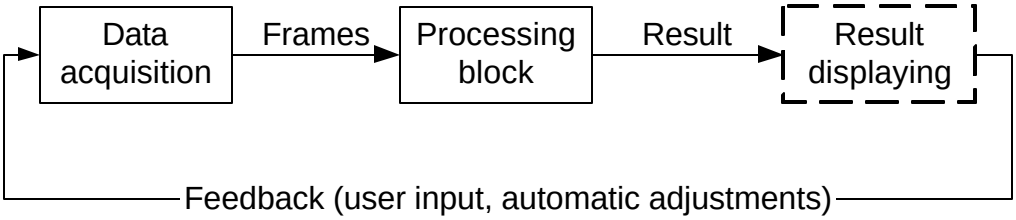


Figure 3.3 Block diagram for Normal Mode

When in Research Mode (Figure 3.4), the system interacts with the user allowing the user to adjust parameters for each step of the processing. Once all plug-ins are configured, the configuration file reflects the changes. This configuration file is then used for Normal Mode operation.

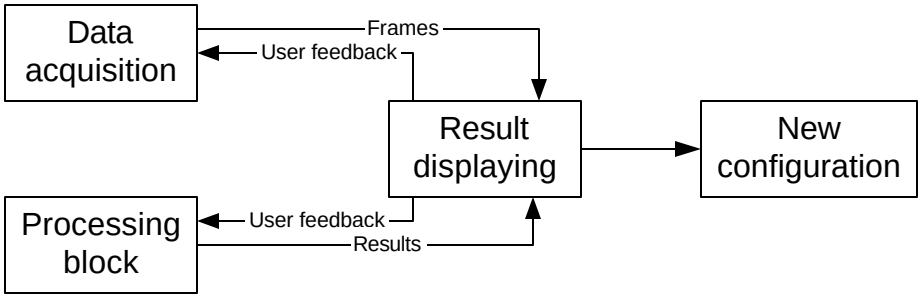


Figure 3.4 Block diagram for Research Mode

3.4 AMORS user interface

Users should have an access to the functionality of the system and the usual way to do so is to design a GUI (Graphical User Interface). Figure 3.5 shows the main window of AMORS. This window allows user to configure the system and perform recognition.

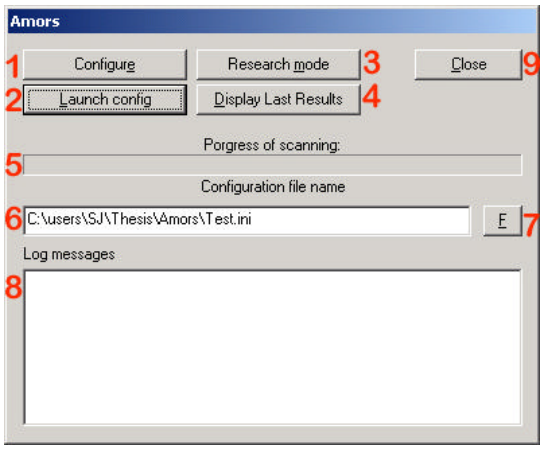


Figure 3.5 Main window of AMORS

Explanation of controls:

1. Configure button allows user to configure the plug-ins for the current configuration file (see item 6);
2. Launch button allows user to launch the system to perform the task according to the current configuration file in Normal Mode (see Chapter 2 Interface description for more information);
3. Research Mode button allows a user to enter the Research Mode (see Chapter 2 Interface description for more information). Step by step setup of each plug-in with displaying the results of the setup is performed;
4. Display last result button allows a user to see the result obtained after using the Launch button;
5. Progress bar shows the progress reported by the system and plug-ins;
6. Configuration file name shows the name of the configuration file used to guide the system work;
7. Browse for configuration file button launches file-open dialog box, where user can pick the configuration file;
8. Log messages shows messages reported by the system or the plug-ins;
9. Close button closes the window and exits the program.

When “Launch config” or “Research Mode” buttons are pressed, the program reads the configuration file, loads plug-in dlls and executes appropriate function for the Normal or Research Mode. For this time user has to wait for the end of the operation to be able to use this window, except that user can abort the operation. Figure 3.6 shows the appearance of the window during the performance.

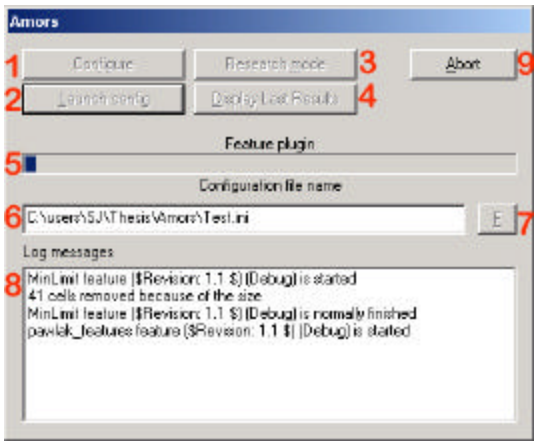


Figure 3.6 Main window during the execution

To configure the system, the user is provided with the dialog box (Figure 3.7), where it is possible to choose plug-ins and set up their parameters.

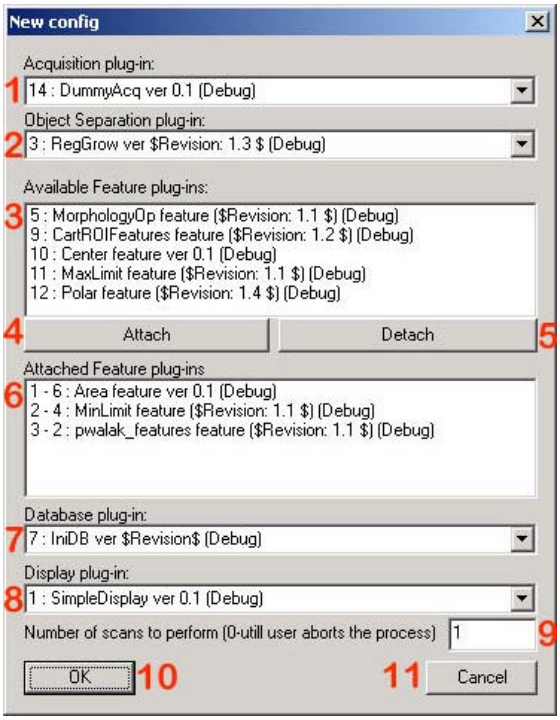


Figure 3.7 Configuration dialog box

Explanation of controls:

1. Name of the Acquisition plug-in (press arrow on the right to see the list of all available Acquisition plug-ins);

2. Name of the Object Separation plug-in (press arrow on the right to see the list of all available Object Separation plug-ins);

3. All available Feature plug-ins;

4. Attach button attaches the selected Feature plug-in to the configuration;

5. Detach button removes the selected Feature plug-in from the configuration;

6. Feature plug-ins, which will be used in the system;

7. Name of the Database plug-in (press arrow on the right to see the list of all available Database plug-ins);

8. Name of the Display plug-in (press arrow on the right to see the list of all available Display plug-ins);

9. Number of scans to perform during the execution. System will loop for this number of execution giving a number of iteration to the Acquisition plug-in. If 0 is used, loop will be infinite, so only user interaction (pressing the Abort button) will stop it;

10. Ok button – when all plug-ins are chosen, press this button to launch setup for each individual plug-in;

11. Cancel button – if you do not want to save any changes use this button.

Name of the plug-in consists of two parts: unique identifier of plug-in and information returned by GetInfo function from the plug-in separated by [space colon space] sequence.

Example:

1 : Simple Display ver 0.1 (Debug)

where the unique identifier is “1” and the description returned by the plug-in is “Simple Display ver 0.1 (Debug)”. Unique identifier is needed in the case if two plug-ins would return the same information. But in most cases user could ignore this number.

The order of Feature plug-ins is important for execution, so in the list of attached plug-ins the order number is added to the name of plug-in following by the dash. Plug-in is attached to the end of the list and if you want to have a particular sequence you should attach the plug-ins in the desired order. If you make a mistake, detach the plug-in and repeat the operation.

3.5 General purpose plug-ins

Mr. Pawlak used TIFF images in his research. To compare the results of my system and his results, I had to use the same data. I have created special acquisition plug-in that reads TIFF image files. Also I have designed SimpleDisplay plug-in to see the results of performance. Even though these plug-ins were designed for human brain cell recognition system, they are very versatile and can be used in a big variety of applications at least at the beginning stage. These plug-ins are described in Appendix D.

CHAPTER 4

AUTOMATIC HUMAN BRAIN CELL RECOGNITION

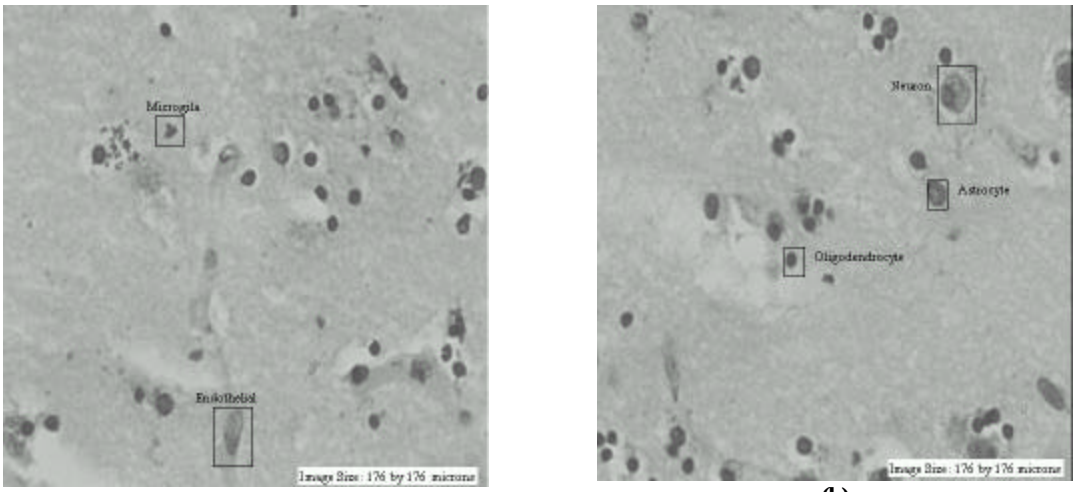
This chapter describes the application created using the research of Mr. Pawlak [2], who was a former graduate student in the ECE department at UNH. His thesis was a research on the possibility of automatic human brain cell recognition to provide help with the pathological study of Huntington's Disease (HD). His research was performed using Matlab, where algorithms were coded. This code was used as the basis for the modules of AMORS to demonstrate how such a task could be performed using the proposed system. Modifications of the original research are discussed. In this chapter I have paraphrased portions from the work of Mr. Pawlak [2] to give a brief explanation of the problem. Appendix D contains the detailed information about the plug-ins discussed in this chapter.

4.1 Problem statement

One of the first regions of the brain to be affected by HD is the Caudate Nucleus, thus the images contained in this thesis are primarily from that region. Neuropathological studies of HD show that as the disease progresses there are decreased neuronal and increased oligodendroglial densities in the Caudate Nucleus [2]. One primary method used to perform the above mentioned pathological study is to observe brain tissue under a microscope, which has a *camera lucida* apparatus. This device acts as a prism, and can be used to superimpose tracing paper on to the slide. Cells are then traced by hand, and identified by an expert (such as Dr. Vonsattel). Clearly, this process is very labor intensive, and can be made faster with the benefit of computers and digital image processing techniques. It is surprising however, that the practice of creating hand sketches during microscopic studies is rather common in modern medical research [2].

Brain tissue found in the Caudate Nucleus primarily consists of these cell types: astrocytes, endothelial, microgila, neurons, and oligodendrocytes. Astrocytes are approximately 10 microns in

diameter, have a grainy texture, and usually are oval or bean shaped. Endothelial cells are blood cells usually found as part of capillary vessels. Microgila cells are rod like or cigar shape in appearance with a process (a faint stringy structure) at each pole (end) and are not associated with blood vessels. Neurons are irregular in shape, and usually are grainy in texture. They are commonly 20 to 30 microns in diameter, but sometimes appear smaller because microscope tissue sections are often cut to a thickness of approximately 7 microns. It is not unusual to observe the dark nucleus of the neuron, which is approximately 1 to 2 microns in diameter. Oligodendrocytes are very dark and circular in nature, much like a dot of ink. They are usually 5 to 7 microns in diameter. Sample brain tissue may be found in Figure 4.1 (a) and (b) where an example of each of the cell types is clearly identified.



(a) **(b)**

Figure 4.1 Examples of cells (a) - Microgila and Endothelial; (b) - Neuron, Astrocyte and Oligodendrocyte

Automation of the cell identification process offers several advantages in addition to reducing the labor involved. Having microscope images appear on a video monitor instead of having to look through the eyepiece greatly reduces eye strain. The automatic computer identification of regions of interest gives the by-product of a computer database, which can be recalled rapidly by researchers throughout the world. Databases of this nature will also record statistics about the cells such as their size and shape factors. Often the experts will have different opinions as to the classification of the different cell types. Formation of a large database will allow researchers to discuss the differences and perhaps reach a better understanding of the roles of the various cell types.

4.2 Image segmentation

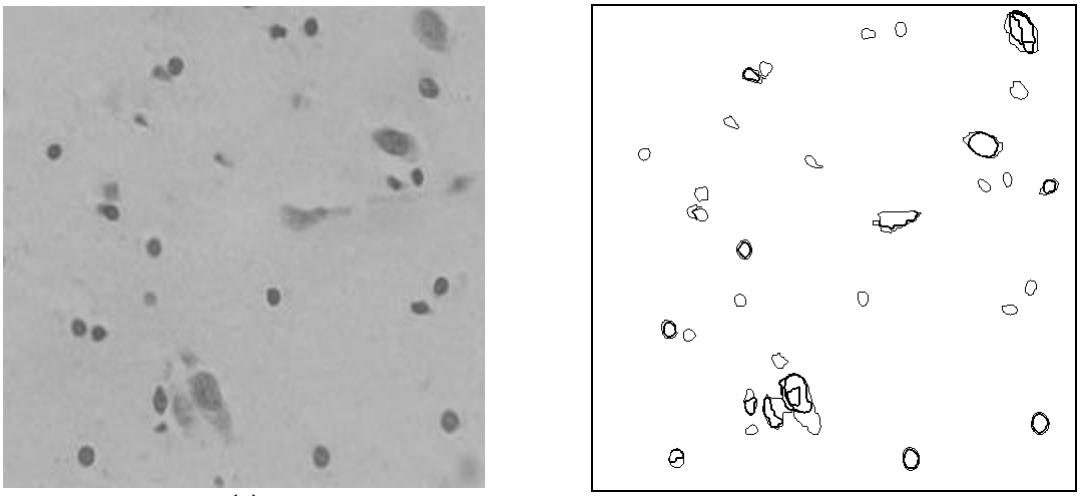
"Segmentation of an image entails the division or separation of the image into regions of similar attribute" [4]. "The ultimate aim in a large number of image processing applications is to extract important features from the image data, from which a description, interpretation, or understanding of the scene can be provided by the machine" [5]. In brief, segmentation determines the regions of interest (ROIs) in an image. This does not mean that the segmentor will try to determine the type (classify) of the region, but merely determine the pixels in an image, which belong to the same item. Specifically, in the application of automatic human brain cell recognition, it is the segmentor's function to find all the brain cells in the image, and to identify them with unique "labels" (i.e. Region 1, Region 2, …).

Over the years, the digital image processing community has developed several segmentation methods, many of them *ad hoc*. Four of the most common methods are: 1) amplitude thresholding, 2) texture segmentation, 3) template matching, and 4) region-growing segmentation. In the first method, amplitude thresholding or window slicing, the image may be arbitrarily subjected to a fixed number of thresholds. Pixels, which fall between an upper and lower threshold, and are in close spatial proximity, are considered to be the same region. Variations of this method will take a histogram of the image and key on peaks or valleys of the histogram to set thresholds. This method was not used because it can often lead to regions, which produce concentric circles when trying to segment the cells. In the second method, texture segmentation, a coarseness metric is computed on the image, and changes in this metric are used to determine boundaries. Texture segmentation is generally considered to be a challenging problem, and was not used because the textures of neurons are considerably different than those of oligodendrocytes. The third method, template-matching segmentation, involves analyzing an image to look for matches against a list of templates. The large variation in brain cell shapes and sizes makes this method impractical. In the fourth method, region-growing segmentation, items of similar amplitude are located based on a seed pixel. Pixels near the seed are analyzed to determine if they are part of the same region. Generally, region-growing segmentation is not as prone to the concentric circle problem as the window slicing approach. The region-growing method is explained in detail below.

The region-growing process used in this thesis is based on the concept of computer stacks [6]. Stacks operate in much the same way as a stack of plates in a cafeteria, that is to say they are a last in - first

out buffer. When pixels of interest are discovered, the neighboring pixels are analyzed to see if they are part of the same region. If so, they are placed on the stack for later processing. During the processing, the stack will initially grow, go through a phase where it will both grow and shrink, and will end by shrinking completely to an empty buffer.

Three arrays are used during region-growing segmentation. The first array is the grayscale image itself, (see Figure 4.2 (a) as an example). The second array indicates if the pixel of interest has already been processed. The third array consists of cell labels. At the end of segmentation, the input image (first array) is transformed from intensity pixels to labeled regions (third array). Figure 4.2 (b) illustrates the transformed image of Figure 4.2 (a). The transformed image shows only the perimeter pixels of all of the regions. Using this method, the boundaries of all ROIs can easily be seen. Pixels inside of the boundaries are all assigned the same label. In addition, the method clearly illustrates the cases where regions are adjacent to (or associated with) each other. This frequently occurs with neurons because the cells are close to the background intensity levels of the slide and often have local regions of different gray levels.



(a) **(b)**

Figure 4.2 (a) original grayscale image; (b) transformed image

Processing begins by selecting a threshold used to analyze the image. Next, the image is raster-scanned (column by column, then row by row) to see if a pixel crosses the trigger threshold and does not already belong to a region. If it has met these conditions, the region-growing algorithm listed in Figure 4.3 is used with the trigger pixel designated as the seed value. If the resultant region is not large enough, the region is not used, and the processing and cell label arrays are properly reset.

```
top:
 do i_loop   seed_i-1 to seed_i+1
  do j_loop    seed_j-1 to seed_j+1
   if not processed
    if input in threshold range
      push i,j on to stack
    end if in threshold range
     set pixel in processing array
     put pixel in region pixels
   end if not processed
  end j_loop
 end i_loop

if anything on stack
  pop stack to change seed
  go to top
end if anything on stack
```

Figure 4.3 Region-Growing Algorithm

This method used by Mr. Pawlak was implemented in the RegGrow plug-in (Object Separation type). Mr. Pawlak found this method more suitable and robust, than other methods (contour method, for instance). But the results still required a human interaction; otherwise some cells will be treated as one cell if they are close to each other. Without human interaction 6.7% (on average) of cells in the view will be segmented incorrectly. Also one has to take into account that on average 50% of cells in the view is clutter or could not be classified even by human. That means that error due unattended segmentation is less then 5% (on average half of improperly segmented cells are clutter), so I decided to leave the automatic segmentation without human interaction.

4.3 Polar transform

The polar transformation used in this thesis is illustrated in Figure 4.4. It is simply a remapping of the Cartesian coordinate input image plane to a polar coordinate-mapped image plane. The most basic transformation example is that of a circle, in which the polar mapping results in a vertical line. Lines, which extend radially from the origin, result in horizontal lines. If an object is rotated, the polar mapping of the object is shifted vertically from the original.

Polar mapping can be very useful in applications dealing with circular objects such as cells. In addition, the polar mapping provides very good insight into the information contained in the edges of the items of interest. It was this property that was tested for its robustness in determining if some of the cell types had thick membranes. If so, the information would have been used in the cell classification process. At the University of New Hampshire, polar mapping has been used for such diverse applications as optical character recognition and image compression [7,8]. In the case of the optical character recognizer, the edge information was used as a feature vector. In the image compression case, polar mapping was used to preserve high-resolution information in the field of interest while reducing data at the periphery.

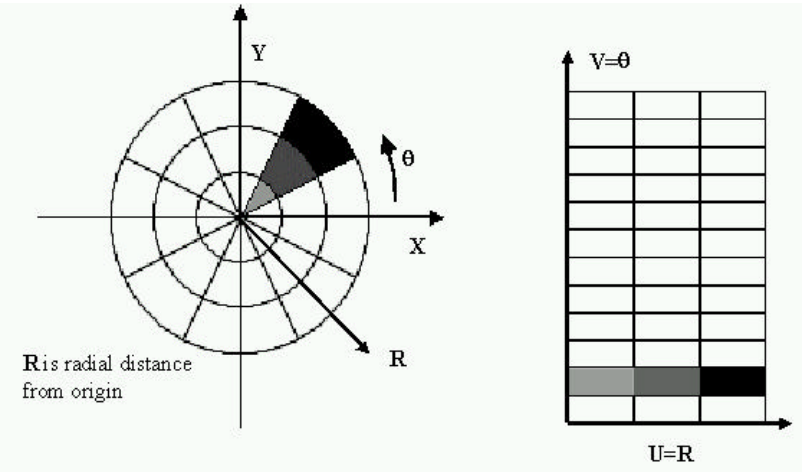


Figure 4.4. Polar Transform Mapping diagram

The Polar transform was implemented in the Polar plug-in (Feature type).

4.4 Feature plug-ins

Mr. Pawlak [2] suggested a set of features which could be used to classify cells. He described them in his work and implemented them in Matlab. I have implemented his methods into the pawlak_features plug-in (Feature type). These features include: central moments, area, roundness, distinctness, standard deviation, centroids, elongation, eccentricity, spread, orientation, minimum radius, maximum radius, mean value, entropy and some features from the polar transformed image.

Matlab has a special way of operating with data loaded from TIFF files (usually all values are increased by 1, so there is no value 0), and some functions use different formulas, which return slightly different results from what I was using in my plug-ins. Also without human refining of segmentation (used

my Mr. Pawlak) the boundaries of cells are slightly different. As a result, parameters calculated by my plug-in and those calculated in Matlab are different by less then 5%. It is just a different way of treating the data, so this fact does not affect the precision of values and the relation between different parameters. It was decided not to spend time trying to change my code to have exact values as given by Matlab, even though it is obviously possible.

The only big difference in parameters was found for the calculation of entropy. I checked the formulas given for this feature and the way it was calculated in Matlab. I found that Mr. Pawlak did not scale the data properly. It is a scaling factor so it did not change the effect of the feature. But I decided to leave a correct formula in my plug-in. As a result I had to change the membership function for the entropy.

4.5 Fuzzy Logic

To some, the term "Fuzzy Logic" conjures up the concept of an oxymoron. However, it is one of the largest growing avenues in machine intelligence. Fuzzy Logic systems are developed using linguistic terms and allow for a pattern identification system to be designed using small training sets. We like to think of ourselves as very precise, and calculating, when in fact one of our greatest strengths as human beings is the ability to make abstract generalizations using a few adjectives and nouns. How big is a large cat? The human mind does not approach this problem by thinking. "Garfield weighs 9.857 kilograms. Cats greater than 7.23 kilograms are large cats. Therefore, Garfield is a large cat." The mind does not place Garfield into to the set of large cats using such a precise and sharp threshold. Rather, the definition of the set of large cats has a fuzzy boundary. A cat, which weighs 6 kilograms, has some membership in the set of large cats, but not as much as a cat which weighs 8 kilograms. It is evident that the mind describes associations or membership in a set in vague terms. Fuzzy Logic parallels this concept by defining "membership functions" which assign a grade of membership ranging between zero and one. In the case of the 8 kilogram cat the membership value may be 0.95, whereas the 6 kilogram cat may be given a membership value of 0.45.

Membership functions are a mapping of a single linguistic variable from a parameter space to a linguistic space. In the Garfield example, the weight (parameter) was mapped to a large size (linguistic) space. Often these membership functions are simple functions such as ramps or triangles, but they can be as

complicated as the designer chooses. In some systems it may be desirable to have multiple size categories such as small, medium, and large. One simply defines three membership functions which all use the same input parameter.

Usually more than one feature characterizes the object of interest. Now the combination of these features defines the object. There could be more then one combination for the same object. Let us call these combinations – rules. In the example with the cat our rule is simple if size is large then cat is big. If we say that a large cat is a cat with large size and small or medium length tail, then the first rules is: if size is large and length of the tail is small it is a big cat and the second rule is: if size is large and length of the tail is medium cat is big.

A special database plug-in, IniDB, was created to support this philosophy. The database for this plug-in is a configuration file which has membership functions for parameters. Then the rules are defined, and objects with the names and rules are used to make the decisions. This plug-in is described in Appendix D.

4.6 Performance

Because of the difference in feature calculations and the changed membership functions, only the final results could be compared. Figure 4.5 shows the configuration of the plug-ins in AMORS.

Because of automatic segmentation less then 5% of the cells were ignored. I had to modify the membership function for the entropy. It is very time-consuming to define the membership functions that would match the rules described by Mr. Pawlak. After changing the membership function and the rules, I achieved 70% of recognition (that is 5% less than with human interaction in the original research). I believe that with more adjusting of the membership functions a better performance can be achieved. The main goal was to illustrate how using the philosophy of this thesis one can solve such a recognition task, rather than to obtain a very high percentage of recognition for this particular classification scheme. Also using self-organized maps was the next step to the more precise recognition. Their performance is described later. Keeping all that in mind I decided to leave fuzzy-logic membership functions with this percentage of recognition.

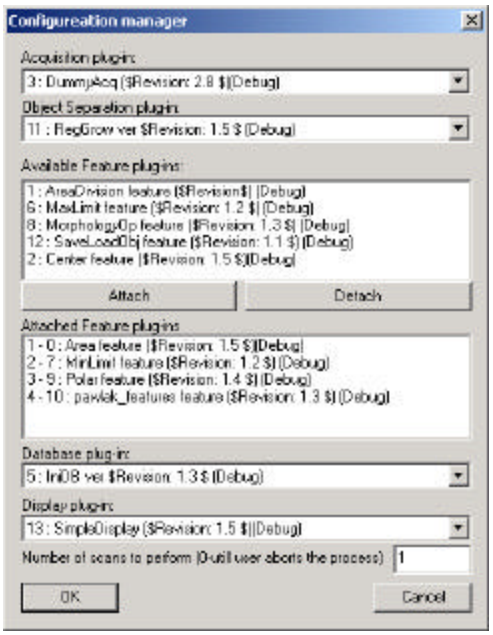


Figure 4.5 The configuration of the application for AMORS.

On the other hand, the main advantages of the implemented system are the easy-to-use interface, speed and less human interaction. I did not perform optimization on the code for the calculations of features and region growing method. I wanted them to be as close to the Matlab code as possible (for easier debugging). But the implemented system works 2.5 times faster then Matlab code. This was expected, because Matlab interprets the code, while in AMORS everything is compiled.

The next step was to improve the methods. Mr. Pawlak proposed a couple of improvements to make his methods more robust and precise. The first one is to make three-dimensional processing and another one is to use self-organizing maps instead of membership functions that are hard to define. The following sections describe these improvements.

4.7 3D acquisition

Three-dimensional digital image processing techniques for automatic human brain cell identification offer the largest performance accuracy gains for the techniques discussed in the original work of Mr. Pawlak [2]. The primary limiting factor that anyone experienced when marking the cell types was the inability to adjust the focus of the slide. The depth of field for the microscope's objective used in the image capture process was less then one micron. Selection of the objective is a trade off between the

resolution and depth of field of the imaged tissue. Medical experts are not typically concerned with the depth of field because they will simply adjust the focus if necessary. In an attempt to remedy the focus adjustment problem, University of New Hampshire graduate student John Canfield recorded "stacks" of microscope images [2]. This “stack” is the set of images for the same view with different focus.

There are things that could be done to improve recognition using stack images. The first one is to recognize a cell at each image and then decide the label by voting. Or the image where the cell has the sharpest edges could be used for recognition. Also some additional information could be extracted, like position of blood vessels. Then this additional information could be used for recognition [2].

I decided to show that AMORS is suitable for such tasks, by implementing a decision scheme by voting. No additional plug-ins were required to do so. The DummyAcq plug-in is capable of loading multiple images into memory, imitating the result of the acquisition with different focus planes. Then each image is treated like it was done in the original research. IniDB asks a user if the sequence should be treated as a 3D stack. And based on the labeling for each cell, it assigns the label with the highest vote for all layers. There was not enough data to really test the efficiency of this modification, because only one “stack” was captured. But the main idea was to show that the system could handle this approach.

4.8 SOM database

Fuzzy logic membership functions documented in this thesis were developed in an *ad hoc* fashion. As such, they may not yield optimal classification accuracy. It is possible that alternate membership functions could be developed which would yield better results. Two factors should be considered in the development of new membership functions: a systematic approach, and the size of the database. Kohonen Self-Organizing Map (SOM) represents a more systematic approach to the development of membership functions [9]. These maps can be used to systematically partition a multi-dimensional decision space, and thus assist in the automatic definition of the membership functions. Any adjustments to the membership functions also have to account for the size of the database. If the database is small (which is the case of this thesis), one must be very careful not to “tune” the membership functions to the available data.

4.8.1 Self-organized maps

The SOM here defines a mapping from the input data space $\boldsymbol{R^n}$ onto a regular two-dimensional array of nodes. With every node $\boldsymbol{i}$, a parametric reference vector $\boldsymbol{m_i}$ in $\boldsymbol{R^n}$ is associated. The lattice type of the array can be defined as rectangular or hexagonal the latter is more effective for visual display. An input vector $\boldsymbol{x}$ in $\boldsymbol{R^n}$ is compared with the $\boldsymbol{m_i}$, and the best match is defined as "response": the input is thus mapped onto this location.

The array and the location of the response (image of input) on it are supposed to be presented as a graphic display. For a more insightful analysis, each component plane of the array (the numerical values of the corresponding components of the $\boldsymbol{m_i}$ vectors) may also be displayed separately in the same format as the array, using a gray scale to illustrate the values of the components.

One might say that the SOM is a "nonlinear projection" of the probability density function of the high-dimensional input data onto the two-dimensional display. Let $\boldsymbol{x}$ in $\boldsymbol{R^n}$ be an input data vector. It may be compared with all the $\boldsymbol{m_i}$ in any metric; in practical applications, the smallest of the Euclidean distances $\|\boldsymbol{x} - \boldsymbol{m_i}\|$ is usually made to define the best-matching node, signified by the subscript c:

$$\|x - m_c\| = \min\{\|x - m_i\|\} \text{ or } c = \arg\min\{\|x - m_i\|\} \quad \text{(Equation 1)}$$

Thus $\boldsymbol{x}$ is mapped onto the node c relative to the parameter values $\boldsymbol{m_i}$.

An "optimal" mapping would be one that maps the probability density function p(x) in the most "faithful" fashion, trying to preserve at least the local structures of p(x). (You might think of p(x) as a flower that is pressed!) Definition of such $\boldsymbol{m_i}$ values, however, is far from trivial; a number of people have tried to define them as optima of some objective (energy) function [9]. In this work I used the stochastic-approximation-type derivation [9] that defines the original form of the SOM learning procedure.

4.8.2 Learning

During learning, those nodes that are topographically close in the array up to a certain distance will activate each other to learn from the same input. Useful values of the $\boldsymbol{m_i}$ can be found as convergence limits of the following learning process, whereby the initial values of the $\boldsymbol{m_i}(0)$ can be arbitrary, e.g., random:

$$m_i(t+1) = m_i(t) + h_{ci}(t) \cdot [x(t) - m_i(t)]$$

where t is an integer, the discrete-time coordinate, and $h_{ci}(t)$ is the so-called neighborhood kernel; it is a function defined over the lattice points. Usually $h_{ci}(t) = h(\|r_c - r_i\|; t)$, where $r_c \in R^2$ and $r_i \in R^2$ are the radius vectors of nodes c and i, respectively, in the array. With increasing $\|r_c - r_i\|$, h_{ci} goes to 0. The average width and form of h_{ci} defines the "stiffness" to be fitted to the data points. Notice that it is usually not desirable to describe the exact form of $p(x)$, especially if x is very-high-dimensional; it is more important to be able to automatically find those dimensions and domains in the signal space where x has significant amounts of sample values!

In this research two options for the definition of $h_{ci}(t)$ are available. The simpler of them refers to a neighborhood set of array points around node c. Let this index set be denoted N_c (notice that we can define $N_c = N_c(t)$ as a function of time), whereby $h_{ci} = \alpha(t)$ if $i \in N_c$ and $h_{ci} = 0$ if $i \notin N_c$, where $\alpha(t)$ is some monotonically decreasing function of time ($0 < \alpha(t) < 1$).This kind of kernel is nicknamed "bubble", because it relates to certain activity "bubbles" in laterally connected neural networks.

Another widely applied neighborhood kernel can be written in terms of the Gaussian function,

$$h_{ci} = \alpha(t) \cdot \exp\left(-\frac{\|r_c - r_i\|^2}{2 \cdot \sigma^2(t)} \right)$$

where $\alpha(t)$ is another scalar-valued "learning rate", and the parameter $\sigma(t)$ defines the width of the kernel; the latter corresponds to the radius of N_c above. Both $\alpha(t)$ and $\sigma(t)$ are some monotonically decreasing functions of time, and their exact forms are not critical; they could thus be selected linear.

The next step is calibration of the map, in order to be able to locate images of different input data items on it. In the practical applications for which such maps are intended, it may be usually self-evident from daily routines how a particular input data set ought to be interpreted. By inputting a number of typical, manually analyzed data sets and looking where the best matches on the map according to Equation 1 lie, the map or at least a subset of its nodes can be labeled to delineate a "coordinate system" or at least a set of characteristic reference points on it according to their manual interpretation. Since this mapping is assumed to be continuous along some hypothetical "elastic surface", it may be self-evident how the unknown data are interpreted by means of interpolation and extrapolation with respect to these calibrated points.

4.8.3 Example

Let us look at the simple example to understand how self-organizing map works. Consider a recognition problem, where one has to separate three types of vegetables: cucumbers, tomatos and watermelons. A picture of each vegetable is made using the same scale. Somehow the program locates the vegetable in the image and finds the area occupied by the vegetable and the maximum distance between two perimeter points of the vegetable (Figure 4.6).

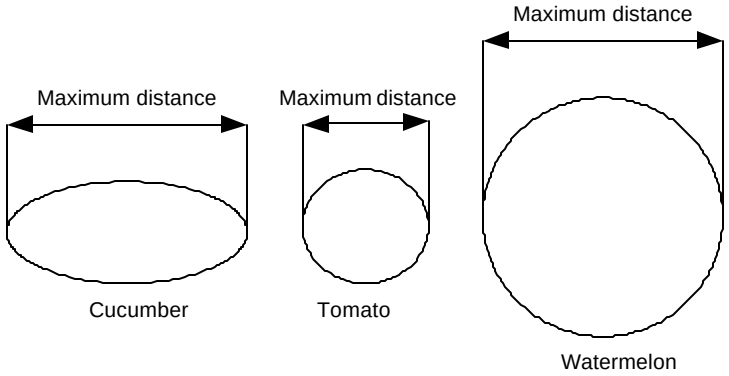


Figure 4.6 Illustration of the maximum distance.

These parameters are used as the separation parameters. We know before hand that on average a watermelon is round and is bigger then a tomato, which is round as well, while a cucumber has the area close to the area of a tomato but the maximum distance close to the watermelon. This is shown in Figure 4.7.

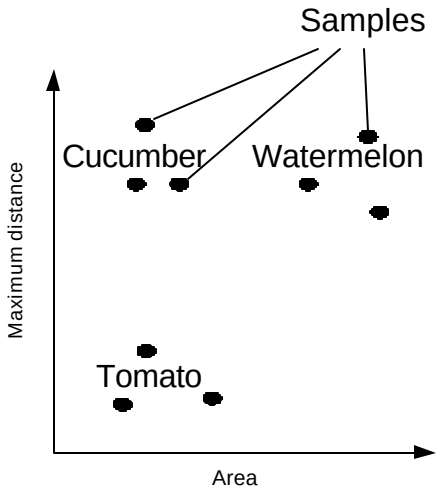


Figure 4.7 The distribution of input samples

One can see that these three classes are easily separated, which makes it easier to check the results. Now let us look how self-organizing maps can handle this situation. Figure 4.8 (a) shows nodes of the map with size 5x4 using hexagonal lattice type. Nodes were randomly placed to form a uniform distribution as shown in Figure 4.8 (b).

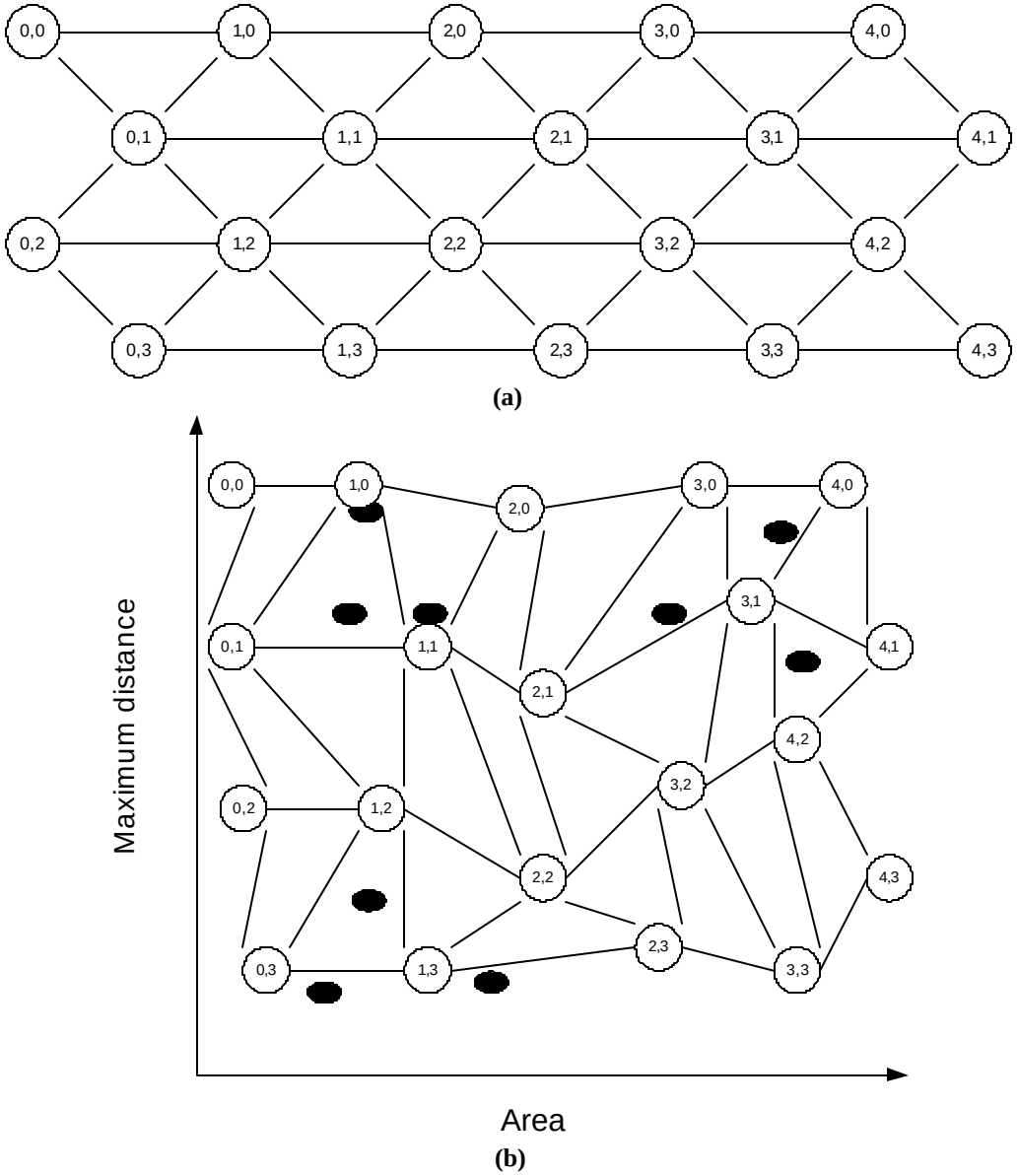


Figure 4.8 (a) map with the size 5x4 (b) randomly initialized map in the sample space

During the training distances between each node and a training sample is calculated. The closest to the sample nodes are moved towards the location of the sample. Figure 4.9 shows the trained map after a number of itterations. Training samples are shown as the black dots.

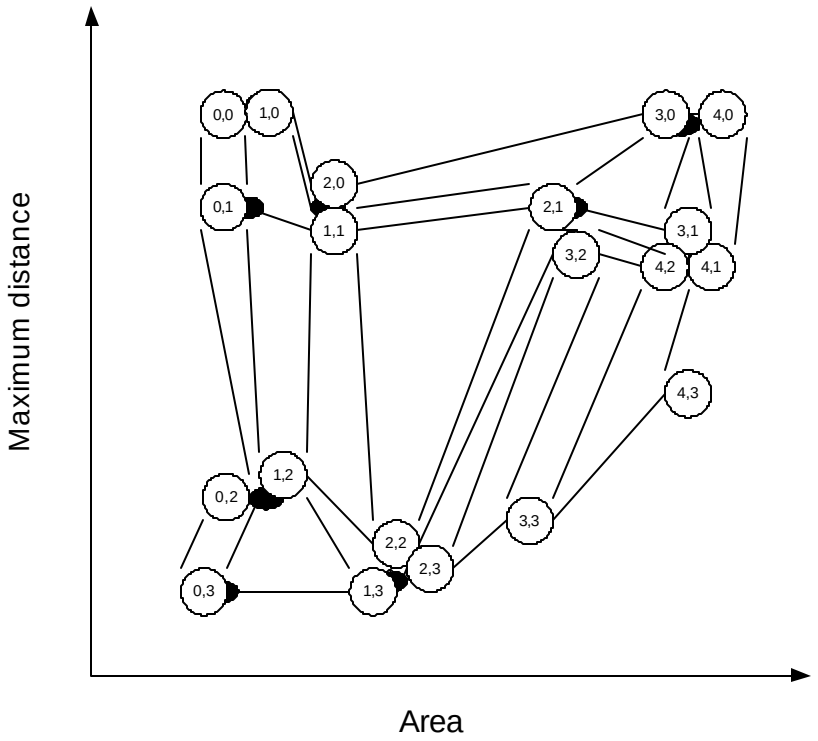


Figure 4.9 The map after training

This is a very simple two-dimensional case, where it is very easy to show the organization of the map. But when the dimensionality of the problem is more then 3, there is no direct visual representation of the map in the sample space. But even if each node is represented as a multidimensional vector, all nodes are organized in a two-dimensional array as shown in Figure 4.10

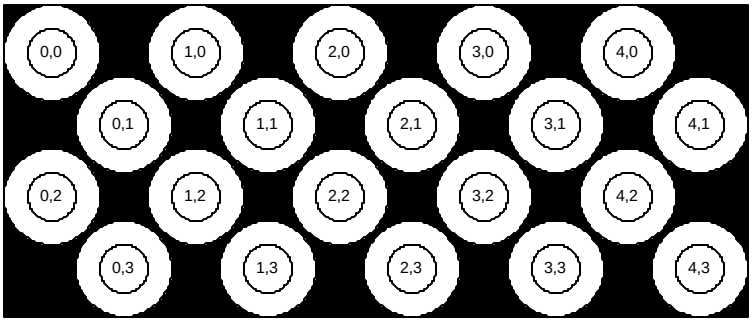


Figure 4.10 Representation of the map

Once the map is organized and the nodes are labeled, a user can see the result as shown in Figure 4.11, where C stands for cucumber, T – for tomato and W for watermelon. Nodes (4,3) and (3,3) were too far from any sample and left unlabeled, while the rest of nodes were labeled with the labels of the names

defined for the samples. When a new sample is acquired, the distance between all nodes and the sample is calculated and this sample is assigned the label of the closest node.

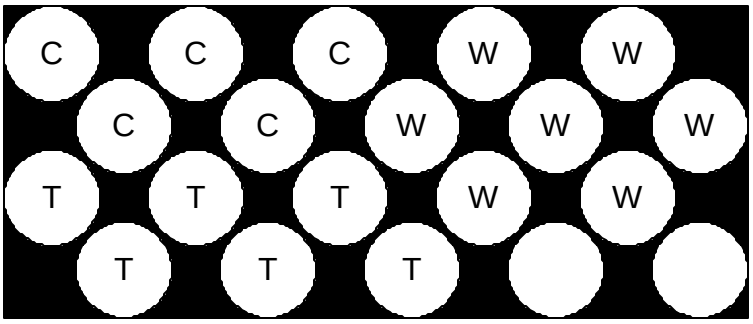


Figure 4.11 Labeled map representation

4.8.4 Implementation

I used "The Self--Organizing Map" Program Package Version 3.1 designed by T. Kohonen, J. Hynninen, J. Kanga, J. Laaksonen [10] in order to make the classification. This package defines 4 steps: initializing map, training map, error evaluation and monitoring. Different functions take care of each step.

Initializing creates the map of a given size with uniformly distributed nodes. During the training nodes are shifted to cover clusters of input data. Labeling is used to assign names for the nodes. The map is composed from a finite number of nodes and because of that the input vector does not match an exact node; instead the closest match is assigned. The average difference between the input vector and the best-match vector is the quantization error. Monitoring is when actual decision is made, a label of the best-matching node is assigned for each input sample.

I wanted to show one more way of extending the system. The SOM software package was compiled into 4 separate files, one for each step. Each executable file takes parameters and data from a given file and outputs the result into another file. Then I created the SOMDB Database plug-in, which creates input data files, executes each step and reads the output files. Such approach allows one to modify the algorithm used for each step without changing the plug-in, as long as the parameters and the data file format stays the same. The goal of such an approach is to create a GUI (implemented in a plug-in), which may stay the same. The plug-in and program parameters are described in Appendix D.

There are two modes of operation for this plug-in. The first one is the "data collection" step and the second one is "monitoring". During the "data collection" step the data file for the map training is

created. The plug-in asks a user to label known cells, so this information can be used for the training and labeling. Once the user supplied all the information it is saved. This information is helpful when there is a need to repeat the same operation but with different training parameters. When all information is gathered, the map is initialized and trained. The quantization error is calculated and shown.

During the "monitoring" step the program uses a trained map to assign labels.

4.8.5 Results

Disadvantage of using membership functions was the problem of their definition. Using SOM solves this problem. In addition, it allows one to analyze what features are beneficial to give more separation of the classes. Mr. Pawlak used five features for cell separation: area, distinctness, entropy, roundness and maximum radius. Figure 4.13 shows the representation of how these features separate cells. The map was constructed as 12 by 8 grid of nodes. After training the quantization error was 6.06. Figure 4.12 shows the parameters used for training of all maps shown below.

Name\Step	Ordering step	Fine tuning
Number of iterations	1,000	10,000
Learning rate	0.05	0.01
Radius of training area	10	3

Figure 4.12 Training parameters for the SOM

Abbreviations used in the following figures are N for Neuron cells, A for Astrocyte cells, O for Oligodendrocyte cells, * for unknown cells or clutter. Nodes without marks were not labeled.

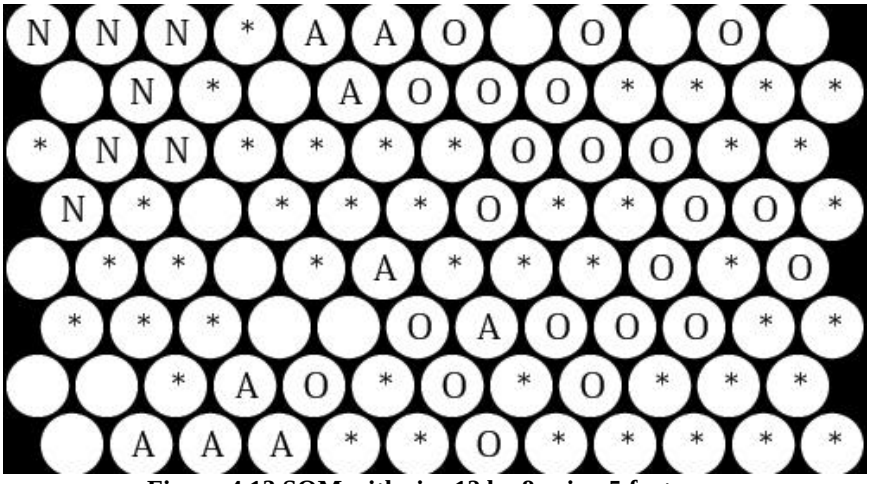


Figure 4.13 SOM with size 12 by 8 using 5 features

As one can see from Figure 4.13, three types of the cell are separated quite well. Neuron cells are clustered in the left-upper corner, while Oligodendrocyte cells are grouped more in the right half of the picture and Astrocyte in the middle. There is an issue of misclassification of unknown cells (marked with the star "*"). This problem could be solved in a couple of ways. The first one is to change the size of the grid to see how it affects the distribution. Figure 4.14 shows how mapping with grid size 20 by 16 with quantization error 4.1. The rest of the training parameters were the same for all tests. Then the training parameters could be changed, to achieve more precise assignments. As one can see the clustering of Oligodendrocyte cells and Astrocyte cells changed.

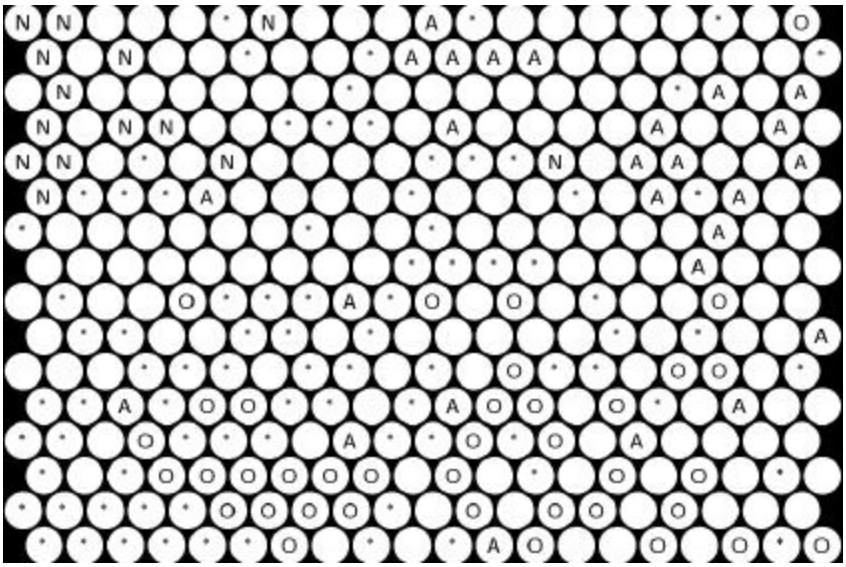


Figure 4.14 SOM with size 20 by 16 using 5 features

Also new features could be added in order to have better separation. Figure 4.15 shows mapping calculated only with 3 parameters: area, distinctness and entropy. Quantization error is 5.6. Figure 4.16 shows mapping calculated using another three parameters: area, roundness and maximum radius. Quantization error is 6.46.

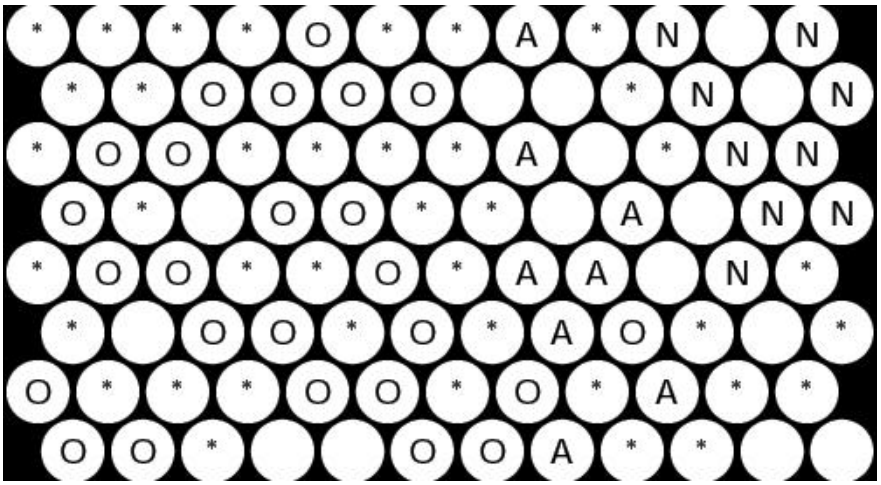


Figure 4.15 SOM with size 12 by 8 using area, distinctness and entropy as features

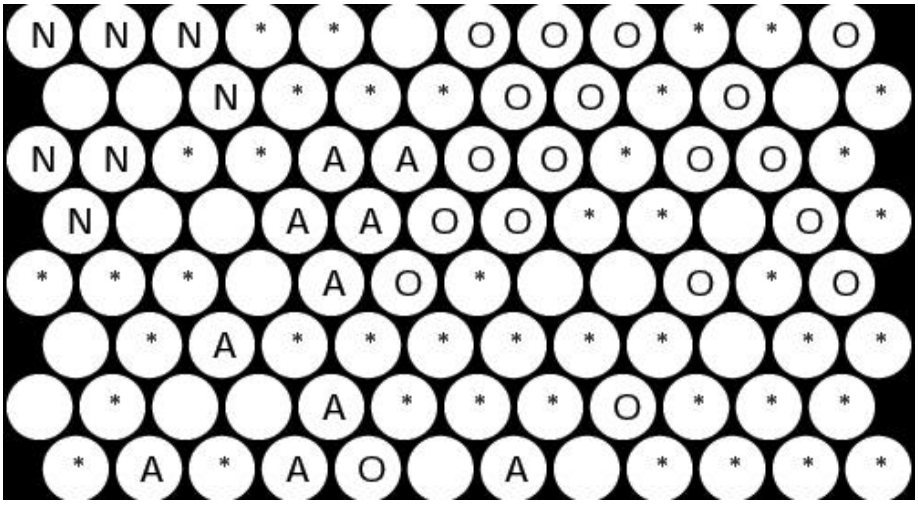


Figure 4.16 SOM with size 12 by 8 using area, roundness and maximum radius as features

Another way to achieve better separation is to have a larger database, which would give a better statistical knowledge. But the amount of data was too limited to perform an extensive research and build a reliable database.

The main goal was to provide a tool for feature research. Now such research can be performed to find the parameters of training, so the approximation would give reliable result and class separation.

CAHPTER 5

COUNTING OF BACTERIA

This chapter describes an application created for the UNH Microbiology Department. During the development of the interface library and AMORS, I tried to find a different task to test AMORS. I made a contact with assistant professor Elise Sullivan from the Microbiology Department at UNH. She defined a problem, which is discussed in this chapter. Appendix D contains the detailed information about the plug-ins discussed in this chapter.

5.1 Problem statement

Microbiology research often requires data about how many bacteria are in a given sample. Many times a researcher has to count bacteria by visual inspection of the specimen using a microscope and counting their number in different areas within the sample. It would be a big help for this to be done automatically.

There are different situations in which counting is required. The main thing to keep in mind is that there could be not only bacteria in the specimen, but also some clutter like pieces of dust and dirt, or even different kind of bacteria which should not be mixed with the kind being counted. Because of counting can be very subjective. So let us consider all these factors as noise, which is interfering with the signal (bacteria). There is not much noise in the specimens where bacteria have been grown in an artificial environment (pure cultures), but some noise is introduced by the air bubbles in the water, for instance.

Figure 5.1 shows an example of the sample taken in an industrial environment with very high noise in it. Figure 5.2 shows a picture of a pure culture.

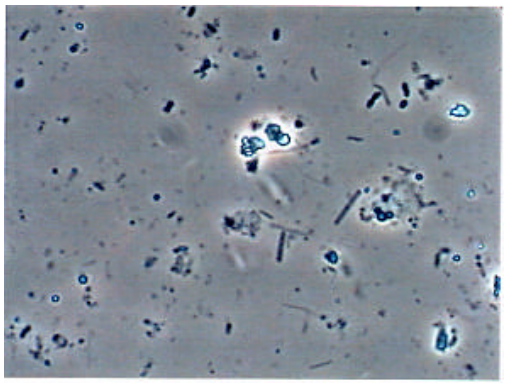


Figure 5.1 Industrial example

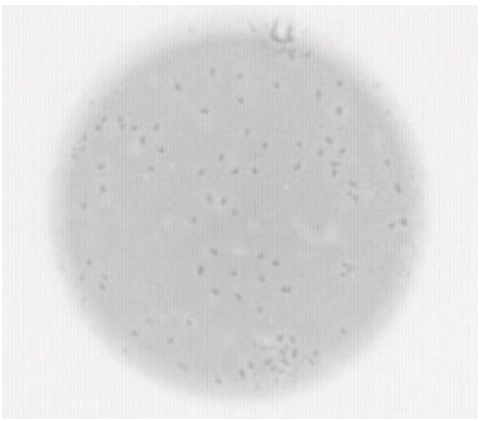


Figure 5.2 Pure culture (inverted)

Because of the different shapes of the bacteria and different levels of the noise in the specimen, different approaches can be used to solve the problem. In my case the primary goal was to calculate the number of bacteria in the sample with a pure culture using tools provided by AMORS.

5.2 Previous work

Even though some vendors of digital cameras for microscopes provide software packages to simplify working with images, I did not find a tool, which would perform automatic counting. Instead, one can be offered an opportunity to make manual counting by clicking on objects of interest and marking them, and after that the total count of markings is available (www.zeiss.com).

5.3 Acquisition and displaying

The translation stage and the digital camera was not available at the time of the experiments. To test the program a special acquisition module was designed, which read the image files in TIFF format. Experiment data images (Figure 5.2) were captured using the hardware board MaxPci. The input to this board was an RS-170A analog composite video signal from a standard RS-170A camera. The camera was mounted on an Olympus BH-2 microscope and the Datacube Max Vision digital image processing system was used for acquisition. The Max Vision system is equipped with an 8bit analog-digital converter producing 256 intensity levels. An Olympus A100PL 1.30oil objective (type of objective that uses oil as an immersion medium) was used. The camera was placed in the monocular path of the microscope using a

special eyepiece with a magnification of 0.3. Images were saved in BMP format by the capturing software. BMP images were converted into TIFF format using Photoshop [13].

A number of images were taken to perform the experiment and saved in TIFF format, so during the experiments the user was asked for the file to use as an input.

Later on a digital camera with resolution 1224x946x24bits was used to acquire images like the one in Figure 5.3

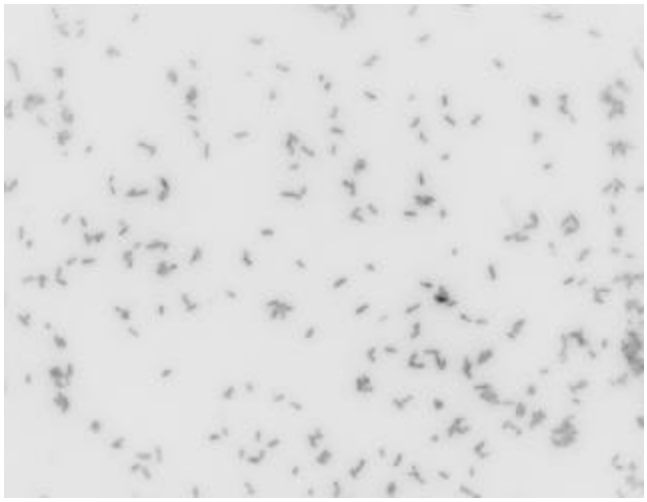


Figure 5.3 Different kind of bacteria in the pure culture (inverted)

There was no need to design an acquisition plug-in, which would work directly with the MaxPci hardware, because it was used only for the experiment. And the camera used to capture the data cannot be accessed by applications other than the one provided with the camera. It was the policy of the company Zeiss. So appropriate programs were used to make TIFF images and the FromFileAcq plug-in was used to load them. Displaying was done using the SimpleDisplay plug-in (see Appendix D for more information).

5.4 Methods

Bacteria should be located in the image, in order to count them. To locate bacteria, the simple thresholding was applied as the first step, so the output was converted into a binary image. For the moment the user should define the value for thresholding, but it is possible to use histograms to find the threshold value automatically. The binary image was supplied as an input to the connectivity algorithm in order to perform blob-analysis. Once blobs were separated their area was calculated and all blobs with an area less then desired were discarded. The information about blobs was submitted to the displaying plug-in. All

procedures were implemented in AMORS interfaces library, so this project utilizes AMORS functionality to simplify implementation.

The following plug-ins were used for bacteria counting, even though they were implemented for testing of the cell recognition application: Area, Simple Rejecter and Simple Display (see appendix D for description of the plug-ins). They are an example of how plug-ins designed for different systems can be utilized without any changes. FromFileAcq, MinLimit, MaxLimit, AreaDivision, MorphologyOp and Blob Separation plug-ins were implemented specially for this application. They are an example of how the system can be extended.

5.5 Results

A number of samples were taken to test the program. The first step was to configure a new script for AMORS. Then the Research Mode was engaged, so the proper settings could be found. Figure 5.4 shows the configuration of the plug-ins used in AMORS to build this application.

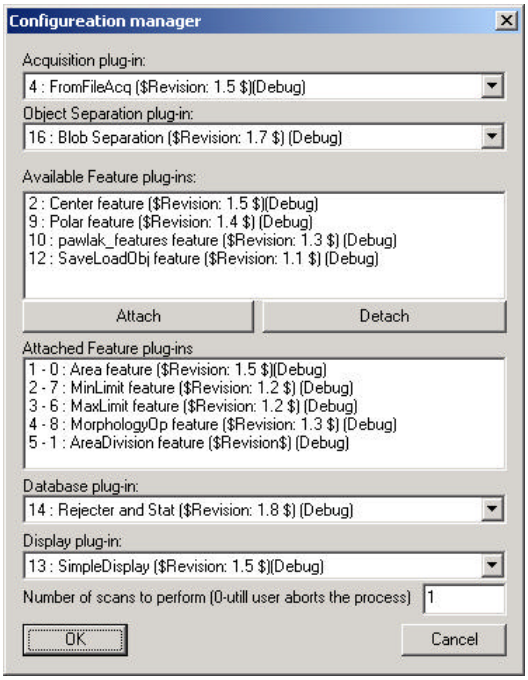


Figure 5.4 The configuration of the plug-ins for AMORS to count bacteria

The system has the following configuration: Acquisition: FromFileAcq plug-in; Object Separation: Blob Separation plug-in; Feature Extraction: Area plug-in, MinLimit plug-in, MaxLimit plug-

in, MorphologyOp plug-in, AreaDivision plug-in; Classification: Simple Rejecter plug-in; Displaying: Simple Display plug-in.

5.5.1 Experiment 1

The original image is shown in Figure 5.5

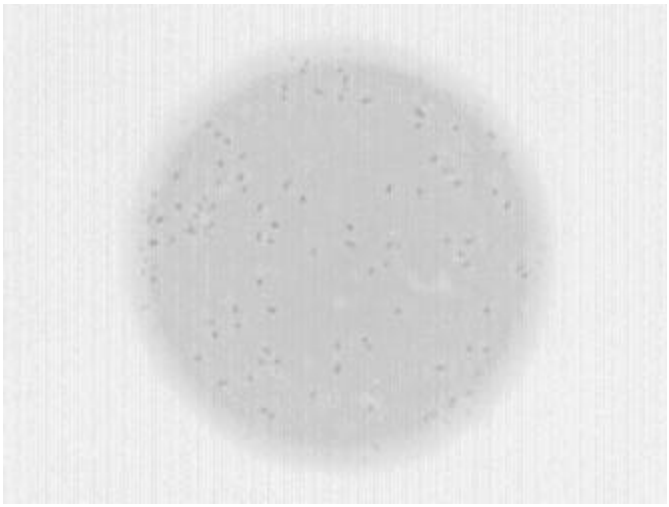


Figure 5.5 Image taken from microscope (inverted)

A square part of the image was extracted and supplied as input to AMORS. The following settings were used: Threshold=55, Minimal area=2;

Figure 5.6 shows the input image and Figure 5.7 shows the result of counting.

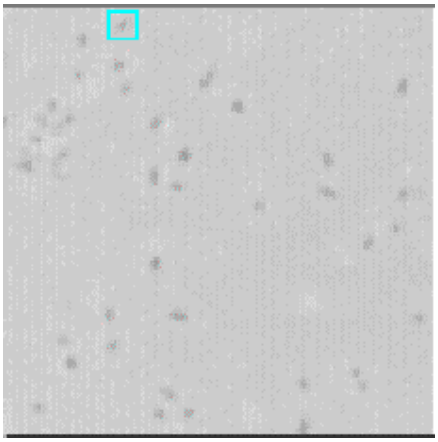


Figure 5.6 Input image (inverted)

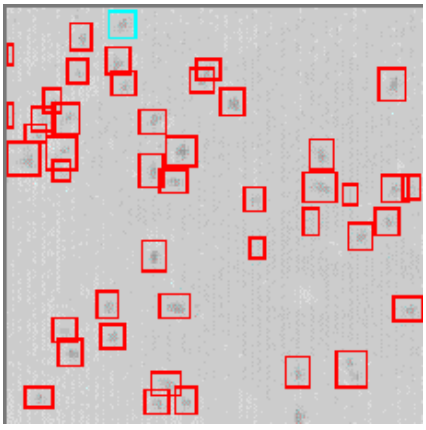


Figure 5.7 Result (inverted)

Rectangles show counted bacteria. The number is 45. It could be noticed that in one case (bottom left) two bacteria were counted as one. So the accuracy of this calculation is: (counted*100%/total number)=45*100/46=97.8%

5.5.2 Experiment 2

Another experiment with a very small amount of bacteria in the image was performed.

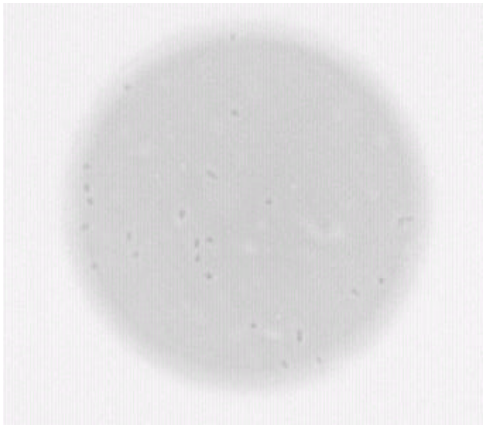


Figure 5.8 Image taken from the microscope (inverted)

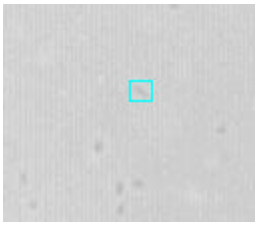


Figure 5.9 Input image (inverted)

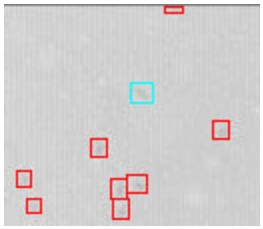


Figure 5.10 Result (inverted)

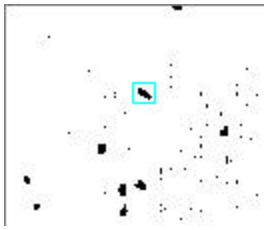


Figure 5.11 Image with applied threshold (inverted)

The number of bacteria found in this experiment is 9 and it is very easy to verify that 100% of the bacteria were calculated. Parameters which were used are: Threshold=50, Minimum area=4.

5.5.3 Experiment 3

In this experiment the goal was to test how the program will react in the case of a very big number of bacteria in the image.

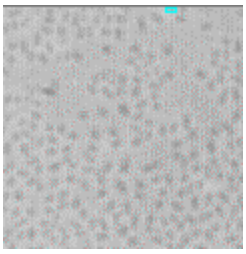


Figure 5.12 Input image (inverted)

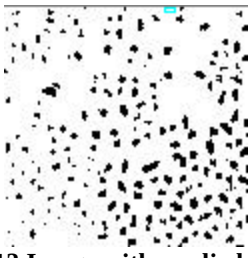


Figure 5.13 Image with applied threshold (inverted)

After adjusting parameters to the following: Threshold=70, Minimum area=1, the result of the counting was quite satisfactory. 219 bacteria were found. From Figure 5.13 one can see that a couple of bacteria were counted as one instead of two, but accuracy is still more then 95%.

5.5.4 Experiment 4

When a new digital camera had arrived, the program was tested with a new high-resolution image data and a different kind of bacteria shown in Figure 5.3 and Figure 5.14. After analyzing it was clear that clustering of bacteria was too high to count them properly. A new feature plug-in was added into the system. This plug-in looked for the blobs with big areas and divided them using average area for one bacteria. In this way the user can set up the system appropriately to count bacteria properly.

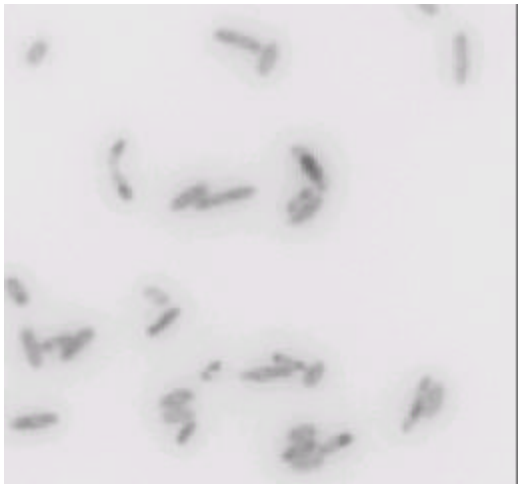


Figure 5.14 Input image (inverted)

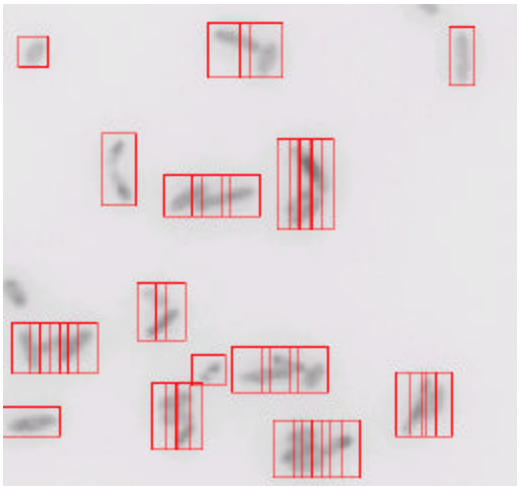


Figure 5.15 Result (inverted)

It can be seen that in some cases two bacteria were split into three parts, and three bacteria were split into two parts. Also some adjusting of the average area could be done, the high-resolution of the input data provides with the large number of bacteria where both combinations are equally probable. And because only the total number of bacteria is important, the overall result has 95% of accuracy.

5.6 Conclusions

Professor Elise Sullivan from Microbiology Department, provided samples of bacteria for testing. She was satisfied with the shown results. The department purchased a digital camera (Experiment 4), so the designed system can be easily used to speed up counting. The accuracy of counting is the same as for

human counting, but the speed is much higher. Tests were done using a Pentium 2, 800 MHz with Windows 2000. Input images had a resolution of 1224 by 946 pixels with 8-bit color depth, average number of counted cells per image was 400 bacteria. Analyzing took around 1 minute (58 seconds). For a human it would take more than 5 minutes.

Equipped with this system the amount of processed views for each sample can be dramatically increased, which improves the accuracy of the result for each sample. Counting of bacteria in a pure culture is useful, but a lot of work should be done with industrial samples. The next step would be to design methods to find and recognize different bacteria in industrial samples.

CHAPTER 6

DISCUSSION

Time is money! This concept dominates in the current business and technology. This is a reason why automatic recognition systems are in demand today. An automatic recognition system can be used to improve the quality or speed of the process. In addition, such a system can be used in areas where a human cannot be present to supervise the process or their presence could put the person in danger (nuclear reactor or the surface of Mars, for instance). The conclusion is that an automatic recognition system may save health, time and money for people. These are well known facts and a lot of research is done in the area of the automatic recognition. Companies were created to answer the increasing demand for such systems. Such companies create some specific systems for specific tasks. There are special software packages, which are used during the creation of the automatic recognition systems, in order to simulate and test them. Many commercial companies will not disclose their research to allow uncontrolled use by other people. They will do it for money. But if we take a look into the research done at universities, we will see a completely different picture. Most of the work is open for use by other universities. Of course, it is not easy to use someone else's research, without the complete understanding of what is done and how it is done. This happens even in case when the same software is used for the same research topics. During the research students are trying to get the result as fast as they can, they need to see if this method works better than the other. Once a result is achieved, students show it in their reports. Modules or scripts that they created to obtain the result are often not well documented or even sometimes not available at all. When someone else wants to repeat the result given in the paper, one often has to rewrite these scripts, methods or algorithms on their own. This thesis is an attempt to solve this problem.

I decided to create a library, which would simplify the creation of automatic recognition systems. The goal was to have an extendable system, which could be used not only for the research used as

examples, but also would simplify the transaction from the research stage to the end-user application. I took the research on cell recognition and made the ready-to-use application. I developed this application keeping in mind the idea of some generalization. I wanted to develop a very flexible and extendable application. During the development I picked another application (bacteria counting). The comparison of methods used for both applications showed the similar parts between applications. This provided me with the idea of a standard way for creating automatic recognition systems. The concept of plug-ins allows the designed system to be easily extendable.

The result of this thesis is the interface library described in the Appendices. This library was used to create two applications for cell recognition and bacteria counting. These applications demonstrate that the proposed method of building a recognition system can be successfully used in practice. This ready to use, documented and free software allows researchers to develop their own systems and reuse plug-ins developed by other people. The separation of the whole system to the smaller modules was done. This gives an opportunity to work on one method at a time and allows reusing of this method later on in a different project. The standard way of connecting the blocks allows such an approach to be used.

The designed library is at a stage from which one can create modifications and improvements. It is ready to use and can be used immediately without modifications. Methods, which require feedback between modules, can also be implemented using the described approach. This is very important for adaptive automatic recognition methods. The next step would be to show how the proposed philosophy works with recognition tasks in other areas, such as character recognition. The interface library provides enough flexibility to build a recognition system for any purpose. AMORS and the currently implemented plug-ins allow researchers to use the system immediately, and improve its performance by adding additional processing using newly designed plug-ins.

It is envisioned that this library will be used at more than one university. A constraint on the source code should be put, in order to maintain compatibility of the code. This thesis can become an open source project, which would help the science community by providing a standard. People in different universities could exchange the plug-ins and they will be compatible and ready to use.

The future improvement to be done is to implement a method to describe a non-linear connection of the plug-ins, when based on the result of one plug-in, the order of the other plug-ins would be changed according to some rules. The system configuration can then be saved and reused later.

Now the library (CD is attached, sources and documentation are also available for downloading from the internet http://sjcomp.virtualave.net/thesis/ or http://www.ece.unh.edu/svpal/) is available for use by the community. It is free to use. The source code is available and documented. Examples of use are given. It can be used on different operating systems, which have a C++ compiler. It is the author's hope that the methods and platform provided would be adopted by others.

APPENDIX A

INTERFACE LIBRARY DESCRIPTION

This appendix describes the details of the interface library. Classes and their members are listed.

A.1 Interfaces.lib library

Goal of this section is to introduce the concept and the organization of the interface library designed the plug-ins. Library is a statically linked library. All plug-ins are using types and objects defined in the interface library. Though this library was implemented and tested under Windows 2000, it was designed using Standard Template Library (StlPort - http://www.stlport.org), which means that it is easy to port to any other platform for which Standard Template Library exists.

Interface library requires you to mention following files in your project:

1. Interfaces.lib – library file;
2. Interface_def.h – header file with definitions of the objects;

A.2 Interface objects and functions

Interfaces_def.h includes definitions for the following header files: AllConstants.h, SJIniFile.h, SJBuffer.h, SJImage.h, SJImageList.h, SJObjectInfo.h, SJObjectInfoList.h, SJSpecimen.h, MainLoop.h, Processing.h, Utils.h;

AllConstants.h - constants and enumerator values returned or required by functions and objects.

SJIniFile.h - class CSJIniFile used to store and retrieve configuration information about working scenario and parameters;

SJBuffer.h - class CSJBuffer used to store a raw image data acquired from the camera or results of processing routines;

SJImage.h - class CSJImage stores the image data and parameters required for the processing; class CSJRGBAColor - describes the color of the pixel;

SJImageList.h - class CSJImageList stores a list of images, so more then one image can be used in the processing;

SJObjectInfo.h - class CSJObjectInfo stores information about the object provided by the plug-ins; class CSJRect describes a rectangular area;

SJObjectInfoList.h - class CSJObjectInfoList - stores information about a set of objects;

SJSpecimen.h - class CSJSpecimen - stores all information about the current processing, from the image data to the variables some plug-ins would like to send to each other;

MainLoop.h - contains routines of main and research loop;

Processing.h - contains some image-processing algorithms;

Utils.h - contains some utility functions.

A.3 Classes description

This section contains total description of the classes and their class members.

A.3.1 CSJBuffer

class CSJBuffer
#include “SJBuffer.h”

Description	Manages storage of one color plane of the image data in the memory. Also provides functions to manipulate with this data
Usage	Create(…) function should be called in order to initialize the variables of this class with the given parameters. Create(…) handles allocating and cleaning of the memory for you

Class member-functions:

CSJBuffer() – Default constructor

Synopsis	CSJBuffer()
Input parameters	None
Return value	None
Description	Default constructor for the class, initializes all member variables
Usage	Called automatically

CSJBuffer(const CSJBuffer&) – Copy constructor

Synopsis	CSJBuffer(const CSJBuffer&)
Input parameters	CSJBuffer& class of original buffer
Return value	None
Description	Copy constructor for the class, initializes all member variables as they are in the input parameter, memory is copied, so object is a copy of the given one.

Usage	Use to initialize an object with already existing one.

Create(int, int, int) – Creating buffer

Synopsis	bool Create(int iWidth, int iHeight, int iBytesPerPixel=ONE_BYTE)
Input parameters	iWidth – width of the buffer in pixels iHeight – height of the buffer in pixels iBytesPerPixel – number of bytes per pixel; supported values are ONE_BYTE (for the 8 bit color) and TWO_BYTES (for up to 16 bit color)
Return value	true – if allocation of the memory was succesfull false – otherwise
Description	Releases allocated memory (if any), sets member variables to the given values, allocates memory for the buffer, clears the momory
Usage	Call this function each time you want to reset the buffer contents

~CSJBuffer() – class destructor

Synopsis	~CSJBuffer()
Input parameters	None
Return value	None
Description	Releases allocated memory
Usage	Is called automatically when object gets released

&operator=(const CSJBuffer&) – assignment operator

Synopsis	CSJBuffer &operator=(const CSJBuffer&)
Input parameters	CSJBuffer& original class to be equal to
Return value	Returns own address
Description	Used to copy parameters and contents of another buffer
Usage	Use when you have to created objects of the class and want them to be the same

Copy(CSJBuffer, bool, int, int, int, int) – creates object with the part of the original object

Synopsis	void Copy(CSJBuffer& bufobj, bool yKeepOriginalBPP=false, int iLeft=0, int iTop=0,int iRight=0, int iBottom=0);
Input parameters	bufobj – data will be copied from this object yKeepOriginalBPP – allows not to have the same value of bytes per pixel, default is false, so bytes per pixel value will be taken from bufobj iLeft, iTop, iRight, iBottom – boundaries of the region to be copied from the bufobj. If all are 0s all memory is copied
Return value	None
Description	Creates buffer with the data from the region of another object
Usage	Use when you want to copy the region of the buffer.

CopyIntoPos(CSJBuffer&, int, int) – copies another buffer into certain position

Synopsis	void CopyIntoPos(CSJBuffer& bufobj, int iLeft=0, int iRight=0)
Input parameters	Bufobj – buffer to get data from iLeft, iRight – position at which insert the data
Return value	None
Description	Copies the data to the given position, the whole input buffer is inserted if there is enough memory allocated
Usage	Use when you want to overwrite some region of the buffer with the data from another buffer

GetHeight() – height of the buffer

Synopsis	int GetHeight()
Input parameters	None
Return value	The height of the buffer in pixels
Description	Returns the height of the buffer
Usage	Use to request the height of the buffer

GetWidth() – width of the buffer

Synopsis	Int GetWidth()
Input parameters	None
Return value	The width of the buffer in pixels
Description	Returns the width of the buffer
Usage	Use to request the width of the buffer

GetBufferSize() – size of the memory used for the memory (bytes)

Synopsis	int GetBufferSize()
Input parameters	None
Return value	Returns the number of bytes allocated by the current object
Description	Function returns the number of bytes allocated by the current object in the memory for the data by multiplying width, height and the number of bytes per pixel
Usage	Use to request the number of bytes allocated in the memory for the current buffer

ClearMemory() – filling memory with 0s

Synopsis	void ClearMemory()
Input parameters	None
Return value	None
Description	Sets all bits in the allocated memory to 0
Usage	Use to reset the buffer to all 0s

FillBufferFrom(byte*,int) – filling buffer from the data in the memory

Synopsis	Bool FillBufferFrom(byte* pbSrcBuf, int iLength)
Input parameters	pbSrcBuf – pointer to the first byte in the memory to begin copying from iLength – number of bytes to copy
Return value	true – if successful (enough space to copy memory) false – otherwise
Description	Fast copying of the memory bytes from the given source to the buffer
Usage	Use it if you have some data already in the memory and you want to use it. No parameter (width or height) is changed.

FillBufferFrom(CSJBuffer&) –copying the data from one buffer to another

Synopsis	bool FillBufferFrom(CSJBuffer& bufobj)
Input parameters	bufobj – source of the data
Return value	true – if copying is successful (enough space and so on) false – otherwise
Description	Fast copying of the data from one buffer to another, without change in the parameters of the buffer
Usage	Use it when the size of the buffers is the same (number of allocated bytes in the memory is the same), but you want to change the formatting. Was buffer 5x3, you want 3x4

GetBytesPP() – bytes per pixel

Synopsis	int GetBytesPP()
Input parameters	None
Return value	Returns number of bytes used to describe one pixel (color resolution)
Description	Function used to return number of bytes used to describe one pixel, possible values are: ONE_BYTE=1, TWO_BYTES=2
Usage	Use to find out how many bytes are used to describe the pixel

GetByteBuffer() – byte buffer address

Synopsis	byte* GetByteBuffer()
Input parameters	None

Return value	Returns pointer to the first byte in the allocated memory
Description	Returns pointer to the first byte in the allocated memory with the data
Usage	Use to get the address of the first byte of the data to manipulate with the data directly

GetWordBuffer() – word buffer address

Synopsis	WORD* GetWordBuffer()
Input parameters	None
Return value	Returns pointer to the first word in the allocated memory
Description	Returns pointer to the first word in the allocated memory with the data
Usage	Use to get the address of the first word of the data to manipulate with the data directly

GetWordValue(int, int) – word value in the buffer

Synopsis	WORD GetWordValue(int iX, int iY)
Input parameters	iX – offset for X iY – offset for Y
Return value	Returns the value of the word in the buffer with the given coordinates
Description	Returns the value of the word in the buffer with the given coordinates
Usage	Use it when you want to get the word value for a particular pixel (be careful no checking on ranges is performed and if you are using one byte per pixel exception could be called)

GetByteValue(int,int) – byte value in the buffer

Synopsis	Byte GetByteValue(int iX, intiY)
Input parameters	iX – offset for X iY – offset for Y
Return value	Returns the value of the byte in the buffer with the given coordinates
Description	Returns the value of the byte in the buffer with the given coordinates
Usage	Use it when you want to get the byte value for a particular pixel

GetValue(int, int) – value in the buffer

Synopsis	WORD GetValue(int iX, intY)
Input parameters	iX – offset for X iY – offset for Y
Return value	Returns the value of in the buffer with the given coordinates
Description	Returns the value of in the buffer with the given coordinates, warps GetByteValue and GetWordValue, picking the correct function depending on the bpp in the buffer
Usage	Use it instead of GetByteValue or GetWordValue for more safty

SetWordValue(int,int,WORD) – sets word value of the pixel

Synopsis	void SetWordValue(int iX, int iY, WORD wValue)
Input parameters	iX – offset for X iY – offset for Y wValue – value to set pixel to
Return value	None
Description	Sets the pixel with the coordinates iX, iY to the wValue
Usage	Use it to set the word value in the buffer

SetByteValue(int,int,byte) – sets byte value of the pixel

Synopsis	void SetByteValue(int iX, int iY, byte b Value)
Input parameters	iX – offset for X iY – offset for Y bValue – value to set pixel to
Return value	None

Description	Sets the pixel with the coordinates iX, iY to the bValue
Usage	Use it to set the byte value in the buffer

SetValue(int,int,WORD) – sets value of the pixel

Synopsis	void SetValue(int iX, int iY, WORD wValue)
Input parameters	iX – offset for X iY – offset for Y wValue – value to set pixel to
Return value	None
Description	Sets the pixel with the coordinates iX, iY to the wValue
Usage	Use it to set a value in the buffer, the appropriate conversion is called, so if there is only one byte per pixel wValue will be converted to byte type

void SetDirectValue(int,WORD) – sets value with a given offset in the buffer

Synopsis	void SetDirectValue(int iP, WORD wValue)
Input parameters	iP – offset in the buffer wValue – value to set pixel too
Return value	None
Description	Sets the value in the buffer with the given offset iP to wValue (buffer is interpreted as a vector)
Usage	If only one byte per pixel is used wValue is converted to the byte type

GetDirectValue(int) – value in the buffer at the given offset

Synopsis	WORD GetDirectValue(int iP)
Input parameters	iP – offset in the buffer
Return value	Returns the value in the buffer with the given offset
Description	Returns the value in the buffer with the given offset (buffer is interpreted as a vector)
Usage	Retrieves the pixel from the memory and returns it as a WORD type

Invert() – inverting values

Synopsis	void Invert()
Input parameters	None
Return value	None
Description	Inverts the data in the buffer
Usage	Call to invert the data (using ClearMemory() and then Invert() makes all pixels to be at the maximum value)

Mean2() – mean of the pixel values

Synopsis	double Mean2()
Input parameters	None
Return value	Returns the mean value for the pixels in the buffer
Description	Returns the mean value for the pixels in the buffer (sum of all pixels divided by the number of pixels)
Usage	Call to get the mean

Std2() – standard deviation

Synopsis	double Std2()
Input parameters	None
Return value	Returns the standard deviation of the pixels in the buffer
Description	Returns the standard deviation of the pixels in the buffer
Usage	Call to get the standard deviation

Min() – minimum value of the pixel

Synopsis	WORD Min()
Input parameters	None

Return value	Returns the minimum value of the pixel in the buffer
Description	Finds the minimum value of the pixel in the buffer, if one byte per pixel is used the value is cast to the WORD type
Usage	Call to get the minimum value

Max() – maximum value of the pixel

Synopsis	WORD Max()
Input parameters	None
Return value	Returns the maximum value of the pixel in the buffer
Description	Finds the minimum value of the pixel in the buffer, if one byte per pixel is used the value is cast to the WORD type
Usage	Call to get the maximum value

Crop(int,int,int,int) – crops data in the buffer

Synopsis	void Crop(int iLeft, int iTop, int iRight=0, int iBottom=0)
Input parameters	iLeft, iTop, iRight, iBottom – region to be saved in the buffer, all outside data will be lost
Return value	None
Description	Crops the data in the buffer to the given area
Usage	If iRight and iBottom are set to 0, then area with top left point (0,0) and bottom right point(iLeft, iTop) is used

A.3.2 CSJRGBAColor

class CSJRGBAColor
#include “SJImage.h”

Description	Describes color of the image
Usage	Used to retrieve and set pixel colors in CSJImage class

Class variables:

WORD r – red value for the pixel; WORD g – green value for the pixel; WORD b – blue value for the pixel; WORD a – additional channel (can be used as a transparency)

Class member-functions:

CSJRGBAColor() – default constructor

Synopsis	void CSJRGBAColor()
Input parameters	None
Return value	None
Description	Initializes all member variables with 0
Usage	Called automatically

CSJRGBAColor(WORD) – constructor

Synopsis	void CSJRGBAColor(WORD w)
Input parameters	w – value for all color components accept member variable “a”
Return value	None
Description	Initializes all member variables with “w”
Usage	Can be used to give a grayscale value to the color on creation of the object

CSJRGBAColor(WORD,WORD,WORD,WORD) – constructor

Synopsis	void CSJRGBAColor(WORD wr,WORD wg,WORD wb,WORD wa)

Input parameters	wr, wg, wb, wa – values to initialize member variables to
Return value	None
Description	Initializes member variables with the given values
Usage	Can be used to give a color value on ??? creation of the object

~CSJRGBAColor() – destructor

Synopsis	void ~CSJRGBAColor()
Input parameters	None
Return value	None
Description	Standard destructor
Usage	Called automatically

Clear() – reset color

Synopsis	void Clear()
Input parameters	None
Return value	None
Description	Sets all member variables to 0
Usage	Sets all member variables to 0

Set(WORD,WORD,WORD,WORD) – set color

Synopsis	void Set(WORD wr,WORD wg,WORD wb,WORD wa)
Input parameters	wr, wg, wb, wa – values to set member variables to
Return value	None
Description	Sets the color of the pixel
Usage	Sets the color of the pixel

SetSame(WORD) – set grayscale color

Synopsis	void SetSame(WORD w)
Input parameters	w – value to set color channels to
Return value	None
Description	Function sets r,g and b variables of the class to w
Usage	Use to set a grayscale color

Average(CSJRGBAColor) – averages color with a given color

Synopsis	void Average(CSJRGBAColor n)
Input parameters	n – color to make an average with
Return value	None
Description	Sums each variable of the class with a corresponding value of n and devides by 2
Usage	Call to find what would be an average color for the current and the given color

Average(double) – averages color with a given value

Synopsis	void Average(double n)
Input parameters	n – value to find an average with
Return value	None
Description	Finds an average for each member variable of the class and the given value n
Usage	Class is summing member variable with the n and divides it by 2

GetGray() – grayscale value

Synopsis	WORD GetGray()
Input parameters	None
Return value	Returns the grayscale value of the color
Description	Sums all color channels and divides them by 2
Usage	Call to get a grayscale value

operator+(const CSJRGBAColor&)

Synopsis	CSJRGBAColor& operator+(const CSJRGBAColor& obj)

Input parameters	obj – color to sum with
Return value	Returns own address
Description	Allows to sum two colors together
Usage	Each component is summed with checking for the limits (so color component can not go above 0xFFFF)

operator-(const CSJRGBAColor&)

Synopsis	CSJRGBAColor& operator-(const CSJRGBAColor& obj)
Input parameters	obj – color to be subtracted
Return value	Returns own address
Description	Allows to subtract one color from another
Usage	Components get subtracted with checking for the limits (so color component can not go below 0)

operator/(const CSJRGBAColor&)

Synopsis	CSJRGBAColor& operator/(const CSJRGBAColor& obj)
Input parameters	obj – color to divide on
Return value	Returns own address
Description	Allows to dived one color by another
Usage	Each component is divided with checking for the limits (so color component can not go below 0)

operator/(const int&)

Synopsis	CSJRGBAColor& operator/(const int i)
Input parameters	i – value to divide on
Return value	Returns own address
Description	Allows to dived color by the value
Usage	Each component is divided with checking for the limits (so color component can not go below 0)

operator<(const CSJRGBAColor&)

Synopsis	bool operator<(const CSJRGBAColor& obj)
Input parameters	obj – color to compare with
Return value	Own address
Description	Compares grayscale value of the colors
Usage	Use to compare two colors

A.3.3 CSJImage

class CSJImage
#include “SJImage.h”

Description	Describes 2d image data with the color depth up to 32 bits (four color channels)
Usage	Used to handle image data in the system. Create(…) function should be called in order to initialize the variable of this class with given parameters. Create(…) handles allocating and cleaning of the memory for you.

Class member functions:

CSJImage() – default constructor

Synopsis	CSJImage()
Input parameters	None
Return value	None
Description	Default constructor, which initializes all variables in the class
Usage	Called automatically

CSJImage(const CSJImage&) – copy constructor

Synopsis	CSJImage(const CSJImage& original)
Input parameters	original – object to initialize with
Return value	None
Description	Initializes the object with the given object
Usage	Use to create a copy of another object

&operator=(const CSJImage&) – assignment operator

Synopsis	CSJImage &operator=(const CSJImage &original)
Input parameters	original – object to get value from
Return value	Own address
Description	Makes the current object to be equal to the given one
Usage	Use to make a copy of the object

~CSJImage() – destructor

Synopsis	virtual ~CSJImage()
Input parameters	None
Return value	None
Description	Takes care of cleaning the memory
Usage	Called automatically

Create(int,int,int,int,float) – creates an image

Synopsis	bool Create(int iWidth, int iHeight, int iColorDepth=COLOR_DEPTH_8BITS, int iNumber=1, float fParameter=1.0f)
Input parameters	iWidth, iHeight – size of the image iColorDepth – color depth, effects bytes per pixel for each channel buffer iNumber – number of the image (for keeping track of the images) fParameter – one can use this value to have an additional label for the image
Return value	true – if creation is successful false – otherwise
Description	Allocates memory for the color channels
Usage	Call it to create or recreate the image

CreateFromFile(const char*,int,int,float) – creating an image from the file

Synopsis	bool CreateFromFile(const char* cpFileName, int iColorDepth=COLOR_DEPTH_8BITS, int iNumber=1, float fParameter=1.0f)
Input parameters	CpFileName – filename to the file on hard disk drive iColorDepth – color depth, effects bytes per pixel for each channel buffer iNumber – number of the image (for keeping track of the images) fParameter – one can use this value to have an additional label for the image
Return value	true – if creation is successful false – otherwise
Description	Allocates memory and loads information from the file (TIFF file format is supported)
Usage	Call it to create the image and load the data from the file

CreateFrom(CSJImage*,int,int,int,int) – creates image from another image

Synopsis	int CreateFrom(CSJImage* Im, int iLeft=0, int iTop=0, int iRight=0, int iBottom=0)
Input parameters	Im – source image iLeft,iToop,iRight,iBottom – rectangle to copy image from
Return value	1 – in the case of success 0 – in the case of error
Description	Creates the image with the data from the given rectangle at Im
Usage	Use it if you want to create an image from the part of another image

GetColorDepth() – color depth for the image

Synopsis	int GetColorDepth()
Input parameters	None
Return value	Returns color depth for the image
Description	Returns color depth for the image, possible values are: COLOR_DEPTH_8BITS=8, COLOR_DEPTH_16BITS=16, COLOR_DEPTH_24BITS=24, COLOR_DEPTH_32BITS=32, COLOR_DEPTH_16BITS_GRAYSCALE=14
Usage	Use it to find out what the color depth is

GetHeight() – image height

Synopsis	int GetHeight()
Input parameters	None
Return value	Returns the height of the image
Description	Returns the height of the image
Usage	Use to find what the height of the image is

GetWidth() – image width

Synopsis	int GetWidth()
Input parameters	None
Return value	Returns the width of the image
Description	Returns the width of the image
Usage	Use to find what the width of the image is

ClearImage() – clears image

Synopsis	void ClearImage()
Input parameters	None
Return value	None
Description	Sets all pixels in the image to 0,0,0 RGB value
Usage	Use it to have a completely black image

GetGrayScaleImage() – grayscale image

Synopsis	CSJBuffer* GetGrayScaleImage()
Input parameters	None
Return value	Returns pointer to the buffer with the grayscale information
Description	Allows to have the access to the buffer with the grayscale values of the image (right now only one channel is used)
Usage	Use it if you need to get some information or if you want to modify the data of the image

GetRedChannel() – red channel

Synopsis	CSJBuffer* GetRedChannel()
Input parameters	None
Return value	Returns the pointer to the red channel of the image
Description	Allows to have the access to the buffer with the red component of the color
Usage	Use it if you need get some information or if you want to modify the data of the image

GetGreenChannel() – green channel

Synopsis	CSJBuffer* GetGreenChannel()
Input parameters	None
Return value	Returns the pointer to the green channel of the image
Description	Allows to have the access to the buffer with the green component of the color
Usage	Use it if you need get some information or if you want to modify the data of the image

GetBlueChannel() – blue channel

Synopsis	CSJBuffer* GetBlueChannel()
Input parameters	None
Return value	Returns the pointer to the blue channel of the image
Description	Allows to have the access to the buffer with the green component of the color
Usage	Use it if you need get some information or if you want to modify the data of the image

GetAlphaChannel() – alpha channel

Synopsis	CSJBuffer* GetAlphaChannel()
Input parameters	None
Return value	Returns the pointer to the alpha channel of the image
Description	Allows to have the access to the buffer with the alpha component of the color
Usage	Use it if you need get some information or if you want to modify the data of the image

GetChannelX() – channel X (one of 4 color channels [1,2,3,4]=[red, green, blue,alpha])

Synopsis	CSJBuffer* GetChannelX()
Input parameters	None
Return value	Returns the pointer to the buffer with the color channel
Description	Returns the pointer to the buffer with the color channel
Usage	It is better to use GetXXXChannel functions instead (where XXX one of Red, Green, Blue, Alpha)

GetColor(int,int) – get the pixel color

Synopsis	CSJRGBAColor GetColor(int iX,int iY)
Input parameters	iX, iY – coordinates of the pixel
Return value	Returns the color of the pixel
Description	Returns the color of the pixel
Usage	Use to get the color of the pixel

SetColor(int,int) – set pixel color

Synopsis	void SetColor(int iX,int iY, CSJRGBAColor pixel)
Input parameters	iX, iY – coordinates of the pixel pixel – new color
Return value	None
Description	Sets the pixel color at coordinates (iX, iY)
Usage	Use to the color of the pixel

MaxPossibleColor() – max possible color in the image

Synopsis	CSJRGBAColor MaxPossibleColor()
Input parameters	None
Return value	Returns the color value
Description	Returns the maximum color value for the image depending on the image color depth
Usage	Use it to find what is the maximum color, which possibly can be stored in the image

MinPossibleColor() – min possible color in the image

Synopsis	CSJRGBAColor MinPossibleColor()
Input parameters	None
Return value	Returns the color value
Description	Returns the minimum color value for the image depending on the image color depth
Usage	Use it to find what is the minimum color , which possibly can be stored in the

image

MaxColor(), MaxColor(int, int, int, int) – maximum color in the image

Synopsis	CSJRGBAColor MaxColor(int iL=0,int iT=0,int iR=0, int iB=0)
Input parameters	iL,iT,iR,iB – left, top, right and bottom limits to search for the value
Return value	Returns the color value
Description	Returns the maximum color value found in the region of the image (if all values are set to 0, all image is searched)
Usage	Use it to find what is the maximum color in the region of the image

MinColor(), MinColor(int, int, int, int) – minimum color in the image

Synopsis	CSJRGBAColor MinColor(int iL=0,int iT=0,int iR=0, int iB=0)
Input parameters	iL,iT,iR,iB – left, top, right and bottom limits to search for the value
Return value	Returns the color value
Description	Returns the minimum color value found in the region of the image (if all values are set to 0, all image is searched)
Usage	Use it to find what is the minimum color in the region of the image

GetNumber() – returns number of the image

Synopsis	int GetNumber()
Input parameters	None
Return value	Returns the number of the image
Description	Returns the number of the image
Usage	Use to find what is the number of the image

GetParameter() – parameter of the image

Synopsis	float GetParameter()
Input parameters	None
Return value	Returns the parameter of the image
Description	Returns the parameter of the image
Usage	Use to find what is the parameter value of the image

GetBytesPerPixelInBuffer() – bytes per pixel in the buffer

Synopsis	int GetBytesPerPixelInBuffer()
Input parameters	None
Return value	Returns the number of bytes per pixel for color channel buffers, it is the same for all of them
Description	Returns the number of bytes per pixel in the buffer, see CSJBuffer::GetBytesPP(). Could be 1 or 2.
Usage	Use to obtain the number of bytes per pixel

Invert() – inverting values

Synopsis	void Invert()
Input parameters	None
Return value	None
Description	Inverts the data in the image
Usage	Call to invert the image

IsEmpty() – check for information in the image

Synopsis	bool IsEmpty()
Input parameters	None
Return value	true – if image size is bigger then 1 by 1 false – otherwise
Description	Shows if the image was created
Usage	Use it for a fast check if image was created

Expand(int,int,CSJRGBAColor) – exapands image width and heigth

Synopsis	void Expand(int iW, int iH,CSJRGBAColor color)
Input parameters	iW,iH – new values for width and height color – additional space will be filled with this color
Return value	None
Description	The image is expanded according to the supplied values
Usage	Use it to make the image bigger

ClipColors(CSJRGBAColor, CSJRGBAColor, CSJRGBAColor)

Synopsis	void ClipColors(CSJRGBAColor minc,CSJRGBAColor maxc,CSJRGBAColor sc)
Input parameters	minc – minimum color maxc – maximum color sc – replacement color
Return value	None
Description	Changes all colors which are less or larger then minc and maxc to the new color sc
Usage	Use it to assign another value to the pixels with color outside of the region

A.3.4 CSJImageList

class CSJImageList
#include “SJImageList.h”

Description	Stores a list of images.
Usage	Used to manage the list of images. Copies of the images are stored in the memory and can be accessed by using GetImage() function. There is no limit on number of images in the list. But memory can be relocated during addition or deleting of a the image, so values returned by GetImage() could be not valid

Class member functions:

CSJImageList() – default constructor

Synopsis	CSJImageList()
Input parameters	None
Return value	None
Description	Default constructor, which initializes all variables in the class
Usage	Called automatically

~CSJImageList() – destructor

Synopsis	virtual ~CSJImageList()
Input parameters	None
Return value	None
Description	Takes care of cleaning the memory
Usage	Called automatically

Add(CSJImage) – adding a new image to the list

Synopsis	void Add(CSJImage newImage)
Input parameters	newImage – image to be added
Return value	None
Description	Adds a new image, by copying it into internal structure.
Usage	Once used, pointers saved from GetImage() could be not valid.

Delete(unsigned int) – deleting an image

Synopsis	void Delete(unsigned int iIndex=0)
Input parameters	iIndex – index of the image to be removed

Return value	None
Description	Deletes the image with a given index if exist one.
Usage	Use it to remove the image from the list

GetLength()

Synopsis	int GetLength()
Input parameters	None
Return value	Returns the number of images added to the list
Description	Returns the number of images added to the list
Usage	Use it to find out what is the valid range of indices for GetImage() or Delete()

GetImage(unsigned int) – get a stored image

Synopsis	CSJImage* GetImage(unsigned int iIndex)
Input parameters	iIndex – index of the image to be returned
Return value	Returns the pointer to the image stored in the memory
Description	Returns the pointer to the image stored in the memory
Usage	Use it to retrieve the image stored in the list

Clear() – clears the list

Synopsis	void Clear()
Input parameters	None
Return value	None
Description	Removes all stored images
Usage	Use it to remove all images from the memory

A.3.5 CSJRect

class CSJRect
#include “SJRect.h”

Description	Coordinates of a rectangle region
Usage	Used to describe the rectangular area (used in CSJObjectInfo class)

Class variables:

int m_iLeft – left boundary
int m_iRight – right boundary
int m_iTop – top boundary
int m_iBottom – bottom boundary

Class member functions:

CSJRect() – default constructor

Synopsis	CSJRect()
Input parameters	None
Return value	None
Description	Default constructor, which initializes all variables in the class to all 0s
Usage	Called automatically

CSJRect(int,int,int,int) – constructor

Synopsis	CSJRect(int iL, int iR, int iT, int iB)
Input parameters	iL – left boundary iR – right boundary iT – top boundary iB – bottom boundary

Return value	None
Description	Initializes all variables with the given values
Usage	Called automatically

~CSJRect() – destructor

Synopsis	virtual ~CSJRect ()
Input parameters	None
Return value	None
Description	Destuctor
Usage	Called automatically

Height() – height of the rectangle

Synopsis	int Height()
Input parameters	None
Return value	Returns the height of the rectangle
Description	Returns the height of the rectangle
Usage	Use to request the height of the rectangle

Width() – width of the rectangle

Synopsis	int Width()
Input parameters	None
Return value	Returns the width of the rectangle
Description	Returns the width of the rectangle
Usage	Use to request the width of the rectangle

Clear() – resetting rectangle

Synopsis	void Clear()
Input parameters	None
Return value	None
Description	Sets all variables to 0
Usage	Use it to reset the rectangle

Area() – area of the rectangle

Synopsis	int Area()
Input parameters	None
Return value	Returns the area of the rectangle
Description	Returns the area of the rectangle, which is multiplication of width and height
Usage	Use it to find the area occupied by the rectangle

PtInRect(int,int) – check if the point is inside of the rectangle

Synopsis	bool PtInRect(int iX,int iY)
Input parameters	iX,iY – coordinates of the point
Return value	true – if point is inside of the rectangle false – otherwise
Description	Checks if the point is inside of the rectangle
Usage	Use it to find if this point belongs to the rectangle area

ShiftBy(int,int) – shifting rectangle

Synopsis	void ShiftBy(int iX, int iY)
Input parameters	iX,iY – offsets for x and y axes
Return value	None
Description	Shifts all points of the rectangle by the given offset
Usage	Use it if you want to shift the rectangle without changing its sise

A.3.6 CSJObjectInfo

class CSJObjectInfo
#include “SJObjectInfo.h”

Description	Keeps information about object
Usage	Used to describe the object parameters and location

Class variables:

CSJRect Rect – position of the object in the image

Class member functions:

CSJObjectInfo () – default constructor

Synopsis	CSJObjectInfo()
Input parameters	None
Return value	None
Description	Default constructor
Usage	Called automatically

CSJRect(int,int,int,int) – constructor

Synopsis	CSJRect(int iL, int iR, int iT, int iB)
Input parameters	iL – left boundary iR – right boundary iT – top boundary iB – bottom boundary
Return value	None
Description	Initializes all variables in the given values
Usage	Called to create an initialized object

~CSJObjectInfo() – destructor

Synopsis	virtual ~ CSJObjectInfo()
Input parameters	None
Return value	None
Description	Destuctor
Usage	Called automatically

SetValue(std::string,int) – set a parameter with an integer value

Synopsis	bool SetValue(std::string strName,int iValue)
Input parameters	strName – name of the parameter iValue – value of the parameter
Return value	true – if successful false – otherwise
Description	Assigns a numerical value of type integer to the given parameter. Function creates or changes the value of the parameter
Usage	Use it when you want to assign a integer value to the parameter

SetValue(std::string,double) – set a parameter with a value of type double

Synopsis	bool SetValue(std::string strName,double dValue)
Input parameters	strName – name of the parameter dValue – value of the parameter
Return value	true – if successful false – otherwise
Description	Assigns a numerical value of type double to the given parameter. Function creates or changes the value of the parameter

Usage	Use it when you want to assign a value of type double to the parameter

SetValue(std::string,std::string) – set a parameter with a string value

Synopsis	bool SetValue(std::string strName, std::string strValue)
Input parameters	strName – name of the parameter strValue – value of the parameter
Return value	true – if successful false – otherwise
Description	Assigns a numerical value of type string to the given parameter. Function creates or changes the value of the parameter
Usage	Use it when you want to assign a string value to the parameter

GetValue(std::string) – get value as a string

Synopsis	std::string GetValue(std::string strVarName)
Input parameters	strVarName – name of the parameter to get value of
Return value	Returns string with a value
Description	Function returns a string value of the parameter
Usage	Use it if you want to get a string value of the parameter even if it was saved as other type, it will be converted to a string

GetValueI(std::string) – get value as an integer

Synopsis	int GetValueI(std::string strVarName)
Input parameters	strVarName – name of the parameter to get value of
Return value	Returns integer value of the parameter or 0 if it is not integer value
Description	Function returns the integer value of the parameter or 0 if it is a string
Usage	Use it if you want to get a value stored as integer or double. If it was double, some information will be lost

GetValueF(std::string) – get value as a double

Synopsis	double GetValueF(std::string strVarName)
Input parameters	strVarName – name of the parameter to get value of
Return value	Returns double value of the parameter or 0 if it is not integer value
Description	Function returns the double value of the parameter or 0 if it is a string
Usage	Use it if you want to get a value stored as integer or double

GetNumVars() – number of variables

Synopsis	int GetNumVars()
Input parameters	None
Return value	Returns number of parameters set for the object
Description	Returns number of parameters set for the object
Usage	Use to find out how many parameters were saved in this object

DeleteValue(std::string) – deletes value

Synopsis	void DeleteValue(std::string strVarName)
Input parameters	strVarName – name of the parameter
Return value	None
Description	Function removes value from the object description
Usage	Use it for temporary information, which you do not want to be stored any more

GetBinaryMap() – binary map

Synopsis	CSJBuffer* GetBinaryMap()
Input parameters	None
Return value	Returns pointer to the binary map for the object
Description	Location of the object is defined as a rectangular area, but not all of this area can be occupied by the object, this is a buffer with two values 0 and 0xFF, where 0xFF denotes that current pixel belongs to the object

Usage	Use it to get access to the binary map of the object

GetTransformed() – transformed image

Synopsis	CSJImage* GetTransformed()
Input parameters	None
Return value	Returns pointer to the transformed image of the object
Description	During feature extraction some algorithm could require to have a transform of the object into different representation (polar, frequency), so one can keep transformed image for use in more then one plug-in
Usage	Use it change or modify

HaveValue(std::string) – if a parameter exists

Synopsis	bool HaveValue(std::string strVarName)
Input parameters	strVarName – name of the parameter
Return value	true – if such a parameter was created for the object false – if not
Description	Function says if there is such a parameter created for the object
Usage	Use it to find if the object has such parameter. Because if you will ask for it is value, you will be given 0 or an empty string even if there is no such parameter

Clear() – reset the object information

Synopsis	void Clear()
Input parameters	None
Return value	None
Description	Clears binary map and transformed image, removes all assigned parameters
Usage	Use it to clear the description of the object

GetNameAt(int) – name of the parameter

Synopsis	std::string GetNameAt(int iIndex)
Input parameters	iIndex – index at which the name of the parameter should be taken
Return value	Returns the string with the parameter name
Description	Returns the string with the parameter name given its index
Usage	Use it to retrieve the name of saved parameter by index. Usually used to list all parameters saved for the object

GetValueAt (int) – name of the parameter

Synopsis	std::string GetValueAt(int iIndex)
Input parameters	iIndex – index at which name of the parameter should be taken
Return value	Returns the string with the value
Description	Returns the string with the value at given its index
Usage	Use it to retrieve the value of the parameter at given its index. It is the same as to use GeValue(GetNameAt(iIndex))

A.3.7 CSJObjectInfoList

class CSJObjectInfoList
#include “SJObjectInfoList .h”

Description	Stores a list of objects (class CSJObjectInfo)
Usage	Used to manage the list of CSJObjectInfo classes.

Class member functions:

CSJObjectInfoList() – default constructor

Synopsis	CSJObjectInfoList()

Input parameters	None
Return value	None
Description	Default constructor, which initializes all variables in the class
Usage	Called automatically

~CSJObjectInfoList() – destructor

Synopsis	virtual ~ CSJObjectInfoList()
Input parameters	None
Return value	None
Description	Takes care of cleaning the memory
Usage	Called automatically

Add(CSJObjectInfoList) – adding a new object to the list

Synopsis	void Add(CSJObjectInfoList newInfo)
Input parameters	newInfo – information about the object to be added
Return value	None
Description	Adds a new object to the list, by copying it into internal structure
Usage	Once used, the pointers obtained with GetObjectInfo() could be not valid

Delete(unsigned int) – deleting the image

Synopsis	void Delete(unsigned int iIndex=0)
Input parameters	iIndex – index of the object to be removed
Return value	None
Description	Deletes the object with a given index if one exists
Usage	Use it to remove an object description from the list

GetLength() – number of objects in the list

Synopsis	int GetLength()
Input parameters	None
Return value	Returns the number of objects added to the list
Description	Returns the number of objects added to the list
Usage	Use it to find out what is a valid range of indices for GetObjectInfo() or Delete()

GetObjectInfo(unsigned int) – get a stored object infromation

Synopsis	CSJObjectInfoList* GetImage(unsigned int iIndex)
Input parameters	iIndex – index of object to be returned
Return value	Returns the pointer to the object stored in the memory
Description	Returns the pointer to the object stored in the memory
Usage	Use it to retrieve the object stored in the list

Clear() – clears the list

Synopsis	void Clear()
Input parameters	None
Return value	None
Description	Removes all stored objects
Usage	Use it to remove all objects from the memory

A.3.8 CSJSpecimen

class CSJSpecimen
#include “SJSpecimen.h”

Description	Class encapsulates the information about the specimen used to analyze
Usage	Used to transfer and save all required information between plug-ins and the GUI

Class member functions:

CSJSpecimen() – default constructor

Synopsis	CSJSpecimen()
Input parameters	None
Return value	None
Description	Default constructor, which initializes all variables in the class, and clears them
Usage	Called automatically

~CSJSpecimen () – destructor

Synopsis	virtual ~ CSJSpecimen()
Input parameters	None
Return value	None
Description	Takes care of cleaning the memory
Usage	Called automatically

operator=(CSJSpecimen&) – assignment operator

Synopsis	CSJSpecimen &operator=(CSJSpecimen& fromobj)
Input parameters	fromobj – specimen to make this object to be equal to
Return value	Own address
Description	Assignment operator
Usage	Use it to make a copy of the object

GetNumberOfLayers() – number of images in the Specimen

Synopsis	int GetNumberOfLayers()
Input parameters	None
Return value	Returns the number of images added to the specimen
Description	Returns the number of images added to the specimen
Usage	Use it to find how many images are saved for this class

GetWidthOfLayer() – width of the images

Synopsis	int GetWidthOfLayer()
Input parameters	None
Return value	Returns the width of the images
Description	Returns the width of the images
Usage	Use it to find quickly the width of the saved images (all of them in general should have the same width, but the width of the first one is returned)

GetHeightOfLayer() – height of the images

Synopsis	int GetHeightOfLayer()
Input parameters	None
Return value	Returns the height of the images
Description	Returns the height of the images
Usage	Use it to find quickly the height of the saved images (all of them in general should have the same height, but the height of the first one is returned

GetImageList() – list of the images

Synopsis	CSJImageList* GetImageList()
Input parameters	None
Return value	Returns the pointer to the list of images
Description	Returns the pointer to the list of images stored for the specimen
Usage	Use it to get the direct access to the stored images

GetTempImages() – list of the temporary images

Synopsis	CSJImageList* GetTempImages ()
Input parameters	None

Return value	Returns the pointer to the list of temporary images
Description	Returns the pointer to the list of temporary images stored for the specimen
Usage	Use it when you want to save something for displaying, but it is not going to participate in the processing. Was made to show the transformations in different processing steps.

GetObjects() – object information

Synopsis	CSJObjectInfoList* GetObjects()
Input parameters	None
Return value	Returns the pointer to the list of information about objects
Description	Returns the pointer to the list of information about objects saved for the specimen
Usage	Use it to get the direct access to the object information

GetVars() – global variables and parameters

Synopsis	CSJObjectInfo* GetVars()
Input parameters	None
Return value	Returns the pointer to the global list of parameters
Description	This list allows storing of the parameters to pass them from one plug-in to another.
Usage	Use it to save the information in one plug-in and retrieve it in another one.

GetIniFile() – configuration file

Synopsis	CSJIniFile* GetIniFile()
Input parameters	None
Return value	Returns the pointer to the configuration file used to execute the plug-ins
Description	Gives you the access to the configuration file
Usage	Use it to see the saved configuration of the plug-ins

ReportProgress(int,float) – reporting status

Synopsis	BOOL ReportProgress(int iStage,float fPercentage)
Input parameters	iStage – stage of the process , possible values are SJ_PLGUIN_TYPE_ACQ, SJ_PLGUIN_TYPE_OBJ_SEP, SJ_PLGUIN_TYPE_DISPLAY, SJ_PLGUIN_TYPE_DATABASE, SJ_PLGUIN_TYPE_FEATURE fPercentage – percentage of completeness
Return value	true – in the case of successes false – otherwise
Description	Reports the progress of the performance (displayed in GUI), the first parameter says what is the state of the system and the second shows the percentage of the progress.
Usage	Use it in your plug-in to report the progress of the operation

SetReportVars(int*,float*) – setting report variables

Synopsis	void SetReportVars(int* piStage, float* pfProgress)
Input parameters	piStage – pointer to the integer value where the current stage will be stored pfProgress – pointer to the float value where the current percentage will be stored
Return value	None
Description	Sets the pointer to the variables which are used for ReportProgress() function
Usage	Use it in your GUI part of the system, to have variables to which the report will be saved. The GUI could update it is indicators according to thsese values

IsLogEmpty() – checks if there are messages in the log queue

Synopsis	bool IsLogEmpty()
Input parameters	None
Return value	true – if there are messages false – otherwise
Description	Shows if there are undeceived messages in the log

Usage	Use it in your GUI part of the system, to check if you need to receive a new messages for the log

AddLogMessage(std::string) – add a log message

Synopsis	void AddLogMessage(std::string str)
Input parameters	str – string with the log message
Return value	None
Description	Adds a new message to the end of the log messages queue
Usage	The plug-ins should use this function to send a log message to the main GUI program

GetLogMessage() – receive a log message

Synopsis	std::string GetLogMessage()
Input parameters	None
Return value	Returns the oldest log message
Description	Function retrieves the oldest log message and removes it from the log queue
Usage	Use it in your GUI part of the system, to update the log information. Use the loop while(!Specimen.IsLogEmpty()) to receive all log messages

SetAbortVar(bool*) – setting abort variable

Synopsis	void SetAbortVar(bool* pyAbort)
Input parameters	pyAbort – pointer to the variable of type bool for abort status
Return value	None
Description	Sets the pointer to the variable which is used to abort the processing
Usage	Use it in your GUI, so the user could abort the operation from it. All plug-ins should check IsAborted().

IsAborted() – check if abort is required

Synopsis	bool IsAborted()
Input parameters	None
Return value	true – if Abort() function was called, or main GUI requires the abort of the operation false – otherwise
Description	Checks the status of the execution
Usage	Use it whenever you want to check if execution should be aborted

Abort() – aborting execution

Synopsis	void Abort()
Input parameters	None
Return value	None
Description	Sets the abort state for the execution
Usage	Use it when you want the execution to be stopped. The plug-ins should interrupt their execution if IsAbort() returns true

Reset() – reseting class variables

Synopsis	void Reset()
Input parameters	None
Return value	None
Description	Resets all variables, clears all images, deletes all object descriptions
Usage	Use it to clear all information from the class

A.3.9 CSJIniFile

class CSJIniFile
#include “SJIniFile.h”

Description	Class to read, write and access information from ini files
Usage	Used to save all required information for the plug-ins setup. Note that contents of the ini file is read at once into memory on the call of Read() function, and all changes (addition, changing, deleting parameters) will be reflected in the ini file only after the call of Write() function. Comments are not supported in the ini files, they will be lost after Write() is called. Notation: Key – part of the ini file in [this key name] Variable – part of the ini file in format: variable_name=variable_value Section – part of the ini file from one Key (including this key) to another key with all variables in between. The only key included into the section serves as the identifier of the section. It is not allowed to have two sections with the same keys; they will be merged on loading. All values are stored as strings and can be retrieved as strings, which then can be converted to double or integer values. To create a section you have to write any variable to this section.

Class member functions:

CSJIniFile() – default constructor

Synopsis	CSJIniFile()
Input parameters	None
Return value	None
Description	Default constructor, which initializes all variables in the class, and clears them
Usage	Called automatically

~CSJIniFile() – destructor

Synopsis	virtual ~CSJIniFile()
Input parameters	None
Return value	None
Description	Takes care of cleaning the memory
Usage	Called automatically

SetPath(std::string) – set the ini file

Synopsis	void SetPath(std::string strNewPath)
Input parameters	strNewPath – complete path to the file, which should be read
Return value	None
Description	Sets the path for the ini file to be read on Read() function or written on Write() function. If the file does not exist it will be created when Write() is called
Usage	Use it before you call Read() or Write() functions. You can use one name to Read() the file and then the other one for Write(). This will create a new file with the contents of the first one

ReadFile() – load ini structure

Synopsis	bool ReadFile()
Input parameters	None
Return value	true – if file was successfully read false – otherwise
Description	Reads the ini file into internal representation
Usage	Use it to load the information of the ini file

WriteFile() – creates\updates the ini file

Synopsis	void WriteFile()

Input parameters	None
Return value	None
Description	Writes the information from the memory into file
Usage	Use it to update the information in the ini file

Reset() – reset contents

Synopsis	void Reset()
Input parameters	None
Return value	None
Description	Removes all section from the ini file
Usage	Use it to delete all sections and parameters of the ini file

GetNumKeys() – get number of keys in the ini file

Synopsis	ini GetNumKeys()
Input parameters	None
Return value	Returns the number of keys stored in the ini file
Description	Returns the number of keys stored in the ini file
Usage	Use it to find how many keys are currently created

GetNumValues(std::string) – get the number of variables in the section

Synopsis	int GetNumValues(std::string strKeyNa me)
Input parameters	strKeyName – key name for the section
Return value	Returns the number of variables for the section
Description	Returns the number of variables for the section
Usage	Use it to find out how many variables the section has

GetValue(std::string,std::string) – get the value of the variable

Synopsis	std::string GetValue(std::string strKeyName, std::string strVarName)
Input parameters	strKeyName – key name for the section strVarName – name of the variable
Return value	Returns the value of variable as a string
Description	Returns the value of variable as a string
Usage	Use it to get the value of the variable

GetValueI(std::string,std::string) – integer value of a variable

Synopsis	int GetValueI(std::string strKeyName, std::string strVarName)
Input parameters	strKeyName – key name for the section strVarName – name of the variable
Return value	Returns the value of variable as an integer
Description	Returns the value of variable as an integer if it was saved as integer or double
Usage	Use it to get the value of the variable

GetValueF(std::string,std::string) – double value of a variable

Synopsis	Double GetValueF(std::string strKeyName, std::string strVarName)
Input parameters	strKeyName – key name for the section strVarName – name of the variable
Return value	Returns the value of variable as an double
Description	Returns the value of variable as an double if it was saved as integer or double
Usage	Use it to get the value of the variable

SetValue(std::string,std::string,std::s tring,bool) – set a value of the variable

Synopsis	bool SetValue(std::string strKeyName, std::string strVarName, std::string strValue, bool yCreate = true)
Input parameters	strKeyName – key name for the section strVarName – name of the variable strValue – value of the variable

	yCreate – flag telling if the key should be created if it does not exist
Return value	true – if the variable was created or changed
	false – if the variable does not exist and yCreate is false
Description	Changes or creates the value of the variable in the given section
Usage	Use it to change or create a new variable of type string

SetValueI(std::string,std::string,int,bool) – set an integer value of the variable

Synopsis	bool SetValueI(std::string strKeyName, std::string strVarName, int iValue, bool yCreate = true)
Input parameters	strKeyName – key name for the section
	strVarName – name of the variable
	iValue – value of the variable to set
	yCreate – flag telling if the key should be created if it does not exist
Return value	true – if the variable was created or changed
	false – if the variable does not exist and yCreate is false
Description	Changes or creates the value of the variable in the given section
Usage	Use it to change or create a new variable

SetValueF(std::string,std::string,double,bool) – set a double value of the variable

Synopsis	bool SetValueI(std::string strKeyName, std::string strVarName, double dValue, bool yCreate = true)
Input parameters	strKeyName – key name for the section
	strVarName – name of the variable
	dValue – value of the variable to set
	yCreate – flag telling if the key should be created if it does not exist
Return value	true – if the variable was created or changed
	false – if the variable does not exist and yCreate is false
Description	Changes or creates the value of the variable in the given section
Usage	Use it to change or create a new variable

DeleteValue(std::string,std::string) – remove the variable from the section

Synopsis	bool DeleteValue(std::string strKeyName, std::string strVarName)
Input parameters	strKeyName – key name for the section
	strVarName – name of the variable
Return value	true – if the variable was deleted
	false – if the variable does not exist
Description	Removes the variable from the section
Usage	Use it if you do not want this variable to be present in the ini file

HaveKey(std::string) – if the section exists

Synopsis	bool HaveKey(std::string strKeyName)
Input parameters	strKeyName – name of the key
Return value	true – if there is a section with such key
	false – otherwise
Description	Function says if there is such a section in the ini file
Usage	Use it to find out if such a section is defined in the ini file

HaveVariable (std::string,std::string) – if the variable exists

Synopsis	bool HaveVariable(std::string strKeyName, std::string strVarName)
Input parameters	strKeyName – key name for the section
	strVarName – name of the variable
Return value	true – if there is such a variable
	false – otherwise
Description	Function says if there is such variable in the section
Usage	Use it to find out if such a variable exists in the section

GetVarName(std::string,int) – name of the variable

Synopsis	std::string GetVarName(std::string strKeyName,unsigned int iIndex)
Input parameters	strKeyName – key name for the section iIndex – index of the variable in the section
Return value	Returns the name of the variable at given index
Description	Function returns the name of the variable at given index
Usage	Use it if you want to print out all variables in the section

GetKeyName(int) – name of the section

Synopsis	std::string GetKeyName(unsigned int iIndex)
Input parameters	iIndex – index of the section in ini file
Return value	Returns the name of the section at given index
Description	Function returns the name of the section at given index
Usage	Use it if you want to print out all sections in the ini file

DeleteKey(std::string) – delete section

Synopsis	bool DeleteKey(std::string strKeyName)
Input parameters	strKeyName – key name for the section
Return value	true – if there is a section with such section false – otherwise
Description	Removes the section with the key and all variables
Usage	Use it if you want to get rid of the section and all its variables

GetPath() – current ini file

Synopsis	std::string GetPath()
Input parameters	None
Return value	Returns the path and the name of the ini file
Description	Returns the path set with the last SetPath() command
Usage	Use it if you want to know where function Write() will write the ini structure

APPENDIX B

INTERFACE FUNCTION DESCRIPTION

The header files with the functions are: MainLoop.h, Processing.h, Utils.h;

MainLoop.h header file contains two functions: MainLoop and ResearchLoop. These functions take care of the flow of the operation in two modes: Normal Mode and Research Mode. Processing.h contains some processing routines. Utils.h contains the utility routines.

B.1 Normal Mode

This mode serves for the normal execution of the configuration file when the data is acquired and processed. User has minimum interaction with the program at this mode. The system works automatically. The plug-ins still can ask for some information. The function for the Normal Mode is decleared in ManiLoop.h.

Synopsis	int MainLoop(CSJSpecimen &Specimen, tagSJProcAddresses &prAddr, WORD wScanNum=1)
Input parameters	Specimen – specimen object to use in the system prAddr – structure with the pointers to the functions from loaded plug-ins wScanNum – number of scans to perform (0 – infinite loop) default is 1
Return value	Returns one of the following: ML_FINISHED – finished correctly; ML_USER_INTERRUPTED – user interuption ML_BAD_PROC_ADDR – bad pointer to the function in the plug-in is given

This structure should be passed to the MainLoop with addresses of functions loaded from the plug-ins.

Function types:

```
typedef int(*fntGetInfo)(std::string&);
typedef int(*fntSetupForDll)(const char*);
typedef int(*fntGetType)();
typedef int(*fntAcquire)(CSJSpecimen&,int);
typedef int(*fntSeparate)(CSJSpecimen&);
```

```
typedef int(*fntAnalyze)(CSJSpecimen&);
typedef int(*fntDisplay)(CSJSpecimen&);
typedef int(*fntInit)(CSJSpecimen&);
typedef int(*fntDone)(CSJSpecimen&);
typedef int(*fntExtractFeature)(CSJSpecimen&);
```

Structure for saving plug-in addresses:

```
const int MAX_PLUGINS = 120;
struct tagSJProcAddresses{
        fntAcquire              Acquire;
        fntDisplay              Display,
                                Update;
        fntSeparate             Separate;
        fntAnalyze              Analyze;
        fntInit                 AcqInit,
                                SepInit,
                                AnInit,
                                DispResearchInit,
                                DispInit;
        fntDone                 AcqDone,
                                SepDone,
                                AnDone,
                                DispResearchDone,
                                DispDone;
        fntSetupForDll AcqSetup,
                                SepSetup,
                                AnSetup,
                                FeatureSetup[MAX_PLUGINS];
        int                     iTotalFeatures;
        fntExtractFeature Feature[MAX_PLUGINS];};
```

B.2 Research Mode

This mode allows the configuration of the plug-ins, displaying the effect of the current configuration. It also allows stepping back to the previous plug-in setup when it is required. In this mode a user can find the optimal setup parameters to be used in Normal Mode. It is declared in ManiLoop.h.

Synopsis	ResearchLoop(CSJSpecimen &Specimen, tagSJProcAddresses &prAddr)
Input parameters	Specimen – specimen object to use in the system prAddr – structure with the pointers to the functions from loaded plug-ins (see description in section 2.4.1)
Return value	Returns one of the following: ML_FINISHED – finished correctly; ML_USER_INTERRUPTED – user interuption ML_BAD_PROC_ADDR – bad pointer to the function in the plug-in is given ML_NO_DISPLAY – No Update function in the display module

In this mode the system executes the setup function of the plug-in and then the plug-in itself. If the return value from the plug-in function is non-zero, then the same sequence of operations is performed on

the next plug-in. In the case when a zero was returned, the setup function for the current plug-in is executed again. If the setup function returns zero then step back to the previous plug-in is performed. The order of the plug-ins is as follows:

1. Acquisition plug-in;
2. Object Separation plug-in;
3. Feature plug-ins according to their order;
4. Database plug-in;
5. Display plug-in;

For each plug-in the following sequence is called:

1. Setup of the plug-in;
2. If step 1 returns 0 value go back to the previous plug-in (if there is no previous plug-in, exit the procedure);
3. Execute the plug-in;
4. Show the result by using Update function of Display plug-in;
5. If step 4 returns 0 value repeat from step 1, otherwise go to the next plug-in.

Use Research Mode when you are not sure what parameters should be used for the plug-ins.

B.3 Processing routines

These functions can be called by any plug-in and are designed to simplify the processing tasks. They are declared in Processing.h.

IncreaseDepth(CSJBuffer&) – changes the depth of the buffer to two bytes per pixel

Synopsis	void IncreaseDepth(CSJBuffer& buf)
Input parameters	buf – buffer object to be modified
Return value	None
Description	Change the depth of the given buffer from one byte per pixel to two bytes per pixel.
Usage	Use it when you want to be able to store wider range of values in the buffer. One byte per pixel gives the range from 0 to 0xFF, while two bytes per pixel gives the range from 0 to 0xFFFF.

Threshold(CSJBuffer,int,WORD,WORD) – thresholds the buffer

Synopsis	int Threshold(CSJBuffer &Buf,int iThreshold, WORD wUp=0xFFFF, WORD wDown=0)
Input parameters	Buf – buffer object to be modified iThreshold – value of the threshold wUp – value to be assigned if the pixel value is greater or equal to the threshold, default is the largest possible value for the buffer wDown – value to be assigned if pixel is less then threshold, default is 0
Return value	1 – if successful 0 – otherwise
Description	Thresholds the values in the buffer.
Usage	Use it when you want to make a buffer with only two values in it.

StretchVals(CSJBuffer&) – stretch values in the buffer

Synopsis	int StretchVals(CSJBuffer &Buf)

Input parameters	Buf – buffer object to be modified
Return value	1 – if successful 0 – otherwise
Description	Maps the values from the region [min in the buffer, max in the buffer] to the region [0 max possible value in the buffer]
Usage	Use it when you want to use the whole dynamic range of the buffer

Connectivity(CSJBuffer*) – labeling of the blobs

Synopsis	WORD Connectivity(CSJBuffer *BufBlobs)
Input parameters	BufBlobs – pointer to the binary buffer to label.
Return value	Returns number of unique labels assigned to the buffer
Description	Labels ??? pixels in the buffer using 4-neighbor method. The maximum number of the labels, which could be assigned is 0xFFFF
Usage	Use it on the thresholded image to get blobs labeled

Find_Blobs(CSJObjectInfoList*,CSJBuffer*,bool) – finding blobs

Synopsis	int Find_Blobs(CSJObjectInfoList *ObjectList, CSJBuffer *BufBlobs, bool yExcludeSE=true)
Input parameters	ObjectList – pointer to the list to put the information about the objects BufBlobs – pointer to the buffer with the labeled blobs yExcludeSE – if true blobs with the width or height equal to 1, or the blobs touching the edge will not be included into the list
Return value	1 – if successful 0 – otherwise
Description	Fills the list of objects with the information about the bounding boxes of the objects
Usage	Use it after you perform the connectivity on the buffer to get the information about blobs.

Change_Value(CSJBuffer&,WORD,WORD) – changing value

Synopsis	int Change_Value(CSJBuffer &Buf, WORD wOld, WORD wNew)
Input parameters	Buf –buffer to be modified wOld – value to change wNew – new value to assign to the pixels which has wOld value
Return value	1 – if successful 0 – otherwise
Description	Changes the given value of the buffer to a new one.
Usage	Use it if you want to change one value in the buffer to another one.

B.4 Utility routines

These functions can be called by any plug-in and are designed to simplify data managing. They are declared in Utills.h.

Format(const char *fmt, ...) – formatting string

Synopsis	std::string Format(const char *fmt, ...)
Input parameters	fmt – string with the format control … – optional arguments (see help for C function printf() for more information)
Return value	Returns the formatted string
Description	Formats the string given the format control string and parameters
Usage	Use when you need to have a string of the certain format. See C help for printf() function

Round(double) – rounding a number

Synopsis	int Round(double dX);
Input parameters	dX – number to be rounded
Return value	Returns the rounded number
Description	Rounds the given number, if the fractional part of the number is greater then 0.5 then the next larger integer is returned. Otherwise just the integer part is returned
Usage	Use it to round a number

APPENDIX C

PLUG-IN INTERFACING

This appendix describes the interfacing between the plug-ins. There are 5types of the plug-ins: Acquisition, Object separation, Feature extraction, Database and Display. Each configuration of the system must have one Acquisition plug-in. Only one per configuration plug-in of Object separation, Database and Display is allowed. But there could be 0 or more Feature extraction plug-ins. All plug-ins should be loaded into the memory and the addresses of the functions should be supplied to the interface (See the attached CD for examples and documentation)

C.1 Common functions for all plug-ins

The described function should be exported from all types of the plug-ins and usually are used by GUI, because the algorithms supplied by the system do not call them internally.

GetType() – type of the plug-in

Synopsis	int GetType()
Declaration	typedef int(*fntGetType)()
Input parameters	None
Return value	Returns the type of the plug-in
Description	Returns the type of the plug-in. The supported values are: SJ_PLGUIN_TYPE_ACQ, SJ_PLGUIN_TYPE_OBJ_SEP, SJ_PLGUIN_TYPE_DISPLAY, SJ_PLGUIN_TYPE_DATABASE, SJ_PLGUIN_TYPE_FEATURE
Usage	Only the GUI uses this function to get the type of the plug-in.

GetInfo(std::string&) – information about plug-in

Synopsis	int GetInfo(std::string& Info)
Declaration	typedef int(*fntGetInfo)(std::string&)
Input parameters	Info – address of the string to which information will be stored
Return value	Returns the type of the setup for this plug-in
Description	Function modifies the supplied string to show the information about the name of the plug-in and its version. Returned value is a type of the setup for the plug-in, supported values are: SJ_MFC_DIALOG_SETUP, SJ_NO_SETUP_DIALOG SJ_MFC_DIALOG_SETUP – says that plug-in has dialog box to ask user about the parameters;

	SJ_NO_SETUP_DIALOG – plug-in does not have any dialog for the setup
Usage	This function is used by the GUI only to find out whether it should call SetupDll() function

SetupDll(const char*) – setup the dll

Synopsis	int SetupDll(const char *iniName)
Declaration	typedef int(*fntSetupForDll)(const char*)
Input parameters	iniName – pointer to the first character of the string with the name of the ini file where the configuration should be saved
Return value	non 0 value – if setup was successful 0 – in the case of problem
Description	Function should set up all parameters for the configuration. It manages communicating with the user and saving the values to the ini file. These values then will be used by Init() function
Usage	This function is used only by the GUI to set up the configuration file

The following functions does not apply to the feature extraction plug-ins:

Init(CSJSpecimen&) – initializing

Synopsis	int Init(CSJSpecimen &Specimen)
Declaration	typedef int(*fntInit)(CSJSpecimen&)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins
Return value	non 0 value – if call is successful 0 – in the case of any problem
Description	Function is supposed to initialize the internal variables and is called once per session. The session is ended with the call of Done()
Usage	Put routines which should be called only once per session in this procedure (Like initializing hardware registers if any). This function is called internally

Done(CSJSpecimen&) – closing

Synopsis	int Done(CSJSpecimen &Specimen)
Declaration	typedef int(*fntDone)(CSJSpecimen&)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins
Return value	non 0 value – if call is successful 0 – in the case of problem
Description	Function is supposed to clear internal variables and is called once per session. The function is the opposite of Init(). And it is called last in the session
Usage	Use it if you have some routines which is called only once per session. This function is called internally

C.2 Acquisition plug-in

This plug-in is responsible for loading the image data into CSJSpecimen object. It interfaces the hardware or the operating system routines with the rest of the system.

Functions to be exported for this type of the plug-in are as follows:

Acquire(CSJSpecimen&,int) – acquiring the data

Synopsis	int Acquire(CSJSpecimen &Specimen,int iNum)
Declaration	typedef int(*fntAcquire)(CSJSpecimen&,int)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins. All data should be acquired and saved into image list of this object iNum – iteration of acquisition
Return value	non 0 value – if call is successful

	0 – in the case of problem
Description	Function is responsible for the acquiring data. Data is saved in the Specimen.
Usage	This procedure should call all routines to acquire the data. iNum variable is used for a continues mode of work to tell at what iteration the system is

C.3 Display plug-in

This plug-in is responsible for the displaying the results of the processing. Whether it should be a new image or some text data, or both at the same time. The result of the processing is stored in the Specimen object, which is passed as a parameter. This plug-in is interfacing the output devices with the system.

Functions to be exported for this type of the plug-in are as follows:

Display(CSJSpecimen&) – displaying the result

Synopsis	int Display(CSJSpecimen &Specimen)
Declaration	typedef int(*fntDisplay)(CSJSpecimen&)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins. All data is taken from image the list or the variable list
Return value	non 0 value – if call is successful 0 – in the case of problem
Description	Function is responsible for the displaying of the data
Usage	This procedure should call all routines to display the data. It is called internally and could interact with a user or just somehow save the data

Update(CSJSpecimen&) – updating the result

Synopsis	int Display(CSJSpecimen &Specimen)
Declaration	Typedef int(*fntDisplay)(CSJSpecimen&) (the same as for Display function)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins. All data is taken from image list or variable list
Return value	non 0 value – if call is successful 0 – otherwise
Description	Function is responsible for the displaying of the data.
Usage	This procedure should call all routines to display the data during Research Mode. It is called internally and could interact with a user or just somehow save the data

ResearchInit(CSJSpecimen&) – initializing Research Mode

Synopsis	int ResearchInit(CSJSpecimen &Specimen)
Declaration	typedef int(*fntInit)(CSJSpecimen&)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins. Ini structure is saved in this object and function should use it
Return value	non 0 value – if call is successful 0 – otherwise
Description	Function is supposed to initialize tge internal variables and is called once per session. The session is ended with the call of ResearchDone() function
Usage	Put routines which should be called only once per session in this procedure (Like initializing hardware registers if any)

ResearchDone(CSJSpecimen&) – closing Research Mode

Synopsis	int ResearchDone(CSJSpecimen &Specimen)
Declaration	Typedef int(*fntDone)(CSJSpecimen&)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins.

	Ini structure is saved in this object and function should use it.
Return value	non 0 value – if call is successful 0 – otherwise
Description	Function is supposed to clear the internal variables and is called once per session. The function is the opposite of ResearchInit() function. And it is called last in the session
Usage	Use it if you have some routines, which should be called only once per session

C.4 Object separation plug-in

This plug-in is responsible for the image segmentation and the object detection. It is called after the acquisition and should find the location of the objects.

Function to be exported for this type of the plug-in is:

Separate(CSJSpecimen&) – finding objects

Synopsis	int Separate(CSJSpecimen &Specimen)
Declaration	typedef int(*fntSeparate)(CSJSpecimen&)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins. All data is taken from the image list or the variable list
Return value	non 0 value – if call is successful 0 – otherwise
Description	Function is responsible for finding the objects in the supplied data
Usage	This procedure should call all routines to find the objects

C.5 Feature Extraction plug-in

The plug-in is responsible for the detecting a certain feature (or a set of features) in the detected objects. Also it could create new objects or delete existing ones. The information is saved into the Specimen object. It is responsibility of the plug-in to perform its operations on all objects.

Function to be exported for this type of the plug-in is:

ExtractFeature(CSJSpecimen&) – performing an operation

Synopsis	int ExtractFeature(CSJSpecimen &Specimen)
Declaration	typedef int(*fntExtractFeature)(CSJSpecimen&)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins. All data is taken from the image list or the variable list
Return value	non 0 value – if call is successful 0 – otherwise
Description	Function is responsible for performing a predefined procedure on all objects in the specimen
Usage	Should perform some operations on the data, extracting the features of the objects, splitting them, deleting them, finding new ones and so on

C.6 Data base plug-in

This plug-in is responsible for making a final decision about the type of the objects found in the specimen.

Function to be exported for this type of the plug-in is:

Analyze(CSJSpecimen&) – assigning the labels to the objects

Synopsis	int Analyze(CSJSpecimen &Specimen)
Declaration	typedef int(*fntAnalyze)(CSJSpecimen&)
Input parameters	Specimen – address of the CSJSpecimen object, which is passed to all plug-ins. All data is taken from image list or variable list
Return value	non 0 value – if call is successful 0 – otherwise
Description	Function is responsible for performing a predefined procedure on all the objects in the specimen
Usage	Usually assigns one of the predefined labels to each object. This data is then displayed by the Display plug-in. It should utilize the data supplied by the feature extraction plug-ins to make the decision

APPENDIX D

DESIGNED PLUG-IN DESCRIPTION

This appendix contains the description of the plug-ins.

D.1 DummyAcq plug-in

This plug-in uses the ini files with extension “des” (DEScription) of the following format: The required section for the description file is [Images] with the variable NumOfImages.

```
[Images]
NumOfImages=X
```

NumOfImages contains the number of images to be loaded during the acquisition. There should be the number of the sections [ImageXXX], where XXX is an integer from 0 to NumOfImages-1. Each section should have the variable Path=path_to_the_picture. This is the location of the TIFF image file to load.

Example of a description file:

```
[Images]
NumOfImages=1
[Image0]
Path=C:\users\SJ\Thesis\Samples\c_15_2.tif
```

Example with more images:

```
[Images]
NumOfImages=3
[Image0]
Path=C:\users\SJ\Thesis\Samples\1.tif
[Image1]
Path=C:\users\SJ\Thesis\Samples\2.tif
[Image2]
Path=C:\users\SJ\Thesis\Samples\3.tif
```

The setup dialog box of the plug-in is shown in Figure D.1

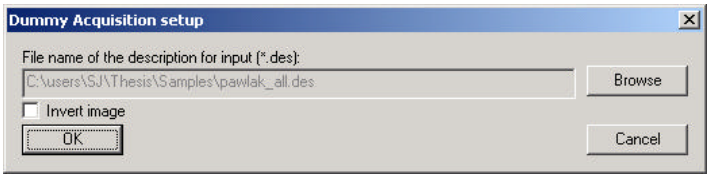


Figure D.1 Setup dialog box of DummyAcq plug-in

With the button Browse you can choose the description file (described latter). Also a user defines if the images should be inverted. Press OK button to save the setup, or Cancel to discard the changes.

D.2 FromFileAcq plug-in

This plug-in was designed to load the information from TIFF image file. It is useful when you want to test a specific configuration with a synthetic data or if you need to emulate the hardware acquisition. When the module has to acquire the data, the plug-in shows the standard file open dialog box to open an image file. A user has to choose the file and it will be used as the data coming into the system. Once the file is chosen, program will ask the user if he/she wants this image to be inverted, sometimes it can be required by the Object Separation plug-in. The supported image format is TIFF. You can create images in any image editor (like Potoshop®[13]) program and save it as a TIFF image. There is no limitation for the size of the image, except the memory of the computer.

D.3 SimpleDisplay plug-in

This plug-in can be used to see most of the information stored in the Specimen object supplied to the Display plug-in. Though it does not provide some specific functionality, which could be required by certain tasks (like saving the data to the disk). It easily can be used for the exploration of the problem and the demonstration of the implementation for a display plug-in.

The main window of the SimpleDisplay plug-in is shown in Figure D.2

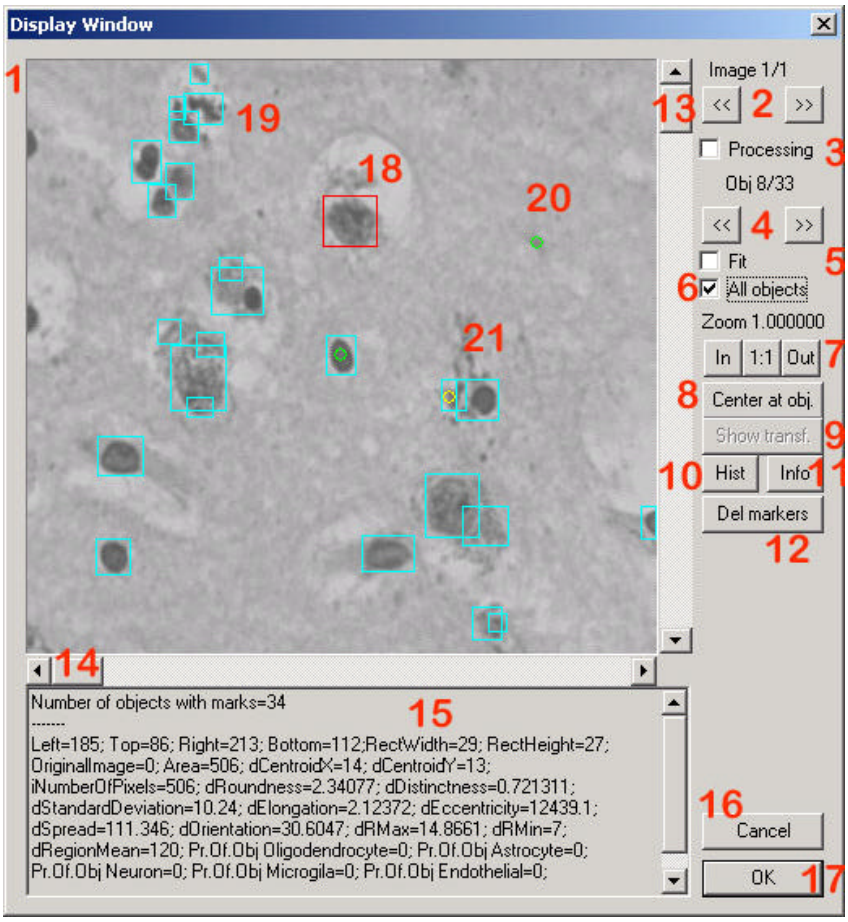


Figure D.2 Main window of SimpleDisplay

Explanation of controls:

1. Area with the image information;

2. The set of controls to choose the image. If more then one image is available. The label shows the number of shown image versus the total number of images. Depending on the control 3 it relates to the temporary images or working images;

3. Processing checkbox – if checked the images added by the plug-ins during the processing are shown;

4. The set of controls to navigate through the objects. The label shows the number of the current object versus the total number of objects. The current object is highlighted with a red rectangle around the object;

5. Fit checkbox – if set the image will be shown completely within the image area. If unchecked the real size of the image is used and to see parts of the images that are not visible, a user should use scrollbars 13 and 14;

6. All objects checkbox – if set a blue rectangle around each object is shown;

7. Zoom controls – the label shows current zoom. In and Out buttons zoom in and zoom out respectively. 1:1 button resets zoom to 1;
8. Center at object button – set the scroll positions and zoom to show the selected object;
9. Show transformed button– shows the transformed image of the object if available;
10. Histogram button – displays the dialog box to pick a parameter to build a histogram for;
11. Information button – displays all objects variables in one list;
12. Delete markers button – removes all markers;
13. Vertical scroll bar – use it if you want to see other areas of the image;
14. Horizontal scroll bar – use it if you want to see other areas of the image;
15. Information about the current object;
16. Cancel button – closes the dialog box and returns zero (used during Research Mode)
17. Ok button – closes the dialog box and returns non-zero;
18. Red rectangle shown to highlight the current object;
19. Blue rectangles highlight objects in the image;
20. Green round mark – an additive mark (if you want to count something or just mark);

    To add the additive mark, double click with the left button of the mouse at the location where you want this mark to be added. If there is a subtractive mark at this location, it will be removed and no additive mark will be added;

21. Yellow round mark – a subtractive mark;

    To the subtractive mark, double click with the right button of the mouse at the location where you want this mark to be added. If there is an additive mark at this location, it will be removed and no subtractive mark will be added;

The information dialog box (Figure D.3) allows anyone to see all variables saved for each object and the global variables for the specimen.

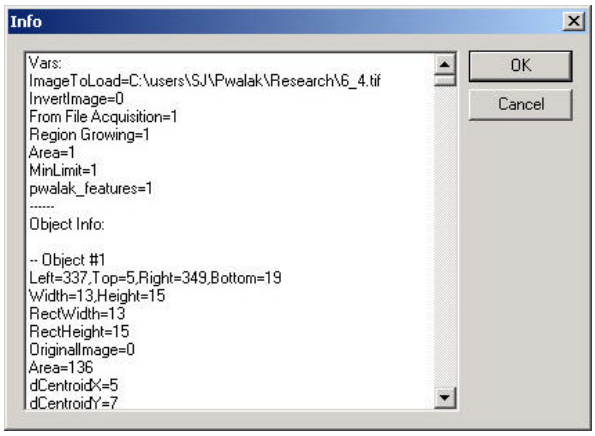


Figure D.3 Information dialog box

When Histogram button is clicked, the dialog box (Figure D.4) is shown, where a user is asked to choose the parameter for which he/she wants the histogram to be built.

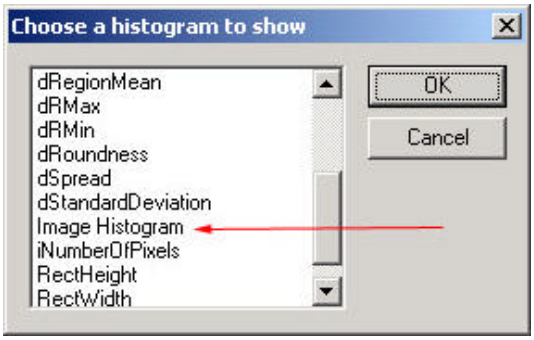


Figure D.4 Choose histogram dialog box

This function looks for all numerical variables in the variable list for each object and then builds the histogram for these variables. One special histogram type is an Image Histogram (marked with the arrow on Figure D.4) – which allows the user to see the histogram of the image in the view area.

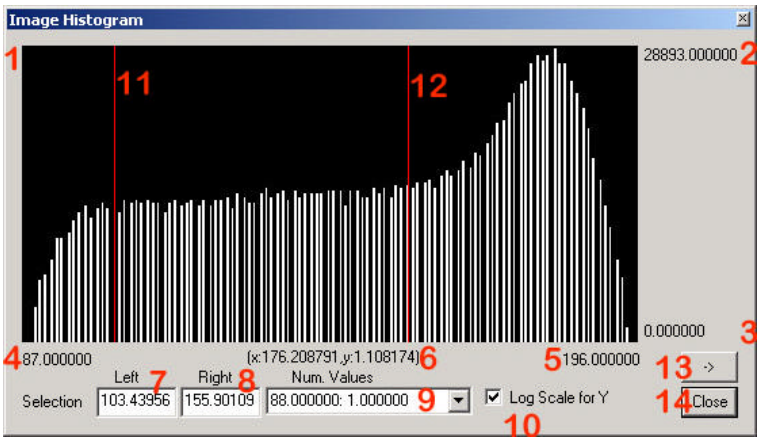
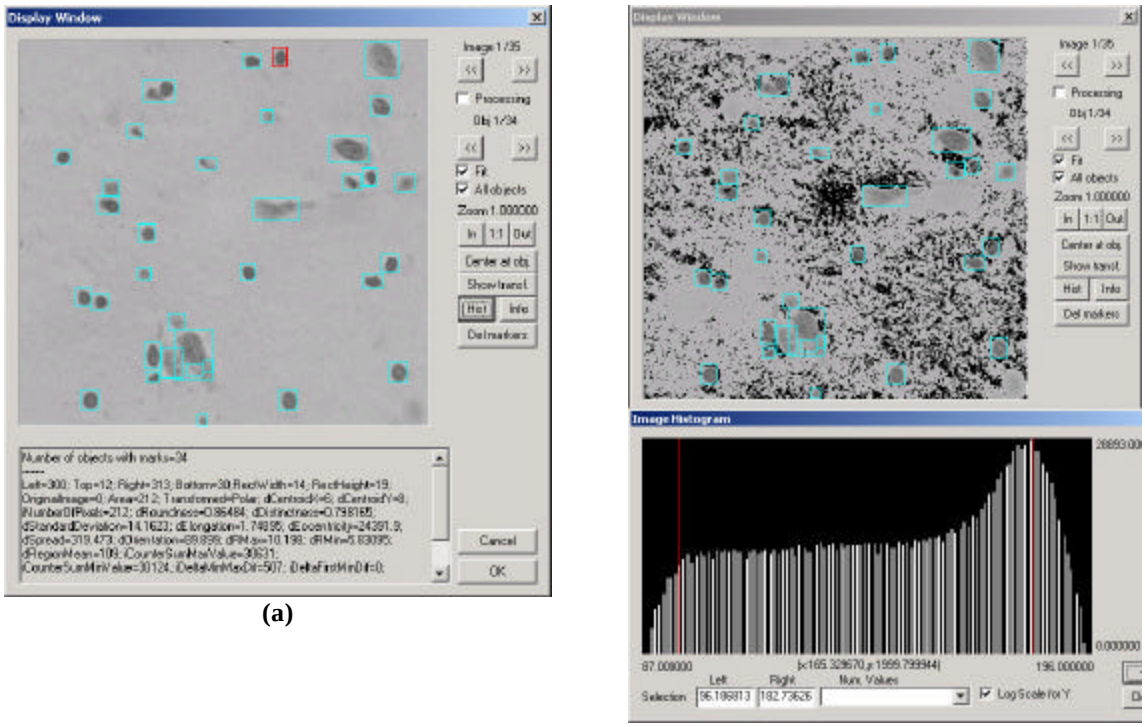


Figure D.5 Histogram window

Explanation of controls:

1. Histogram itself;
2. Maximum value of the histogram (maximum number of objects (pixels) with the same value);
3. Minimum value of the histogram (always 0);
4. Minimum value of the parameter for which the histogram is built;
5. Maximum value of the parameter for which the histogram is built;
6. Value of X and Y under the mouse cursor;
7. Value for the left selection marker;
8. Value for the right selection marker;
9. Values of the bins which are shown: xxx: yyy, where xxx – value of the parameter, yyy – number of the objects (pixels) with this value;
10. Log scale for Y check-box – if checked Y axe of the histogram has logarithmic scale. It is useful when the scale of the Y axis has a big range;
11. Left selection marker (it is set by the left mouse click on the histogram, could not be on the right side of the right selection marker);
12. Right selection marker (it is set by the right mouse click on the histogram, could not be on the left side of the left selection marker);
13. Apply to Image button – press it to see only the selected range of the objects (pixels) (which has values in the selected region) See Figure D.6 (a) and (b);
14. Close button – closes the window;



(a) **(b)**

Figure D.6 Original image (a) and image with selection applied (b)

D.4 Area and Center plug-ins

These are the Feature plug-ins, which calculate the area occupied by each object and the center coordinates of each object. Area plug-in uses the binary mask of the object to find the number of the pixels for the object. This value is stored as the information about the object in the variable with the name “Area”. Center plug-in uses bounding rectangle of the object to find the center point of the area occupied by the object.

D.5 MinArea and MaxArea plug-ins

These two plug-ins are the Feature plug-ins. They require Area plug-in to be used before them. They take the value stored by Area plug-in and compare it to the defined during the setup value. If object is less (for MinArea plug-in) or more (for MaxArea plug-in) then object is deleted. If Area plug-in was not executed before these plug-ins, user will be asked if the execution of the rest of the plug-ins is required or the system should halt the execution.

D.6 IniDB plug-in

This is a Database plug-in, used to handle fuzzy-logic with membership functions. The only parameter, which is set during the setup, is the name of the database configuration file. The database is an ini file with the extension dbi or ini. Database consists of 3 different parts: Membership functions for parameters, Rules, Object descriptions.

The plug-in loads parameters, membership functions and object descriptions with the rules. Then it loops through all objects, for each object a rule is calculated. For each parameter the value with maximum weight is picked (Small, Medium, Large). Then this calculated rule is compared with the rules of the objects until one matches. If no match was found "unknown" label is assigned. If there is no value assigned for the parameter in the rule, then it is not considered during comparison.

Object description format:

A key section should have the name [Obj%d], where %d is an integer (Examples [Obj0], [Obj1],[Obj345]). Numbers should be sequential and start from **1**. The maximum number of objects is 32,000. Parameters required for this section are Name – the label to be assigned to the object, Rules – the number of rules defining this object (Coma "," separates rules). The following is an example of the object description:

```
[Obj1]
Name=Oligodendrocyte
Rules=55,56,64,28,1,57,281
```

Rules description format:

A key section should have the name [R%d], where %d is an integer (exmples [R1],[R45],[R540]) Object description refers to this number in the variable Rules. If no object has the number of the rule in its Rules variable the rule is ignored. The maximum number of rules is 32,000 and it is the maximum number, which could be assigned to the rule. The rule contains the values of the parameters. In the form P%d=[Small|Medium|Large], where P%d is a section name of the parameter membership function, and it can have one of the three values: Small, Medium or Large (case sensitive).

This is an example of the rule:

```
[R55]
P1=Small
```

```
P2=Large
P3=Small
P4=Medium
P5=Small
```

Parameters description format:

A key section should have the name [P%d], where %d is an integer (examples [P1],[P2],[P438]). Numbers should be sequential and start from **1**. The maximum number of objects is 32,000. Four parameters are required in this section: Name, Small, Medium and Large. Name is the name of the feature as it is saved in the ObjectInfo list for the object. Small – region described by two points "small 1" and "small 2" refer to Figure D.7. First point says till what value of the parameter Small description has weight of 1. Second point says where weight becomes 0. The weight is linearly calculated in between of these points. Medium region is described by 4 points: "medium1", "medium2", "medium3" and "medium4". Large region: "large 1", "large 2". All numbers are separated by comas ","

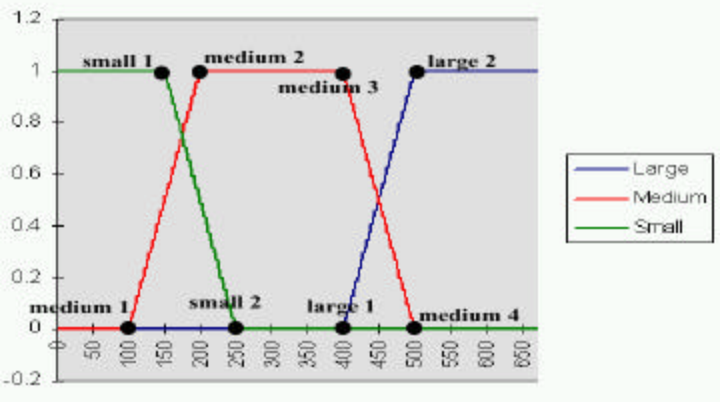


Figure D.7 Membership functions

This is an example of the parameter section for the Figure D.7:

```
[P1]
Name=Name_for_this_feature
Small=150,250
Medium=100,200,400,500
Large=400,500
```

The following is an example of the complete database used for human brain cell recognition:

[P1]
Name=Area
Small=150,270
Medium=100,270,400,500
Large=400,500

[P2]
Name=dDistinctness
Small=0.65,0.72
Medium=0.71,0.73,0.78,0.83
Large=0.78,0.83

[P3]
Name=dEntropy
Small=0.019,0.0206
Medium=0.019,0.02065,0.029,0.032
Large=0.03,0.032

[P4]
Name=dRMax
Small=7.5,10
Medium=7.3,7.7,12.5,15
Large=12.6,15

[P5]
Name=dRoundness
Small=0.75,0.85
Medium=0.70,0.75,0.8,0.85
Large=0.75,0.85

[Obj1]
Name=Oligodendrocyte
Rules=55,56,64,28,1,57,281

[Obj2]
Name=Astrocyte
Rules=2,3,122,131,123,121,1221,1212

[Obj3]
Name=Neuron
Rules=189,180,207,215,206,216,188

[R55]
P1=Small
P2=Small
P3=Small
P4=Medium
P5=Small

[R57]
P1=Small
P2=Large
P3=Large
P4=Small
P5=Small

[R58]
P1=Small
P2=Large
P3=Large
P4=Small
P5=Medium

[R56]
P1=Small
P2=Large
P3=Small
P4=Small
P5=Medium

[R64]
P1=Small
P2=Large
P3=Medium
P4=Small
P5=Small

[R28]
P1=Small
P2=Medium
P3=Small
P4=Small
P5=Small

[R281]
P1=Small
P2=Medium
P3=Large
P4=Medium
P5=Medium

[R122]
P1=Medium
P2=Medium
P3=Medium
P4=Medium
P5=Medium

[R1221]
P1=Medium
P2=Small
P3=Medium
P4=Large
P5=Medium

[R2]
P1=Small
P2=Large
P3=Large
P4=Medium
P5=Medium

[R3]
P1=Medium
P2=Large
P3=Medium
P4=Medium
P5=Medium

[R131]
P1=Medium
P2=Medium
P3=Large
P4=Medium
P5=Medium

[R123]
P1=Medium
P2=Medium
P3=Medium
P4=Medium
P5=Large

[R121]
P1=Medium
P2=Medium
P3=Medium
P4=Medium
P5=Small

[R1212]
P1=Medium
P2=Small
P3=Medium
P4=Medium
P5=Large

[R189]
P1=Large
P2=Small
P3=Small
P4=Large
P5=Large

[R180]
P1=Large
P2=Small
P3=Small
P4=Large
P5=Large

[R207]
P1=Large
P2=Medium
P3=Small
P4=Large
P5=Large

[R215]
P1=Large
P2=Medium
P3=Small
P4=Large
P5=Medium

[R206]
P1=Large
P2=Medium
P3=Small
P4=Large
P5=Medium

[R216]
P1=Large
P2=Medium
P3=Small
P4=Large
P5=Large

[R188]
P1=Large
P2=Small
P3=Small
P4=Large
P5=Medium

[R1]
P1=Medium
P2=Small
P3=Small
P4=Medium
P5=Small

D.7 SOMDB plug-in

This is a Database plug-in. It provides the GUI to collect the data for training the self-organizing map (SOM) and makes decisions based on this map (See Section 4.8 for description of the SOM). External programs perform all routines. The following section describes the details.

There are 5 programs:

1. randinit.exe – program initialize the map for training;
2. vsom.exe - trains the reference vectors;
3. vcal.exe – labels the map vectors;
4. visual.exe – generates a list of coordinates corresponding to the best-matching unit in the map for each data sample in the input data;
5. qerror.exe – calculates the average quantization error.

For more details about these programs refer to "The Self--Organizing Map Program Package Version 3.1" [10]

D.7.1 Parameters

Various programs need various parameters. All the parameters that are required by any program are listed below. The meaning of the parameters is obvious in most cases. The parameters can be given in any order in the commands.

-din	Name of the input data file.
-dout	Name of the output data file
-cin	Name of the file from which the reference vectors are read
-cout	Name of the file to which the reference vectors are stored
-rlen	Running length (number of steps) in training
-alpha	Initial learning rate parameter. Decreases linearly to zero during training
-neigh	The neighborhood function type used. Possible choices are step function (bubble) and Gaussian (gaussian)
-topol	The topology type used in the map. Possible choices are hexagonal lattice (hexa) and rectangular lattice (rect)
-ydim	Number of units in the y-direction
-xdim	Number of units in the x-direction
-radius	Initial radius of the training area in som-algorithm. Decreases linearly to one during training

D.7.2 Format of input/output files

All data files (input vectors and maps) are stored as ASCII (American Standard Code for Information Interchange) files for their easy editing and checking. The files that contain training data and test data are formally similar, and can be used interchangeably. The input data is stored in ASCII-form as a list of entries, one line being reserved for each vectorial sample. The first line of the file is reserved for status knowledge of the entries; in the present version it is used to define the following items (these items MUST occur in the indicated order; in data files the optional items are ignored):

1. Dimensionality of the vectors (integer, compulsory).
2. Topology type, either hexa or rect (string, optional, case-sensitive).
3. Map dimension in x-direction (integer, optional).
4. Map dimension in y-direction (integer, optional).
5. Neighborhood type, either bubble or gaussian (string, optional, case-sensitive).

Subsequent lines consist of n floating-point numbers followed by an optional class label (that can be any string) and two optional qualifiers (see below) that determine the usage of the corresponding data entry in training programs. The data files can also contain an arbitrary number of comment lines that begin with '#', and are ignored. (One '#' for each comment line is needed.)

If some components of some data vectors are missing (due to data collection failures or any other reason) those components should be marked with 'x' (replacing the numerical value). For example, a part of a 5-dimensional data file might look like:

```
1.1  2.0  0.5   4.0  5.5
1.3  6.0   x    2.9   x
1.9  1.5  0.1   0.3   x
```

The map files produced by the program, and the user usually does not need to examine them by hand. The reference vectors are stored in ASCII-form. The format of the entries is similar to that used in the input data files, except that the optional items on the first line of data files (topology type, x- and y-dimensions and neighborhood type) are now compulsory. In map files it is possible to include several labels for each entry.

D.7.3 Purpose of each program

randinit.exe – This program initializes the reference vectors to random values. The vector components are set to random values that are evenly distributed in the area of corresponding data vector components. The size of the map is given by defining the x-dimension (-xdim) and the y-dimension (-ydim) of the map. The topology of the map is defined with option (-topol) and is either hexagonal (hexa) or rectangular (rect). The neighborhood function is defined with option (-neigh) and is either step function (bubble) or Gaussian (gaussian).

Example: *randinit -xdim 16 -ydim 12 -din file.dat -cout file.cod –neigh bubble -topol hexa*

vsom.exe – This program trains the reference vectors using the self-organizing map algorithm. The topology type and the neighborhood function defined in the initialization phase are used throughout the training. The program finds the best-matching unit for each input sample vector and updates those units in the neighborhood of it according to the selected neighborhood function.

The initial value of the learning rate is defined and will decrease linearly to zero by the end of training. The initial value of the neighborhood radius is also defined and it will decrease linearly to one during training (in the end only the nearest neighbors are trained). If the qualifier parameters (-fixed and -weight) are given a value greater than zero, the corresponding definitions in the pattern vector file are used.

Example: *vsom -din file.dat -cin file1.cod -cout file2.cod -rlen 10000 -alpha 0.03 -radius 10*

qerror.exe – The average quantization error is evaluated. For each input sample vector the best-matching unit in the map is searched for and the average of the respective quantization errors is returned.
Example: *qerror -din file.dat -cin file.cod*

visual.exe – This program generates a list of coordinates corresponding to the best-matching unit in the map for each data sample in the data file. It also gives the individual quantization errors made and the class labels of the best matching units if the latter have been defined. The program will store the three-dimensional image points (coordinate values and the quantization error) in a similar fashion as the input data entries are stored.

Example: *visual -din file.dat -cin file.cod -dout file.vis*

vcal.exe – This program labels the map units according to the samples in the input data file. The best-matching unit in the map corresponding to each data vector is searched for. The map units are then

labeled according to the majority of labels 'hitting' a particular map unit. The units that get no 'hits' are left unlabeled.

Example: *vcal -din file.dat -cin file.cod -cout file.cod*

D.7.4 Setup dialog box for SOMDB

The plug-in itself is a GUI, supplied for the ease of the parameters setup. During the setup user will be asked about the location of the database file (simple ini file). After that user has an ability to set up the parameters of the programs. Figure D.8 shows the setup dialog box.

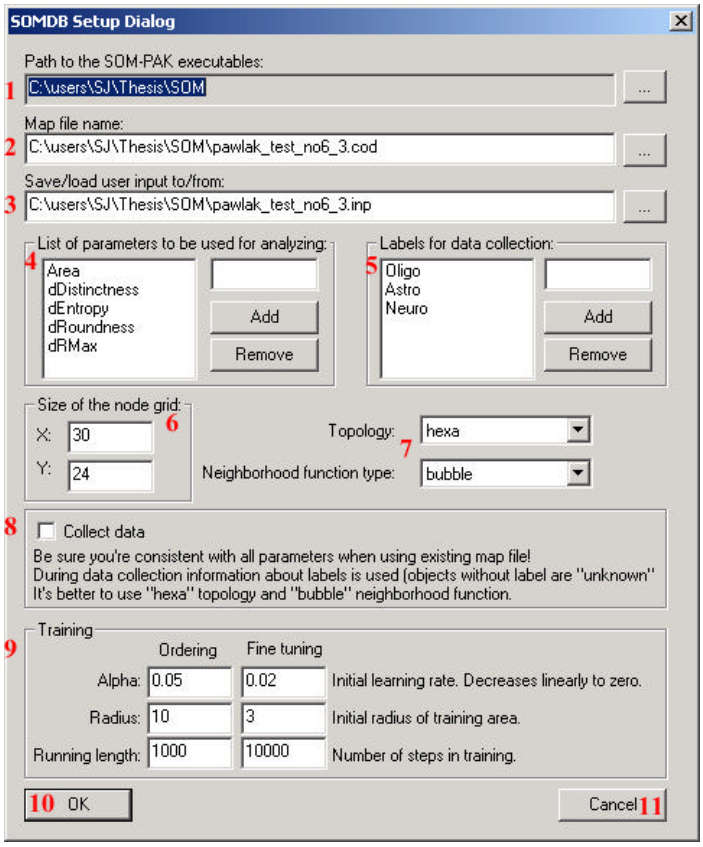


Figure D.8 SOMDB setup dialog box

Explanation of controls:

1. Path to the SOM directory, where all executable files are located;
2. Map file name to use as a parameter for the programs;
3. User input is saved to this file;
4. List of the parameters to create a vector;
5. List of labels to be used during manual labeling;

6. Size of the grid, the larger is grid the better coverage of the sample space is performed but more time is required;
7. Topology and Neighborhood function (see description of the SOM method);
8. If checked data will be saved; a user will be asked to make manual labeling and learning will be performed;
9. Parameters of learning (see description of the SOM method);
10. Saves the setup and closes the dialog box;
11. Discards the data and closes the dialog box;

D.8 Blob separation

This is an Object Separation plug-in, which uses a simple technique to segment images and locate objects in the images. It uses a threshold (this is the only value a user can adjust during the setup procedure) to make a binary image, and then connectivity is performed using this binary image. I used 4-connectivety [11], which looks for neighbors from top, left, right and bottom sides. Based on the result of connectivity different blobs are located in the image. Originally designed to be used for bacteria counting application.

D.9 Simple Rejecter plug-in

This is a Database plug-in, which makes a decision based on the area of the blob. Minimum allowed area is set during the setup. If object has the area less then defined, it will not be count as a valid object and will be removed. This plug-in requires Area plug-in to be used in order to have Area feature of the objects. The plug-in also calculates average area of the objects.

D.10 MorphologyOp plug-in.

This is a Feature extraction plug-in. The plug-in is capable of performing a sequence of erosions and dilations on the binary map of the object. A user can set the kernel of any size or shape. The order and the number of erosions/dilations are set during the setup. The same kernel is used for all operations. Figure D.9 shows the setup dialog box for this plug-in.

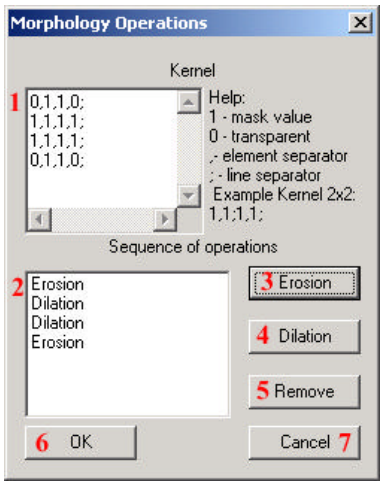


Figure D.9 MorphologyOp setup dialog box

Explanations of controls:

1. Kernel is a binary two-dimensional array. Elements of one line should be separated by the coma ",", lines are separated by the semicolon ";". Splitting onto lines is just for user convenience.
2. List of the operations to be performed on the object;
3. Adds the erosion to the list;
4. Adds the dilation to the list;
5. Removes the selected operation;
6. Saves the changes and closes the dialog box;
7. Discards the changes and closes the dialog box;

Each object is considered to be one blob. The sequence of the erosion and the dilations is performed on each object. Then connectivity is performed. If more then one blob is found, parameters of original blob are added to the new objects, and the original object is deleted.

One has to be careful when this procedure is performed. For instance, if Area was calculated and then MorphologyOp was executed, then new blobs would have Area value of the old object, but the actual area is smaller for each object.

D.11 AreaDivision plug-in

This is a Feature extraction plug-in. It goes through each object and if the area of the object is larger then a given value, procedure divided this object onto a set of the smaller objects with the given

average area. The old object is deleted. This produces new objects of the smaller size. New objects have the parameters of the original object. Area plug-in is required to be executed prior to this plug-in. This plug-in was designed for bacteria counting application. A user can set two values: the average area of an object, and the minimum area for division. When the area of the object is larger than the minimum area, then this object is divided by the average area.

D.12 Polar plug-in

This is a Feature extraction plug-in. It does not take any parameters. The plug-in makes the polar transform for all objects. The transformed image is saved in the Transformed member variable of the ObjectInfo class. After that any plug-in can use the transformed image. The plug-in was designed for automatic human brain cell recognition application.

D.13 SaveLoadObj plug-in

This is a Feature extraction plug-in. It can save all variables of the objects and variables of the CSJSpecimen into file. Later a user can load this information, without performing all operations again. I used this plug-in during the evaluation of the SOMDB performance. This plug-in must be included as the last one and once all cells where found, their parameters and locations will be saved. Then one could use just this feature plug-in to load all parameters and check how the database classifies them. This procedure does not require any processing. During the setup a user will be asked if he wants to load or save the information and the location of the file with information.

LIST OF REFERENCES

[1] Merriam-Webster online dictionary: http://www.m-w.com/

[2] A. J. Pawlak, "Automatic human brain cell recognition", Master's Thesis, September 1998

[3] R.O. Duda, P.E. Hart, D.G. Stork, “Pattern Calssification”, 2nd edition, Wiley-Interscience Publications, 2000

[4] William K. Pratt, Digital Image Processing, Second Edition, John Wiley & Sons, 1991, page 597

[5] Anil K. Jain, Fundamentals of Digital Image Processing, Prentice-Hall, 1989, page 342

[6] Erik J. DeGraaf, Personal Conversation, May 29, 1996

[7] Tomasz J. Motor, Machine Printed Character Recognition Using Multiple CMAC Neural Networks and Polar-Log Mapping for Feature Extraction, Master’s Thesis, 1995

[8] Mark A. Abbott, A study of Polar-Log Coordinate Mapping as a Preprocessor for Image Compression, Master’s Thesis, 1991

[9] T. Kohonen, “The Self-Organizing Map”, Proceedings of the IEEE, Vol. 78, No. 9, September, 1990, pp. 1464-1480

[10] T. Kohonen, J. Hynninen, J. Kanga, J. Laaksonen. SOM_PAK The Self--Organizing Map Program Package Version 3.1, Laboratory of Computer ad Information Science, Helsinki University of Technology, Finland, 1995

[11] R. C. Gonzales, R. E. Woods, “Digital Image processing”, Addison-Wesley, 1993

[12] Matlab software package the language of technical computing http://www.mathworks.com/

[13] Adobe Photoshop professional image-editing software http://www.adobe.com/