Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS

Jan Horáček

Segmentace a odhad hustoty ve stehennim kloubu

Segmentation and Density Estimation in Femur

Department of Software and Computer Science Education

Supervisor: Mgr. Lukáš Maršálek,
Study Program: Computer Science, Software Systems

2008

Most of all I would like to thank my supervisor and the co-author of our method *Mgr.Lukáš Maršálek* for his time, patience and advices. His involvement in our new method development was very significant and he gave me many very important hints that made the method possible.

Next I would like to thank *Mgr. Václav Krajíček* for introducing me into the medical segmentation problems and their solution and provided or pointed out most of the literature from which this thesis came. *MUDr. Martin Horák* provided me with a lot of medical information, prepared all the CT test data and commented the final application. *Mgr. Jana Timková* and *Mgr. Katarina Figurová* helped me with the insight into statistical data analysis and provided support when LaTeX stopped behaving nicely.

I would also like to thank my parents for supporting me my whole life, which is not always an easy task, and my friends at dormitory for their patience, friendly environment and their help in times the deadlines were approaching at a lightning speed.

I declare that I wrote the thesis by myself and listed all used sources. I agree with making the thesis publicly available.

Prague, April 11, 2008 Jan Horáček

Contents

List of Figures

List of Tables

Název práce: Segmentace a odhad hustoty ve stehenní kosti
Autor: Jan Horáček
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí diplomové práce: Mgr. Lukáš Maršálek
e-mail vedoucího: lukas.marsalek@mff.cuni.cz

Abstrakt: Existují dvě hlavní metody pro léčbu zlomenin krčku stehenní kosti - totální endoprotéza a spojení speciálním kovovým šroubem. Při dostatečně kvalitní kosti je preferován šroub. Ovšem pokud je příliš řídká, šroub se může proříznout kloubní hlavicí, což vede k dalším operacím.
Tato práce se zabývá novou metodou pro měření hustoty kostní trámčiny kloubní hlavice, které slouží lékaři jako další vodítko pro rozhodnutí, zda aplikovat šroub nebo totální endoprotézu. Náš přístup je založen na poloautomatické segmentaci kloubní hlavice z CT dat, založené na hledání optimální cesty v polárních souřadnicích na axiálních řezech, kde základ cenové funkce tvoří kombinace vlastností okostice, zejména směrový charakter 3D gradientů a velikost gradientů ve 2D řezech, kde tvoří typické "kanálky" . Pomocí vyplňování a morfologických operací je pak vytvořen objem, jehož vlastnosti jsou dále měřeny.
Výsledná implementace byla experimentálně ověřena na RTG klinice FN na Bulovce a umožňuje RTG specialistovi intuitivně, pomocí grafického rozhraní vytvořit přesný odhad hustoty kloubní hlavice v rozpětí 1 až 3 minut.

Klíčová slova: segmentace, femur, objemová data, CT, Dijkstra, polární souřadnice

Title: Segmentation and Density Estimation in Femur
Author: Jan Horáček
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Lukáš Maršálek
Supervisor's e-mail address: lukas.marsalek@mff.cuni.cz

Abstract: There are two basic ways of surgical treatment of femoral neck fracture - total endoprosthesis or a special metal screw in the bone. In case of a dense enough bone the screws are preferred. But for a too sparse bone, the screw may cut through the femoral head, which results in further surgical operations.
We present a novel method for femoral head density measurement, which serves as another hint for the doctor's decision, whether to apply a screw or total endoprosthesis. Our approach is based on semi-automatical femoral head segmentation from CT dataset based on finding optimal path through polar coordinates on axial slices. The cost function is based on a combination of corticallis properties, mostly the directional behavior of 3D gradients and their size in 2D slices, where they form typical "channels". The final volume is computed using filling and morphological algorithms and its properties are further measured.
The final implementation was experimentally validated on RTG clinic of Bulovka hospital and allows radiologists to intuitively and accurately estimate the femoral head density in approximately 1 to 3 minutes.

Keywords: segmentation, femur, volumetric data, CT, Dijkstra, polar coordinates

Chapter 1

Introduction

1.1 Femoral neck fracture

1.1.1 Old people

For older people, each surgical operation is a burden the body has great problems coping with. It may even be fatal for very old or sick people. Each operation may also result in immobility for the patient for some time and moreover the immobility itself may result in health problems. So it is necessary to minimize the number of surgical operations and guarantee their positive outcome as much as possible. Unfortunately these people are also the ones, who have accidents that may result in *femoral neck fracture*.

1.1.2 Femoral neck

The *femoral neck* is a very important part of the human skeleton. For the sake of mobility the femur is not a straight bone. The femoral neck displaces the femoral head several centimeters from the axis of femur. This enables us to move our legs very freely. Straight femur would be much more resistent to fractures, but would restrict the movement we are able to do. This way, we are able to walk, jump, run, kick and many other complicated movements, such as dance.

The actual reason why the femur (and femoral neck) has its shape should be better as a topic for discussion with a physician but we are more interested in the physical properties of this small part of the bone. It must transfer the whole body weight from the pelvis through the femoral head to the femoral

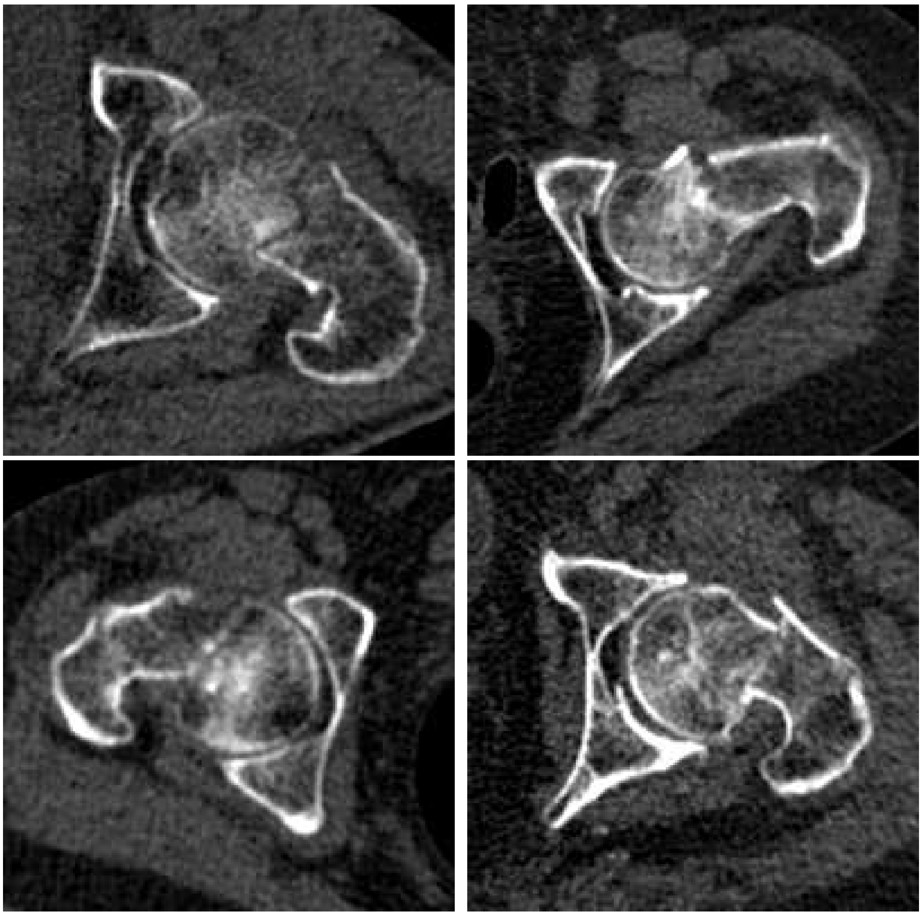


Figure 1.1: Examples of femoral neck fractures on axial CT slices. The most apparent result of a fracture is a discontinuity of the bone corticallis. Sometimes even discontinuities in the trabecular bone are clearly visible.

body and then to the knee. Femur itself is the biggest and strongest bone of the skeleton and we may say that the femoral neck is the weakest part of this bone.

The inner part of the femoral neck and head is a trabecular bone that has a very delicate internal structure to endure the lifetime burden and to stay light at the same time. A young and healthy person is not endangered by a fracture in this part of the body, unless subjected to very intensive stress such as some very serious accidents (more information on the human pelvis and hip joint can be found in any text book for medical students, for example [1]). But as the age progresses, the bones start to loose their strength. Very old people, people with bad eating habits and people with genetic dispositions or a disease that may affect the development of osteoporosis are all endangered with femoral neck fracture.

It is enough to fall on a slippery pavement and if the person is unlucky, he or she suffers a femoral neck fracture. This is a serious matter, because the accident can happen unexpectedly, the correction is very difficult and the healing process lengthy and painful.

There are basically two methods for femoral neck fracture treatment. Both are difficult surgical operations. One is *total endoprosthesis* and the second is fastening the broken bones together with a special *metal screw.*

1.1.3 Total endoprosthesis

Total endoprosthesis is a temporary solution. It usually takes about five to twenty years for the endoprosthesis to get loose, or become unusable by other means for example by interaction of increased number of friction particles with biological tissue, by dislocation, etc. [5]. The accident risk is increased with younger age, higher activity, male gender, overweight or femoral head osteonecrosis of the patient. Even when it is well done and the person is using it according to doctor's advice, it is rarely a solution that enables a person to painlessly walk till the person's death. It is the same as with many other artificial additions to human body.

When a person walks (or even jumps), the whole body weight is resting on the femoral neck, so when we replace the original bone with something artificially fastened to the rest of the bone, it is obvious that it takes considerable care to make it as good as the solution nature blessed us with. The endoprosthesis is usually cemented inside the hollow part of the bone. So if the cement breaks or the femoral bone is torn apart, the person is unable to walk anymore.

1.1.4 Screw

Another solution to femoral neck fracture, which is nowadays very popular, is screwing together the two fractions of the femoral bone (femoral head and the body of the femur) with a special *metal screw.* There is a variety of available screws. Technically all consist of two parts: one is a metal rod fastened from the side with several small screws to the femoral body, the second is a quite large screw that goes through the rod inside the end of the femoral body and continues to the femoral head. The angle between these two parts is usually 130 degrees. Newer rods have an addition: a pike at

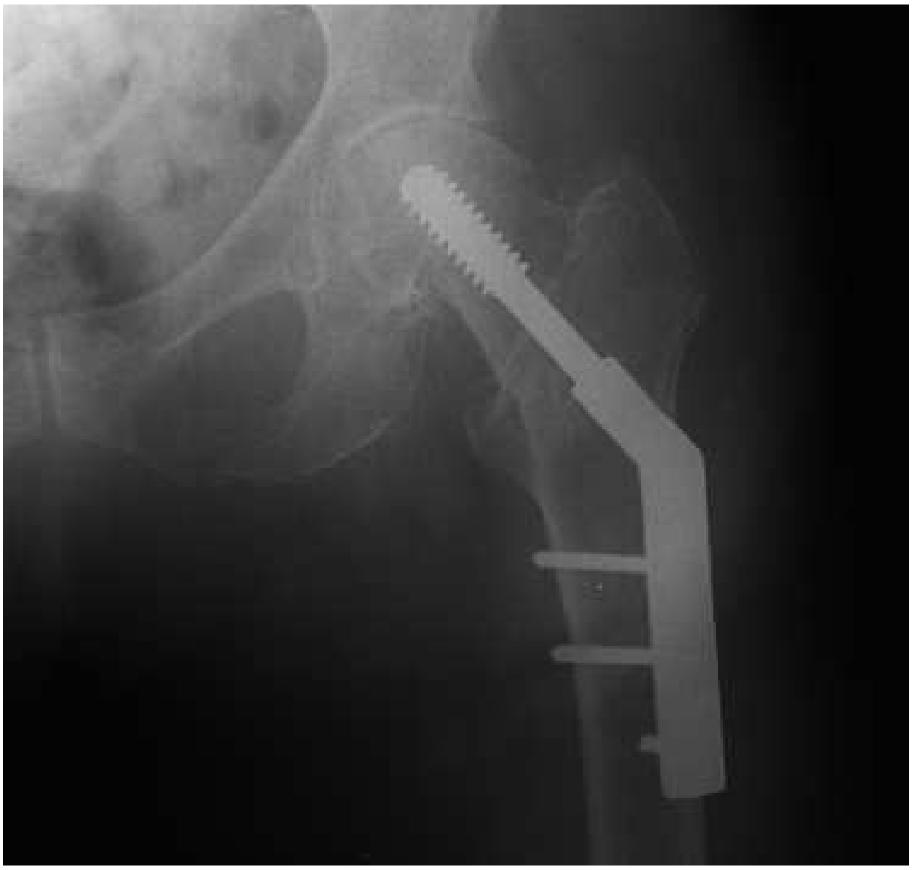


Figure 1.2: Example of a special metal screw, a post-operative CR image.

the lower end of the rod that is inserted in the hollow part of the femur. It helps the rod to fasten as much as possible to the body of the bone.

The screw is in a stationary contact with the rest of the skeleton, it only acts as a complementary addition to the femur, which helps the femoral neck to spread the force inflicted by the body weight during walking to a larger area of the femoral body. So the joint (the acetabulum and the femoral head) remain the same. Human technology is not yet able to make a better part than the human body itself, the original joint and ligaments are still better than anything we are able to artificially manufacture.

However, the screw thread is quite small, about 1cm wide and several centimeters long (usually 2 to 4cm). So the force which is inflicted by the body weight must be transferred by this part of the screw to the rod. If the spongy bone in the femoral head is too sparse, it may happen that (during walking or other physical activity) it could begin to cut through the femoral head and finally destroy it. That would result in another surgical operation and replacement with total endoprosthesis (if applicable).

1.2 Femoral head density estimation

There are several ways to help the doctor decide, whether to apply a screw, or directly total endoprosthesis. As to this date the best seems to be a *survey at the rheumatologist* combined with a *computed tomography examination of vertebra for osteoporosis.* That gives the best insight to possible problems and patient's suitability for undergoing either total endoprosthesis or screw surgical treatment.

We are looking for a new hint that would with certain probability predict the suitability of either surgical treatment. Even though the method for examining osteoporosis in vertebra is already in common use, it only gives us a rough estimation of global osteoporosis of the patient's skeleton, but not actual state of the femoral head. So the basic principle that comes into mind is to take this examination locally to the femoral head. A simple explanation of our method would be:

1. segment the data from a CT scanner and find the femoral head
2. perform some sort of quality analysis of the segmented spongy bone

The second step may be simplified by some sort of statistics, that will tell us the mean value, standard deviation, histogram, etc. However, the first step is a nontrivial one and will be the topic of this diploma thesis.

1.3 Computed Tomography

The data acquired from computed tomography is volumetric. It consists of a number of slices, each usually with square dimensions, the resolution may vary between 256x256 to 4096x4096 voxels per slice. Medical CT scanners produce usually slices with resolution 512x512. Higher resolutions are acquired only in industrial environments and the needed radiation exposure is not suitable for (or bearable by) a human body.

Each voxel represents a block of volume, one side is usually square, with dimensions around 1mm (may be from 0.5mm to 2mm, depending on the field of view selected by the radiologist during the examination). The "length" of the block is called *collimation* and represents the distance between two slices. Usually it is not larger than 5mm, today it is possible to set collimation to a value close to 0.5mm, so the voxels are almost cubic.

The value of each voxel is given in *Hounsfield units*. Hounsfield units are a special value defined for the ability of matter to absorb X-Rays. The lowest value is defined as -1000HU and represents air (absorbs minimal amount of radiation), 0HU is water and +3000 is almost total absorbtion, which is for example in metals. Human body tissues have values around 0, for example fat is -120HU, muscles +40HU and bones may wary from +100HU to +500 or even +800HU. It is also worth noting that the CT data is quite good for bone recognition or complete organ recognition, but not good enough to provide detailed information about the structure of individual organs, muscles or ligaments. The slices are saved in DICOM format [3] as 12-bit per voxel values.

The slices may overlap, normally they overlap only by a small amount, but it can be even more than half of the collimation.

We would like to have as narrow slices as possible, because for example if we have collimation 5mm, the gap between acetabulum and the femoral head may disappear completely – its width is usually 1mm to 5mm. On the other hand, very narrow collimation introduces a lot of noise in the image. For us the narrow collimation is much better, because we are able to filter the noise but we are unable to recover information lost due to long voxels.

Also, we usually have no chance to take another CT scan of a person, if the one we are working on is too noisy or unclear. One CT scan of human pelvis is hundreds of times more intensive than a simple x-ray image. It is desirable to minimize the exposure of the patient to x-rays. We have to work with what we get the first time, so a robust technique with possible user interaction is needed.

A nice introduction to the function of computed tomography can be found in [7].

1.4 Summary

Femoral neck fracture is a serious accident which results in a complicated surgery operation. The two possible solutions are total endoprosthesis and fastening together with a screw. Both of these operations are heavy-duty and if done inappropriately or to a unsuitable patient, it may begin where it started or even worse and burdens the patient extensively.

Any hint that would help the doctor decide which type of treatment to apply is appreciated. With the expansion of computed tomography availability, the CT inspection is done anyway to see how bad the broken bone is. So our additional processing of these data is almost *free of charge* (from the patient's burden point of view). It can be done in a matter of minutes including data loading and correct segmentation, so even the radiologist's time is spared.

But in the end this insignificantly looking examination may save a lot of trouble to the patient and the doctor.

1.5 Goals of the thesis

The goal of this thesis is to provide a theory for a novel method for femoral head density estimation and implement it in a working application. This method would help a specialist decide which surgical treatment is better for a patient, whether to use total endoprosthesis or a special metal screw.

The method should be based on computed tomography data of pelvis and its surroundings. The result should be given as a mean value and a standard deviation of the segmented femoral head excluding the femoral corticallis, both given in Hounsfield units. As an addition, histogram of the segmented data may be shown as well. A specialist should be able to use this application to segment the femoral head as quickly as possible and use the results in further analysis.

1.6 Structure of the text

The following *Chapter 2* is a short overview of the most relevant methods used in medical and CT data segmentation. A short introduction to each of them is given.

Chapter 3 is an introduction to our approach of the femoral head segmentation. The method is discussed from a broader perspective from the point of view of the data properties and a brief discussion of bone quality analysis is given.

Chapter 4 gives a detailed explanation of the theoretical background of our preprocessing step. Both techniques and parameters of those techniques are discussed.

Chapter 5 gives an explanation of the segmentation step performed on preprocessed data. The actual finding of the femoral head corticallis is described, as well as the final postprocessing steps to obtain the whole volume.

Chapter 6 presents the structure and ideas behind the actual implementation of our method into a fully working application. The focus of the description is on the structure of the program and logical connections between different components. A detailed explanation of individual methods is done only for the most important ones, the rest are either self-explanatory or commented in the code.

Chapter 7 is a summary of various measurements done with our application. Also some possible utilizations are proposed.

Chapter 8 is a final summary of the whole thesis and also gives some hints about possible future research and optimization done on our method.

Appendix A is a description of the application used for our segmentation algorithm from the point of view of a user or radiologist that wants to work with it.

Appendix B is a short and simple introduction to show novice users the way our segmentation is done in our application.

Chapter 2

Background research

2.1 Medical segmentation

An extensive research has been done in the area of image processing and computer vision, supported by a solid theoretical background. But most of the work was done in the area of 2D image processing and 2D pattern recognition. A number of techniques exists, each of them is good for something else. The needed basis for any image processing including some elementary mathematical background may be found in [8], image segmentation is mentioned in [8, pages 689-794]. The basics of 2D image processing with implementation examples in *MATLAB* system can be found for example in [9].

However, the CT data are volumetric, so we need some other means than 2D image segmentation, because the result is also in 3D. A simple overview of techniques used during medical data segmentation is in [14] and [19].

A nice classification based on approach is given in [14]. It consists of three main groups of methods, most of which can be used directly on 3D data or can act as a preprocessing step on them.

Structural techniques – these methods are based on analysis of the structure of the region, which results in segmentation

Stochastic techniques – the whole decision is based only on a per voxel basis or a local neighborhood, no other information about the structure of the region is applied

Hybrid approaches – have characteristics of both structural and stochastic methods

Basically any complete medical segmentation usable on real world data consists of at least two steps:

1. Preprocessing
2. Segmentation

Preprocessing usually consists of one or several image enhancement methods, which bring the usually noisy and damaged real world data closer to something more suitable for mathematical description. The knowledge needed for the preprocessing method design or selection is an analysis of how the data can be damaged and what can be done to reconstruct them. The following segmentation is then based more on the knowledge what exactly the data *should* look like and how to mathematically model the real world phenomena.

A short list of problems that can be encountered during the femoral head segmentation from CT data is discussed in [17]. We have solved some of them (such as defining the intact part of the bone without fracture) during our preprocessing step, but some remain and will be discussed in this thesis as well (such as the structure of the trabecular bone and its relation to the structure of the femoral head corticallis).

There are many different segmentation techniques, some are good for almost any type of data, some are more sensitive to the type of data processed. In medical CT data segmentation, there are a few techniques, that can be used or have been described as useful. A basic and very simple technique usable almost anywhere with more or less successful results is *thresholding*. More elaborate technique, but still quite simple is a sort of *edge detection*, which is not usable by its own, but is important as a preprocessing step for many other techniques.

As we dig more deeply into the problematics of medical CT data segmentation, a fact about the structure of the data cannot pass by unnoticed. The human body has usually more or less the same structure in regard of the relative position of different organs. Also the shape of those organs in the body of a healthy person is very similar. So using this knowledge, we can try to use techniques such as various *deformable models*. Techniques directly using the information about the structure are called *atlas based segmentation*, which try to fit a known shape to the given data.

Some of the basic segmentation techniques are discussed further in this chapter.

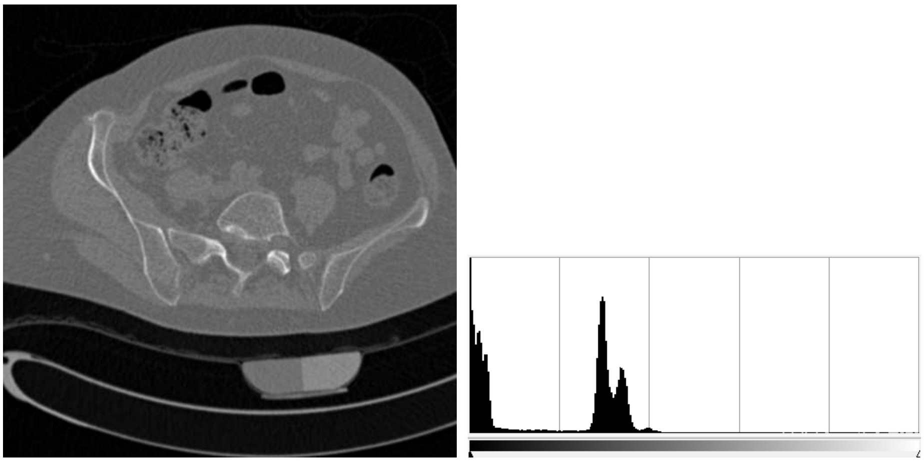


Figure 2.1: Example of a CT slice and its histogram. (*Note:* the image has been enhanced for viewing purposes, histogram represents original 12-bit data.)

2.2 CT data properties

Let's analyze the actual data properties, so that we would be able to choose the right method. We are interested in two properties:

1. Absolute values
2. Spatial characteristics

On fig.2.1 we can see the *absolute values* and amount of each of those values. In the histogram, you can see two major peaks. The first peak represents air and other gasses. The second peak represents the mass of the body and can be further subdivided into three parts. The biggest part (with the smallest density values) represents fat and softer organs, the second part represents mostly muscles and denser organs and the third small (barely noticeable) represents mostly parts of the examination table, but also the most dense organs and some softer bones. Unfortunately, bones themselves do not form a noticeable peak in the histogram, their values are spread over a large area that begins somewhere in the second major peak (somewhere between the second and third mentioned part) and go on far beyond this peak into higher density values.

Fig.2.2 shows a magnified part of our interest, i.e. femoral head. A line has been drawn over this part (shown in cyan) and the absolute values encountered along this line are drawn in a graph on the right side. Here we can examine the spatial relations between the parts of the body we want to

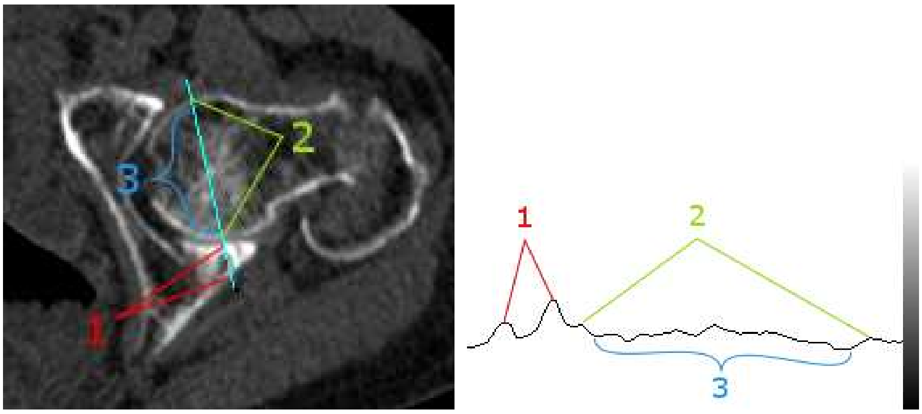


Figure 2.2: Example of a CT slice and a 1D cut through this slice. (1) pelvis, (2) femoral head corticallis, (3) femoral head trabecular bone (*Note:* the image has been enhanced for viewing purposes, graph represents original 12-bit data.)

segment. Here we can see that (1) the pelvic corticallis is very significant in the data, but (2) the femoral head corticallis has a tendency to disappear in the structure of trabecular bone (3) inside the femoral head.

If you look only at the graph and hide the description, you can notice that it is almost impossible to tell where the femoral corticallis is. The trabecular bone has everything more intensive than the corticallis. That means local maxima and minima, both larger and smaller features, it is locally both denser than the corticallis, but in other places less dense than the surrounding organs. Another difficulty is the pelvis being the major feature in this cut and presenting all features of our femoral head corticallis, but in a much stronger way. So it seems that segmenting femoral head corticallis from both pelvis and trabecular bone would be a very difficult problem.

Fortunately, even though the trabecular bone has a quite significant internal structure, it is still more random than the corticallis. This implies that a method for segmenting this part of the human body would have to exhibit both structural and stochastic features, either together at the same time, or separated into several phases.

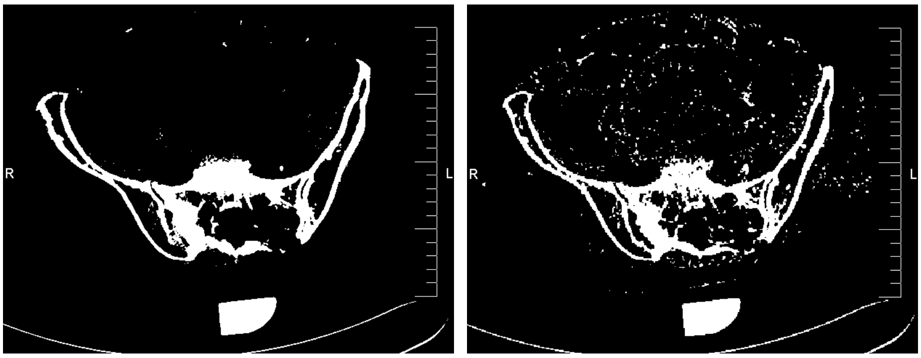


Figure 2.3: Thresholding CT slices of pelvis. Threshold 70, left image is without noise, collimation 5mm, right image is noisy, collimation 0.75mm

2.3 Current medical segmentation methods

2.3.1 Thresholding

Probably the simplest method for segmentation. This method works without problems on both 2D and volumetric data. We can classify it as a purely *stochastic* technique. The input from user (or other algorithm) is one or more *thresholds* which define intervals of values that are used to assign voxels to different classes. Defining only one value makes a *binary* segmentation of volume. A survey of different thresholding techniques is in [22].

Thresholding without additional advancements is very sensitive to input data quality, for example *noise* may add very disturbing artifact, as can be seen in fig.2.3.

Another disadvantage is sensitivity to the selected threshold. Thresholds with very similar values do not necessarily end up with similar segmentations. Sufficient results are obtained only on images with good contrast between areas of interests that should be segmented into different classes. But even then it is important to choose the threshold wisely.

For example for our CT data (fig.2.1) we can define a threshold somewhere between 70HU and 150HU. Using everything above this threshold and discarding everything below will give us a rough human skeleton. Of course, it may omit some sparse bones or add some dense organs, but is enough for a rough bone localization. If we want only an interval (for example most of the soft tissues in the body), we can choose two thresholds, such as -150HU as the lower threshold and +80HU as the higher threshold. This will omit most of the bones and harder tissues together with air and other gasses.

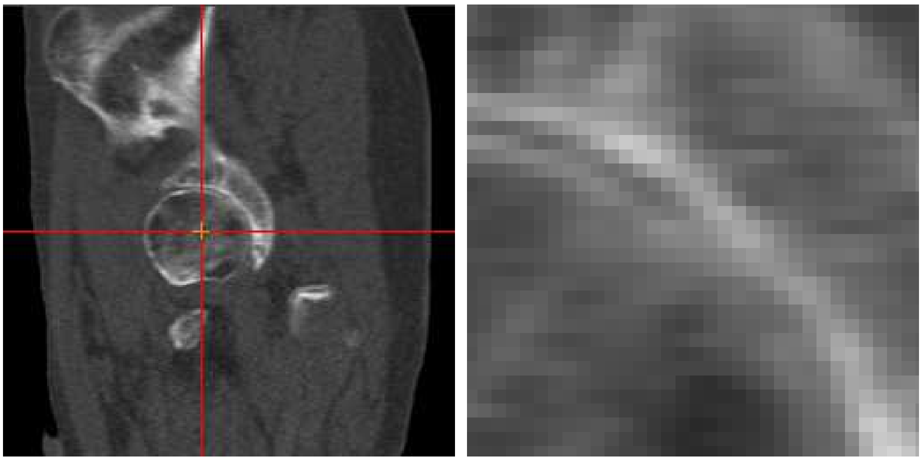


Figure 2.4: Sagittal cut through a femoral head and a magnified part.

Automatic or semi-automatic algorithms for threshold selecting usually derive the input value from histogram. Histogram properties have been discussed in Section 2.2. These histogram based algorithms either utilize some additional information about estimated histogram layout (such as the description in Section 2.2) or perform some general histogram analysis (for example try to represent the histogram as a combination of gaussian curves and take the major gaussian curves as representations of significant features of the image).

For a detailed thresholding overview, see [23].

More advanced techniques utilize also other information, for example a nice technique implementing thresholding together with segments merging is given in [29]

For the purpose of segmenting femoral head, thresholding is not a good choice, because it ignores the structure of the bone. The biggest problem that we face is a clear segmentation of pelvis and femoral head. Pelvis is usually (at least older people have it this way) much denser than the worn down femoral head, sometimes we even cannot see where the corticallis of the femoral head is (fig.2.4). Thresholding may show us where both of the bones are, but is unable to separate one from the other.

2.3.2 Isosurfaces

Isosurface is a surface placed at the points in space, in which the values are equal to input isovalue. We can imagine it as the border of two classes

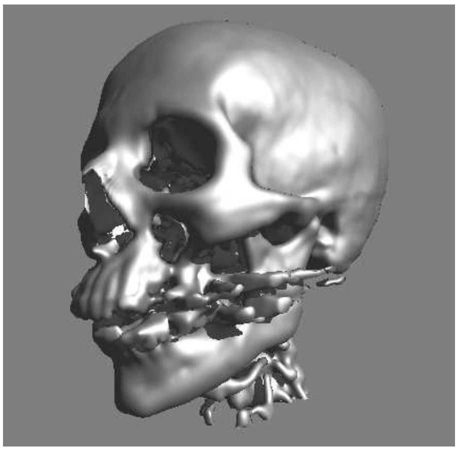


Figure 2.5: Example of an isosurface.

after binary thresholding. This method in its basic form is usually good for visualization, but is not good enough for practical utilization in medicine segmentation.

However, we can enhance this technique with different algorithms that track the evolution of curves in the volume. That gives a smoother result than thresholding and even some sub-voxel shape estimation.

The evolution of isosurface may be tracked by numerical techniques such as placing a *set of marker points on the evolving surface* and then changing their position according to the moving surface. Another technique that exploits a strong link between moving surfaces and computational fluid equations is *level-sets* (more on level-sets in [14, pages 9-11]). But this is more a topic for *deformable models* discussed in Section 2.3.6.

Even though isosurfaces together with some evolution tracking algorithm already belong to the *structural* techniques, they exhibit very similar properties to thresholding. They are too dependent on the absolute values of voxels, which vary greatly in the femoral head. The same reasons apply to why we cannot use this technique, even with its enhancements.

2.3.3 3D Edge detection

Edge detection is the basic method for image processing in the same way as thresholding is the basis for segmentation. Edge detection in its purest form does not segment anything, it only acts as a preprocessing step for further manipulation. We can of course interpret the result of edge detection as a segmentation of image into two classes – edges and homogenous areas. In this sense edge detection may be regarded as a *structural* technique.

Edges are a discontinuity in image intensity. This implies that the more contrast between the areas of interest we have, the better the result is. On the other hand, edge detection is usually very sensitive to noise.

A number of techniques for edge detection in 2D has been described. For example gradient operators Roberts, Sobel, Prewitt (see [9, pages 707-712]) or the Canny edge detector [9, pages 719-725] . They are mentioned in every image processing textbook. For example Roberts operator, in Liu [15] may be extended to 3D as well.

Usually edge detection is used either as a preprocessing step or for correcting results from another method. A method that uses edge detection (*Canny's edge detector*) for modifying results obtained by region growing is described in [27].

A simple edge detection is not usable for our purpose, because the spongy bone inside the femoral head is not homogenous and introduces false boundaries. But if we can somehow enhance the edge for corticallis and suppress the internal trabecular bone, that may be the way for our segmentation. In truth the method described in this thesis is based on enhanced edge detection algorithm.

2.3.4 Atlas based segmentation

Atlas based segmentation is a *structural* technique that merges image segmentation with image registration [16].

A basis for this segmentation is a so called *atlas*, which is a prototype of one or more model segmentations done by a professional. The prototype dataset is iteratively deformed to match a target dataset. If we have more prototypes, we try to select the most suitable (that means the one we need to deform the least to register it on the examined data). When the prototype is selected and registration is finished, we segment the examined data the same way as the deformed prototype (the same deformation is applied on the segmentation labels). An application of several different registration approaches with multiple specimen is in [21].

When the prototype set is well chosen, the results from this segmentation are usually very good, because we utilize very much information about the structure being segmented. This information obtained from a specialist is stored explicitly in the prototypes in the form of the manual segmentation.

On the other hand this method is very intensive on the preparation of the *atlas*. The specimen must be properly selected and precisely segmented, usually by hand. That may take several weeks or even months only for the preparation of a single application. Also the complexity of the deformation has to be nontrivial in order to deform the prototypes properly. That means that the overall time complexity is also very high, because registration with nontrivial deformations take a lot of computational resources.

A femoral head segmentation method that is based on a prototype registration is described in [18].

We did not use this algorithm because there is a need for an extensive specimen preparation done by a professional. We wanted to find an algorithm that would allow us utilize the information about the structure of the segmented region right in the code and no additional data would be needed. Also some of the patients have their hip joint dislocated from the accident and that prevents us from using atlases with pelvis – those would be the most beneficial for us in case of normal joints, because the information about the pelvis would bring us the most advancement over our current method.

2.3.5 Region growing

Probably the simplest *hybrid* approach to segmentation. We need two inputs for the algorithm: a *seed* and a *homogeneity criterion* [10]. The algorithm starts from the seed and adds other regions around it as long as the homogeneity criterion is satisfied. However, region growing may be quite sensitive to noise or discontinuities in the object boundary. An example of a medical segmentation with adaptive region growing is in [20].

Region growing in its simplest form is unusable for our purpose because of the structure of the femoral head. We need to find either only the trabecular bone or the complete femoral head and then remove the corticallis manually. But the structure of the trabecular bone is in this sense quite similar to the structure of the corticallis, so we would either be unable to find the corticallis, or lose some parts because the trabecular bone would look like the corticallis and prevent us from reaching to the whole volume of the femoral head.

What we also tried was using this technique on data much more suitable for this algorithm. We took the data from our preprocessing step in Chapter 4, adjusted them and used several thresholds to fill them from the seed point given by user as a point lying on the femoral head corticallis. But it turned out that either we cannot get the whole corticallis or the al-

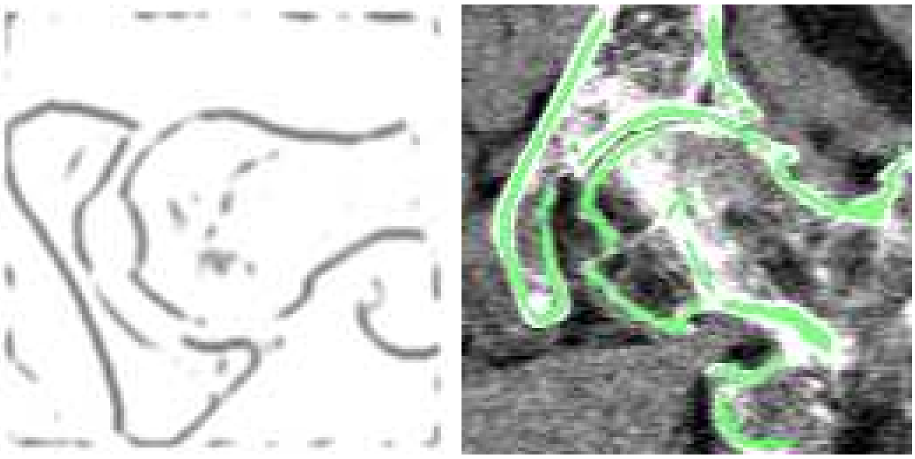


Figure 2.6: Example of region growing on our preprocessed data. Left image is one axial slice from our thresholded and adjusted cost function from Chapter 4, right image is one frontal slice from the original CT data with green mask from region growing on adjusted cost function.

gorithm *leaks* into other parts of the data. Unfortunately, mostly happen both of these effects at once unless we use very high or very low threshold and we were unable to find a suitable threshold for one dataset that would prevent both of these cases to happen, less so to find a universal threshold or algorithm for finding one for any data.

On fig.2.6 you can see the results of our tests. Even though the cost function seems to be suitable for a filling algorithm as seen on the left image, on the right image with mask there is a clearly visible place in the upper part where the algorithm has a tendency to *leak* into other parts of the skeleton (in this case pelvis, but similar places for leaking into the trabecular bone may be found as well).

The structure of the data implies that we would need a very sophisticated criterion for defining the trabecular bone, because as we have seen the method is unsuitable for finding the corticallis even after extensive enhancement. But finding this criterion would not be an easy task, the trabecular bone is quite hard to describe in a simple mathematical equation.

2.3.6 Deformable models

Deformable models are usually some solids, curves or surfaces defined within the image and are being deformed according to applied internal and external forces. External forces are from the data being segmented and try to deform the object so that it moves towards the data. Internal forces keep the object smooth.

Deformable models have been popularized for image processing in [26]. They have also other uses, for example in computer graphics. Deformable models for image processing can be divided into three groups: *energy minimizing snakes*, *dynamic deformable models* and *probabilistic deformable models*.

Snakes [11] are the most popular of the three deformable models groups. They are planar contours trying to minimize the weighted sum of internal energy (tension or smoothness of the curve) and external energy (defined over the whole volume domain, usually with local minima around edges).

Dynamic deformable models (mentioned in [14, page 7]) is another approach, that constructs a dynamic system governed by a functional that is being evolved into equilibrium between external and internal forces. These models are able to segment not only static shapes, but also shapes evolving through time.

Probabilistic deformable models incorporate also the information about the estimated shape of the object in terms of probability distributions. This method also provides a measure of uncertainty of the estimated shape parameters after the fitting has been computed [25].

A thesis focused especially on deformable models and their practical usage in medical volume data segmentation is in [12]. Measurement of internal organs volume based on deformable models is in [13].

Deformable models was one of our options if the method we have implemented for segmentation showed up that the shortest path search algorithm does not work well. The preprocessing step would stay the same, but the segmentation itself would be probably done with *snakes*. We went the way of the simplest algorithm possible and it showed up that our shortest path search algorithm performs well enough on the femoral head corticallis segmentation. The deformable models would need a much more sophisticated implementation.

Chapter 3

Method Overview

3.1 Problem

The problem being solved in this thesis already mentioned in *Introduction* is stated as follows:

Problem 1 *Let's have a CT scan of pelvis, where one of the femoral necks is broken. Let's assume that it is broken somewhere beyond the end of femoral head, so that the femoral head is as complete as possible. Set a user-defined plane that ends the non-broken part of the femoral head. Segment the spongy bone bordered from one side by corticallis and by user-defined plane from the other side. In this segmentation, we want* only *the spongy bone, we do not want the corticallis to be included. From this segmentation, compute the mean value and standard deviation of the voxels that represent the spongy bone.*

An example of what we want to actually get from the segmentation algorithm as a result can be found in fig.3.1.

3.2 Real world data

First we look at the shape of the bone to be segmented. In an *ideal* case, the femoral head is almost *spherical*, from one side it is connected to the femoral neck and from the other side there is a small depression. Most of the volume is composed of *trabecular* bone usually with an apparent internal structure. The border of the bone is a little bit denser cortical bone. The

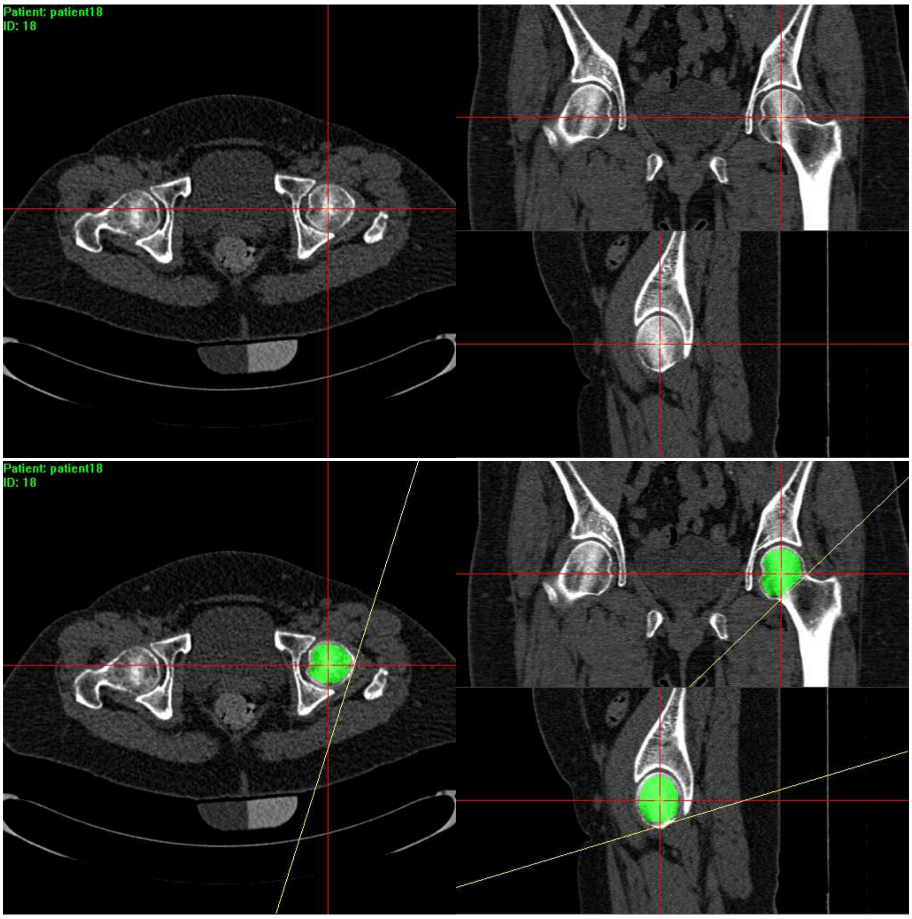


Figure 3.1: Example of a correct segmentation result.

pelvic acetabulum, where the femoral head is mounted to pelvis, has some very similar properties, also with quite strong cortical bone and has almost the same shape as the spherical part of femoral head.

However, this is true only in an ideal case. Real world data may be quite different in several ways. The femoral neck fracture is usually diagnosed to older patients, who had an accident (usually on some slippery surface, a critical time of the year is the whole winter, especially the first snow). These patients may have osteoporosis in different stages of acuteness, so the bone quality is usually decreased and some parts of the bone may be less apparent on the CT scan. Also the corticallis is worn down and its density may be much lower than for example the density of the trabecular bone inside the femoral head.

The *shape* of the joint is not directly related to age, but of course it may be a little affected. Real world pelvic joints may have a shape somewhere between a perfect sphere, to something that with a bit of imagination may look like an ellipsoid with sharp edges and dents. The small depression on the medial side that is inserted into acetabulum is usually quite clearly visible with all of the patients.

The *size* of the bone is around 4cm, with a tolerance of ±2cm.

The fracture does not alter the shape of the joint part of the femoral head, because this surgical treatment using a screw is only applied to patients, where the fracture is in the femoral neck or further in the femur and the femoral head is almost intact.

We can assume that the femoral bone to be segmented is almost convex volume with almost smooth surface (however, there can be dents and also the small depression should be correctly segmented). The overall shape is close to a deformed sphere. The corticallis has a locally higher HU values than its close surrounding, but the trabecular bone may have it a little higher and the pelvic acetabulum may have it very much higher (and it is very close, so it may mislead the segmentation).

3.3 Solution approach

We try to present a solution as simple as possible, both from the implementation point of view and from the theoretical point of view.

We are looking for a method that would be able to find the corticallis of the femoral head with little additional information other than that it is a deformed sphere and it is locally more dense than its close surroundings.

The method presented here focuses only on segmentation, bone quality analysis is given just as an example. We divide our method into these parts:

1. Preprocessing
2. Segmentation
3. Bone quality analysis

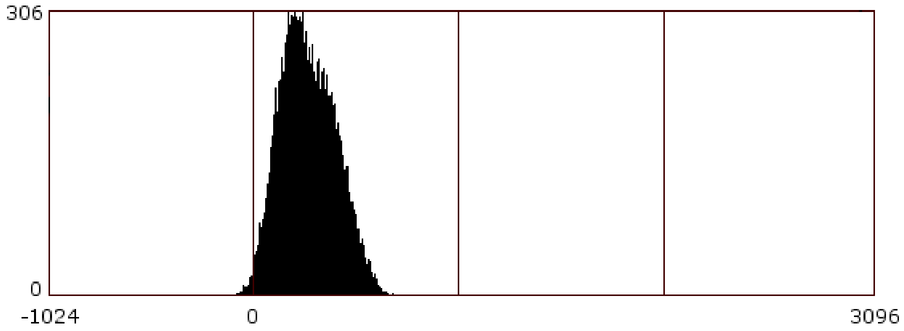


Figure 3.2: Histogram of segmented data in fig.3.1. Horizontal axis is in Hounsfield units, vertical axis shows quantities. The result mean value was 265.02HU, standard deviation 131.75HU.

3.4 Bone quality analysis

Because *bone quality analysis* is not the primary goal of this thesis, let's discuss only the most basic analysis. There are three very simple possibilities how to characterize the segmented bone:

- Mean value
- Standard deviation
- Histogram

We can compute all three with little effort. Mean value and standard deviation have an important attribute that they are both scalar values, so they can be used as a very simple indicator. The disadvantage is that many different bone structures may have the same mean value and standard deviation (both good and bad bones), so it can be also a little bit misleading. However, it is probable that very bad bones affected by osteoporosis would have also lower mean value. This way we would have an indicator that would warn us about weak bones, but it will not guarantee us that the bone would be good.

Histogram (see fig.3.2) shows us more information about the density properties of the bone, but still does not contain any information about the actual structure of the bone. Still, if there could be found some metric for these histograms that would decide whether the bone is good or not, it would bring considerable advancement to this theory. Histogram itself is a 1D function that is not easily compared and measured, at least we do not know of any suitable method for bone quality analysis. Maybe good bones exhibit certain features in the histogram, but this is still a topic for further research.

At the time of writing of this thesis we compute all three properties of the segmented data, but for the actual bone quality analysis, we use only the mean value and standard deviation, histogram is there only for research purposes and does not participate in the final analysis.

No decision is done in the program, the application only tells raw results to the doctor and it is up to his evaluation abilities to decide what is the estimated bone quality.

Chapter 4

Preprocessing

Preprocessing prepares the original CT data for us. Real world data are burdened with hight levels of noise and are (as already mentioned several times) not very clear in the case of old patients. We perform several steps, which are all *stochastic*, all performed the same way on each voxel and do not take into consideration the bone as a whole.

We want to enhance the corticallis as much as possible and suppress the trabecular bone inside, so that the femoral head corticallis would be a dominant edge in the final image.

4.1 Removing noise

CT scans with very small collimation (in our case 0.6mm – 0.8mm) have quite a lot of *noise*. For simplicity we can assume that this noise is an *additive Gaussian white noise*. It turned out that for removing this noise it is enough to use a lowpass *Gaussian filter* with convolution mask size 5x5x5. The discrete approximation of Gaussian filter is separable, so it is very fast to use on 3D data. The mask size of 5^3 is enough for removing most of the noise, but on the other hand it does not modify the visibility of corticallis very much.

Some of the other noise-removal algorithms can be applied as well, but usually the time complexity is not very good for processing 3D data. Here the separable Gaussian filter serves us well.

4.2 The cost function

The cost function should represent the CT data in a way, that the ridge of the corticallis should be around zero, other parts of the body should have higher values. In close neighborhood of the corticallis there should be nonzero values that would define the corticallis as a *channel*.

A short overview how we will build the final cost function follows:

1. Compute *directional corticallis enhancement* functions c_X, c_Y, c_Z, that utilize our model of how the corticallis *should* look like in a 1D cut. Detected corticallis in given direction is represented with higher values than its surrounding.
2. Compute *complete corticallis enhancement* function $c(x, y, z)$ by taking the maximum of $c_X(x, y, z), c_Y(x, y, z), c_Z(x, y, z)$ for each point (x, y, z). Again, the corticallis is represented with locally higher values.
3. Compute function $d(x, y, z)$ as a clamped and lowpass filtered linear transformation of corticallis enhancement function.
4. Compute *final cost* function $e(x, y, z)$ suitable for graph search algorithms as a clamped linear combination of function $d(x, y, z)$ and the magnitude of the 3D gradient vector.

4.3 Gradient component

The first idea for enhancing the corticallis was some sort of 3D edge detection. It turned out that simple edge detections (as described in Section 2.3.3) based on 1^{st} or 2^{nd} order derivatives are not enough, because even after lowpass filtering it enhances the trabecular bone and pelvic acetabulum more than the corticallis. This confuses the segmentation algorithm and the resulting segmentation is usually not very good.

However, the *gradient* brings some important information, that we can make use of. For example the magnitude of gradient grows on both sides of the corticallis, while keeping the center around zero. The same happens with pelvic acetabulum. Also the direction of the gradient is helpful, because that can give us a hint on which side of the corticallis we are.

Our method utilizes the information on how the femoral head corticallis looks like. The width is usually 2–4 voxels and the density is decreasing on both sides. The idea is to look on both sides in each of the 3 axial directions of the actually processed voxels and search for gradients, that are as big as possible, with opposing directions pointing as much as possible towards each other.

4.3.1 Enhancing corticallis

We consider a voxel at position (x, y, z). Its value is:

$$value = f(x, y, z)$$

Final output from this step is a *corticallis enhancing* function $c(x, y, z)$ which should be very high in places where we find the corticallis and zero or close to zero everywhere else.

A gradient at position (x, y, z) is:

$$\nabla f(x,y,z) = \begin{bmatrix} g_x \\ g_y \\ g_z \end{bmatrix} (x,y,z) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} (x,y,z)$$

Here we will give an example for computation in the X axis and how to compute an enhancing function $c_X(x, y, z)$ in this direction. We look in both directions in each axis and look at the two closest neighboring gradients, let's call them $g_{L1}, g_{L2}, g_{R1}, g_{R2}$.

$$\begin{aligned} g_{L1} &= \nabla f(x-1, y, z) \\ g_{L2} &= \nabla f(x-2, y, z) \\ g_{R1} &= \nabla f(x+1, y, z) \\ g_{R2} &= \nabla f(x+2, y, z) \end{aligned}$$

For each *left* and *right* version, we take the bigger one:

$$g_L = \begin{cases} g_{L1}, & |g_{L1}| > |g_{L2}| \\ g_{L2}, & |g_{L1}| \leq |g_{L2}| \end{cases}$$

$$g_R = \begin{cases} g_{R1}, & |g_{R1}| > |g_{R2}| \\ g_{R2}, & |g_{R1}| \leq |g_{R2}| \end{cases}$$

Next we compute an angle between each of those two gradient vectors and a vector pointing from the position where the gradient was taken to

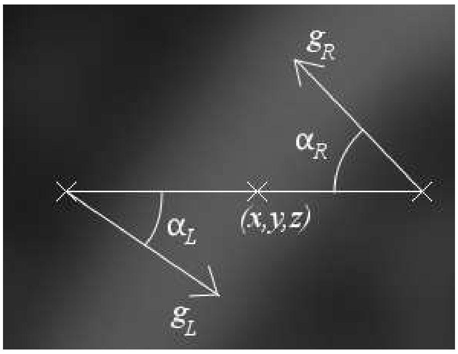


Figure 4.1: A graphical representation of angles α_L and α_R. The brighter path in the middle of this image represents magnified corticallis.

the point (x,y,z). We get angles α_L, α_R. A graphical representation of these angles can be found in fig.4.1.

$$cos(\alpha_L) = \left\langle g_L, \begin{bmatrix} x \\ y \\ z \end{bmatrix} - pos(g_L) \right\rangle$$
$$cos(\alpha_R) = \left\langle g_R, \begin{bmatrix} x \\ y \\ z \end{bmatrix} - pos(g_R) \right\rangle$$

If any of these two angles α_L, α_R is greater or equal than $90°$ ($cos(\alpha_i) \leq 0$) then it means that they point away from the point (x, y, z), the neighborhood of the point (x, y, z) is probably not the local maximum in the X direction and thus we do not compute further and finalize the *corticallis enhancing* function as $c_X(x, y, z) = 0$.

If the two angles are *less* than those $90°$, we end the first phase of choosing the gradients and proceed with computing the function using only those gradients, not their position or other information. The almost final function value is computed by multiplying the magnitudes of gradients(g_L and g_R) with a negative cosine of their angles (the more opposite they are, the bigger the function value is).

$$\begin{aligned} \beta_g &= acos\left(\frac{\langle g_L, g_R\rangle}{|g_L| \cdot |g_R|}\right) \\ c'_X(x, y, z) &= -cos(\beta_g) \cdot |g_L| \cdot |g_R| \end{aligned}$$

A small and final adjustment to the X directional corticallis enhancing function is limiting the cosine of the angle β_g between g_L and g_R to be more than a constant M_g. This limits the two gradients to be as opposing as possible (see also Section 4.3.2).

$$c_X(x,y,z) = \begin{cases} c'_X(x,y,z), & \beta_g <= -M_g \\ 0, & \beta_g > -M_g \end{cases}$$

Now we compute functions c_Y and c_Z. The computation is similar to computation of c_X, the only difference is that we take the neighbor gradients not along the X axis, but along Y, resp. Z axis. The final *corticallis enhancing* function is computed as a maximum of these three functions.

$$c(x,y,z) = max(c_X(x,y,z), c_Y(x,y,z), c_Z(x,y,z))$$

4.3.2 Notes on gradient component

By testing we found that looking in one direction by one and two voxels gives results good enough. Looking only at one neighbor voxel is not enough, because it may happen that the corticallis has a width of two voxels and this way we would not find for both of these voxels two sufficiently matching gradients. On the other hand, looking at three neighboring voxels is too much, because this may already introduce error by hitting the pelvic acetabulum in places, where the femoral head and acetabulum are in close proximity.

This discussion is of course largely dependent on the data we process. In our case we use standard examination CT datasets that have voxel dimensions around 0.5–0.8mm.

A very interesting property of the corticallis enhancement method is that we compute the method in all three dimensions. Computing it only in the axial slice would result in unclear data for upper and lower extremes. The 3D approach creates clear path for corticallis even for these extremes. On the other hand we are using only the X, Y and Z directions, not any other. According to our method, which gets the values in a rather discrete way, it may seem that this would result in lowered corticallis recognition ability in places, where the corticallis is not perpendicular to any of these three axes, but has an angle of roughly 45° with all of the X, Y and Z axes. In truth it showed up, that it is enough to do it only in these three directions, the ability to detect corticallis begins to diminish with angles greater than 60° and there the result from another direction is usually taken (remember: the final corticallis enhancing function is taken as the *maximum* from these partial directional functions).

The mentioned minimal angle constant M_g introduced in the final cost function computation is another parameter of the segmentation. Our testing suggested that a value of around 100° helps a little bit to the segmentation, but not very significantly. Larger values (for example around 130°) make more probable good segmentation of clearly visible segments where the corticallis is seen as a local maximum (and appears as a more dense line in the CT slice), but impedes the segmentation in parts where the corticallis is not seen as a line but rather as an edge of a dense region. So a value just above 90° seems to be a good choice. The reason to this is the noise, that makes more dense regions nonhomogeneous and in these regions somehow randomizes the gradient direction.

In the end our *corticallis enhancement* method proved to be quite good in localizing corticallis, while suppressing most of the other structures and trabecular bone. Of course, noise may bring local peaks of the cost function, but in the overall view these peaks are mostly solitary (and after the Gaussian filter also quite small) and insignificant in the segmentation step where some global localization of connected paths is applied.

4.4 Local gradient channels

When we have the cost function, we have to adjust it for the actual segmentation method we choose. For our method that is based on dynamic programming approach (*Dijkstra's shortest path search* algorithm), we have to process the function in a way that the corticallis should be around zero and all other parts should manifest themselves as non-zero.

Another information that we would like to add is the information obtained from the magnitude of gradient mentioned in Section 4.3. Simple gradient magnitude after some sort of noise-removal makes for us very nice *channels*, or paths with very small values on the ridge of corticallis, which are bordered with non-zero values in close proximity. Also very high peaks are found on the borders of very dense bones, such as the pelvis. In this sense gradient magnitude gives us similar results to our *corticallis enhancement* method, only with reversed values (small for corticallis, big for its neighborhood).

However, it also enhances *channels* in the trabecular bone and is often confused by noise. So we would like to utilize the information from both our corticallis enhancement method and the gradient magnitude to increase the robustness.

Corticallis enhancement method is able to pinpoint the corticallis, while gradient magnitude makes accidental confusion of pelvis for femoral head less probable. The gradient magnitude does not spoil the cost function on places where the actual femoral corticallis is, but adds other *penalties* around, so that we can stick to it during segmentation.

The actual cost function adjustment is as follows. First we clamp and stretch the function $c(x, y, z)$, so that the values from 100 to 1000 are stretched over the interval 0 to 256 (this interval has no hidden meaning, it is just for viewing purposes, other intervals like 0 to 1 can be used, only the final constants need adjusting). Values below 100 and over 1000 are clamped to 0, resp. 256. Next, the function is filtered with Gaussian filter, kernel size 5^3 (the reason for this kernel size is the size of the features in our CT data, more discussion in Section 4.1). The Gaussian filter makes the cost function smoother. That is better for optimal path search than the one we obtained from our method in Section 4.3.1, which exhibits strong edges and homogenous flat spaces. Let's call this result a modified cost function $d(x, y, z)$.

$$d(x,y,z) = \sum_{a,b,c=-2}^{2} \left(clamp \left[\frac{(c(x+a, y+b, z+c) - 100) \cdot 256}{900}, 0, 256 \right] \right) \cdot G(a,b,c)$$

where G(a,b,c) is the gaussian kernel.

The final combination of gradient magnitude and function $d(x, y, z)$ is done through linear combination and clamping. Let's call the final cost function $e(x, y, z)$.

$$e(x,y,z) = clamp[(A \cdot |\nabla f(x,y,z)| + B \cdot d(x,y,z) + C), 0, 256]$$

Where the constants A, B and C represent the final weights that combine our modified cost function with gradient magnitude. In our case these constants were chosen as follows:

$$\begin{aligned} A &= 1 \\ B &= -0.35 \\ C &= 80 \end{aligned}$$

These constant values were selected empirically, there is definitely a room for optimization. However, even these values were selected with some research and value estimation to make the segmentation possible.

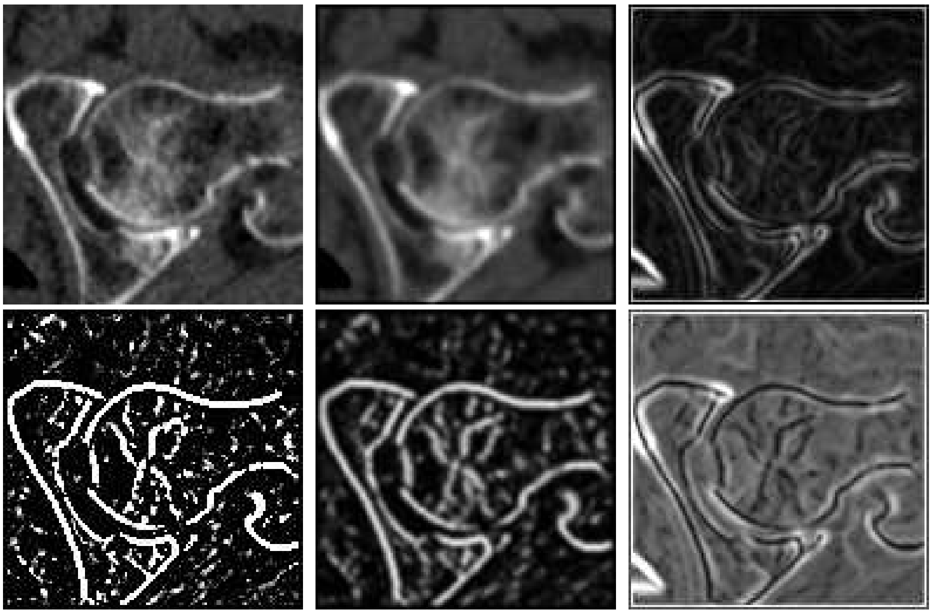


Figure 4.2: Cost function construction (from top left to bottom right): (a) Original data. (b) Gauss-filtered original data. (c) Gradient size approximation. (d) Corticallis enhancing function $c(x, y, z)$. (e) Function $d(x, y, z)$. (f) Final cost function $e(x, y, z)$.

The cost function $e(x, y, z)$ is clamped to interval 0 to 256. We have to make sure that it is non-negative everywhere (for the sake of correct graph-search algorithm, more in Section 5.2). The upper limit serves the purpose that we do not have one solitary peak that would attract the graph-search algorithm. A step-by-step example of cost function construction can be seen in fig.4.2.

Chapter 5

Segmentation

5.1 Segmentation algorithm

5.1.1 Slice-by-slice segmentation

A very nice method for segmenting internal organs that inspired our method is presented in [28]. This method is good for segmenting homogenous organs on a slice-by-slice basis. An interesting idea is to transform the slice into polar coordinates, where the segmentation is much easier for organs that resemble the shape of a deformed sphere or a deformed cylinder.

We need to find several input parameters: the upper and lower extremes where to begin and end segmentation and the center of polar transformation in each slice. However, this is enough only for homogenous organ segmentation. For our case we also need a starting point for segmentation and if the segmentation went wrong, then correction points that add additional shape information to the algorithm.

It is important to know for our computation, that the shape of the femoral head should resemble a deformed sphere. So path finding in polar coordinates is one of the best solutions, because even in places where there is no relevant information for segmentation, the algorithm would have tendency to go along the shortest path (in polar coordinates along a line, in spatial coordinates it would go along a circle) until it hits another defined path.

The 3D segmentation we chose is based on a slice-by-slice basis. In other words, the final 3D segmentation is composed of a series of 2D segmentations. Each of these slice segmentations creates a contour that locates the femoral corticallis. A summary of individual steps performed during the process is as follows:

1. Subdivide the volume into individual slices
2. Perform segmentation on several user-defined slices that provide seeds for the rest of the slices
3. Use seeds from step 2 and perform segmentation on the rest of the data
4. Floodfill each slice, so that the whole femoral head is filled into the segmentation mask

5.1.2 Slice selection

The input for each slice segmentation is a set of *waypoint* voxels (at least *one* voxel), which define the corticallis. If the bone is well defined, only one waypoint is necessary. However, if the femoral corticallis is unclear, more correcting *waypoints* may be added to guide the segmentation. See fig.5.1 for an example.

Output from each slice segmentation is a set of voxels, that form a curve around the femoral head. This implies, that if the data are clear enough, then only one control voxel would be necessary. This input voxel would provide us with enough information to segment one slice perpendicular to all other slices and provide one or two seeds for each slice to be segmented. This way we are able to segment the whole volume with seeds generated from the first slice. Of course we need to make sure that this one slice is segmented so that it goes through all the slices and really provides a seed even for the slices on upper and lover extremes.

However we have found out that only one *control slice* (the one that creates seeds) is not enough for real world CT scans. There is a part of the femoral head, which seems to be critical for our algorithm with most of the patients. In terms of human body directions that is the proximal posterior direction, or *upper rear* part of the femoral head. Here most of the friction happens, the articular cartilage is thinned and the density of corticallis is reduced, so that it is barely visible in the CT scan. The pelvis is usually not so damaged, so it stands out very much over the corticallis and confuses our algorithm. Here is the place we need to add most of the information.

If we look at all three slices through the body perpendicular to each of the base axes, we can see that the most unclear part is the sagittal cut.

Here we can add most of the required information in one or a few slices. So the first control cut is in the sagittal plane. Increased effort in segmenting this slice gives us an advantage in segmenting the other slices, because it generates seeds in the most difficult part of the femoral head.

But sometimes one slice is not enough, because the unclear area may be quite large. So it is a good idea to be able to add another control slice in the sagittal cut. In our case any number of sagittal cuts is possible, the only problem with each new slice is that it has to be segmented correctly, otherwise it introduces error into the rest of segmentation.

Sagittal control cuts are good for adding information into unclear areas. But anyway, sometimes the final segmentation in individual slices may exhibit strange behavior because of local parameters for this segmentation. For example it may happen, that a part of the trabecular bone stands out in a few slices and in these slices it is enough to confuse the segmentation while not changing the other slices. We need to bring in some coherency between the individual slices.

The simplest solution is to bring another control cut. Not in the sagittal plane anymore, but rather in the frontal plane. We have usually enough information from the sagittal cuts, so this segmentation goes almost automatically without any need for correcting. In case it goes wrong anyway (usually in places where the corticallis intersects the user-defined cutting plane – more in Section 5.3), then of course we can edit this segmentation as well. This frontal plane segmentation usually brings enough information to control the coherency of the final slices.

Now starts the final segmentation in axial planes. It finds the extremes from the sagittal control cuts and performs segmentation in each slice we have seeds for. This part goes usually completely automatically. In case any of the slices is too unclear, we need a facility to correct the segmentation on a slice-by-slice basis. But here we are usually without problems, only a very small number of slices goes wrong (they are usually near the extremal positions along the up/down direction).

So a short summary of slice selection is:

1. Perform *one or more* control cuts in the sagittal plane directly from the user-defined input voxels (for example see sagittal slices in fig.5.1)
2. Perform *one* control cut in the frontal plane, input voxels are seeds from step 1 and user defined correcting voxels.
3. Perform segmentation on a slice-by-slice basis for *all* slices between

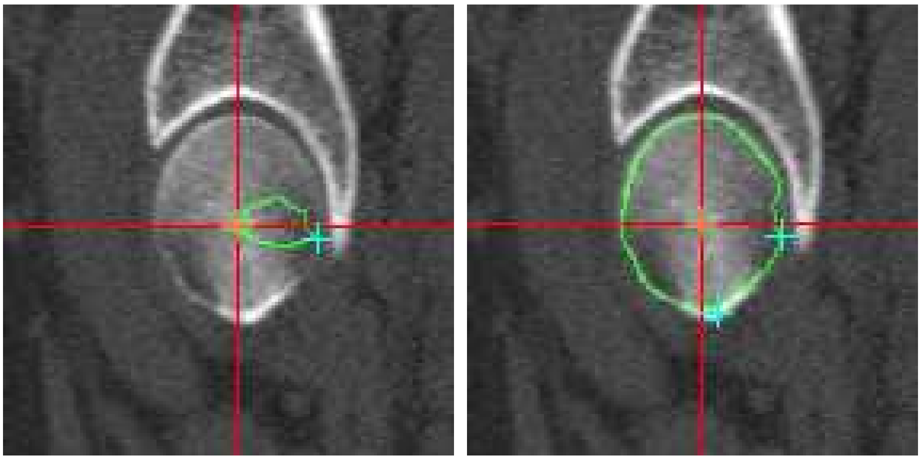


Figure 5.1: Example how one border voxel may be insufficient, but already two lead to correct segmentation.

the upper and lower extremes, input voxels are all voxels given by user together with all results from the steps 1 and 2

5.2 Slice processing

5.2.1 Graph search algorithm

The cost function $e(x, y, z)$ has some interesting properties, as mentioned in Section 4.4. The position of the femoral head corticallis is usually given by values around zero and they form a path that does not branch very much. There may be also other paths that represent other body parts that may look like corticallis in the CT scan, but these should not be connected to our femoral head corticallis with a clear path. These data properties imply that some sort of a shortest path graph search algorithm may be used.

Our method utilizes Dijkstra's shortest path graph search algorithm. The actual usage will be described later.

5.2.2 Problem definition and solution

We need to segment the corticallis in each slice. Our *input* is a set of one or more control points and the CT slice data. *Output* should be a set of voxels that define the border of the femoral head, or more precise, the *corticallis*.

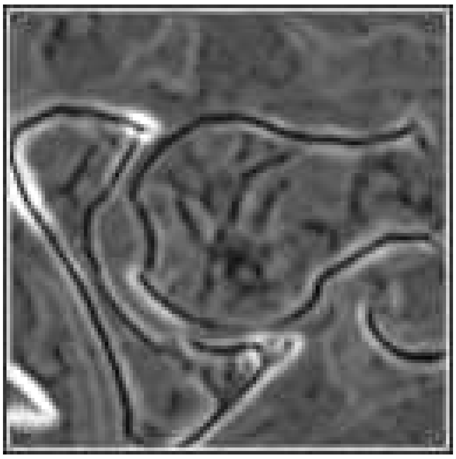


Figure 5.2: Example of a cost function $e(x, y, z)$.

The shape of the femoral head (as was already mentioned several times) reminds the shape of a deformed sphere. So we have chosen to do this 2D segmentation in polar coordinates. If the user chooses correctly the center of the femoral head, it is enough to use a vector perpendicular to our slice and going through this central point, that defines the central point of polar transformation at the point of intersection with this slice.

We then choose an initial angle step size and a radial step size and transform the preprocessed cost function $e(x, y, z)$ into a 2D cost function $p(u, v)$, where $u \in [0°, 360°)$ and $v \in [0, r)$, r is the maximal radius of segmentation (should be by our estimation at least one third bigger than the maximal distance of corticallis from center). When visualized as a greyscale image, this cost function $p(u, v)$ looks like a slice from $e(x, y, z)$ being *unwrapped* around our given central point (see fig.5.3). The corticallis is now projected into a roughly straight path from one side of the polar cut to the other, broken only in the place where there is femoral neck (in case we are looking at frontal or axial slice, in sagittal slice it should be an almost straight line). We also transform the initial control voxel positions into these coordinates. They should be projected into points lying hopefully on the cortical path.

Now our problem is simplified into the following:

Problem 2 *Find a path from one starting control point around the whole polar cut. It must cross the* angular *border, but not the* radial *one and on its way it must hit all the other control points. The path it takes should be through points with small values (as to why, look into Section 4.4).*

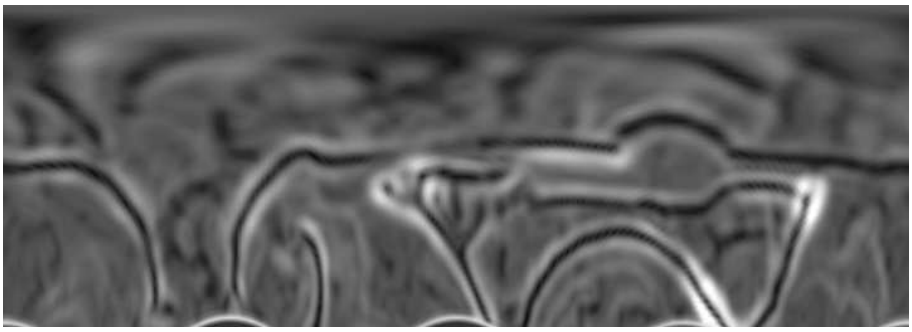


Figure 5.3: Polar transformation of the cost function $e(x, y, z)$ from fig.5.2.

As you can see, the definition of this problem is not very clear, at least not in a mathematical sense. Especially the phrase "through points with small values" is very unclear. This is because we do not know what exactly we want. We only know that our cost function $e(x, y, z)$ transformed into polar coordinates has lower values (theoretically should be around zero) in places where we find corticallis or similar structures and higher values elsewhere.

The path that consists of low values is almost continuous, but may be broken in some places and some segments may be missing completely. Additionally, we have a set of control points that we want to *hit* in a specific order (this order is given by their respective positions along the angular – or u – coordinate). This further reduces the problem:

Problem 3 *Find the cheapest way in an image from starting point to end point. The way may wrap around the horizontal coordinate, but each successive path pixel cannot have its horizontal coordinate lower than its predecessor in case it does not wrap around the whole image. For each horizontal coordinate there may be only one path pixel.*

For this, it would be advisable to use some sort of graph search algorithm. We have used *Dijkstra's shortest path graph search* algorithm. We define the weighted directed graph for this algorithm as follows: vertices V are the pixels in the polar cut image. Edges E lead from each pixel to its right neighbor, upper right neighbor and lower right neighbor if they exist. For vertices at the horizontal end, the right neighbors are defined as vertices created from pixels at the beginning of horizontal coordinates (with coordinate $u = 0$). The *weight* of each edge is defined as the original pixel value of the vertex the edge is pointing to.

Let's consider we have k control points $C_0, \ldots, C_{k-1}$ sorted by their u coordinate. Now we want to find a path between each two consecutive control points in this image (the *last* control point horizontally has the *first* control point as its consecutive neighbor). So we define the graph as mentioned above and run the shortest path search algorithm for each pair $C_i, C_{(i+1) mod\ k}$ where $i \in [0, k-1]$.

This way we will obtain the shortest path around such defined polar cut. Now we transform it back from polar to spatial coordinates. We have now a set of border voxels that define the femoral head corticallis.

5.3 User-defined cutting plane

The bone is not of a healthy person, but is broken somewhere in the femoral neck. We need to add some additional information about this fracture, to limit the segmentation only to the femoral head. Actually we want only the femoral head, even if most of the femoral neck is intact and connected to the head, because this segmentation is meant to provide some information about the quality of the bone where we will place the screw – and that is in the middle of the femoral head. The femoral neck is insignificant for us, or even unwanted, because it may bring an error to our femoral head trabecular bone quality estimation.

We had to design a facility for the doctor to provide us with this information. The simplest way for both the program and for the doctor is to define a *plane*. More elaborate methods are sometimes confusing and hard to visualize or imagine. A plane is easily controllable by three points in space and a precise placement can be displayed even without volumetric data visualization just by intersection with each of three basic slices through the CT data (frontal, sagittal and axial).

However, implementation of this general cutting plane into the method was somehow tricky. The first idea to stop the shortest path search algorithm at the plane cannot be used, because that would ruin our expectation about the shortest path and we will find the shortest path from one of the control points to this plane but not to the intersection of this plane with femoral corticallis.

This implies the second idea to stop the algorithm at the intersection of the plane with femoral neck corticallis. That would be equivalent to adding another control points on the intersections of the plane and corticallis and then artificially connect these two new control points through the neck, which is not possible, because we do not know, where this intersection is, we are actually trying to find it.

So the third idea was to change the data for our graph search algorithm in a way that would clearly define, where this plane is. On the other hand it should not spoil the result, if the real corticallis is very close to this plane but does not intersect it. So what we did was to artificially create another *channel* through the preprocessed data, that would look like the corticallis for the algorithm and would ease the path search through it but would not attract the algorithm in case the corticallis is parallel to it and very close.

The implementation of this ides is to create a matrix of values with the same dimensions as the original data. In this matrix we set all voxels equal to 1, only those whose center is closer to the plane than n_V voxels are set to 0. Next follows Gaussian filter with mask size m_G.

These constants are next parameters of our segmentation, but in this case they do not alter the actual corticallis segmentation, but only help to guide it through places where there is no corticallis present. In our case these constants were:

$$\begin{aligned} n_V &= 1 \\ m_G &= 5 \end{aligned}$$

When we have this mask, we simply multiply voxel-vise the original preprocessed data with this mask. That would create the wanted *channel* inside the original data. From the worst-case point of view this is not correct, because the segmentation may overcome this boundary and also add some parts of the bone behind this plane, but by testing on many CT datasets it is clear that it is not the case of real world data, where this safely guides the segmentation over parts of the bone where no corticallis is present.

5.4 Final volume

After the *segmentation* step we have located the corticallis and it is marked with a mask. To get the whole volume we assume that all slices are cutting through a deformed sphere, as was the first assumption about the shape of the femoral head. So they do not form difficult shapes and we can use standard 2D *floodfill* algorithm on each slice to get the whole volume. There will be errors along the *poles* of the femoral head (the top and especially the bottom of the segmentation), but these represent only a very small number of wrong voxels and are usually without problems discarded in the last morphological step that follows.

So now we have the whole femoral head but we want only the trabecular bone inside. We can notice that the corticallis has almost the same thickness along its whole surface. The absolute value of this thickness can wary from patient to patient, but one patient usually has the corticallis thickness uniform. And this corticallis happens to be at the border of our segmentation. So the first and simplest method to deal with this is some sort of morphological filter, like *erosion*. Erosion on a 6-neighborhood or a 26-neighborhood (3D equivalents of a 2D 4-neighborhood and a 8-neighborhood) removes one layer of voxels from the border of our set. We used erosion on a 6-neighborhood and all our results are computed with it.

Each patient may have different corticallis thickness (usually 2-3 voxels), so it is up to the examining specialist to decide how much should be removed. In our testing datasets it turned out that it is good to remove 3 layers each time, because even for patients that could require only 2 layers it introduces only a minor error in the final statistical analysis (only a few percent of the resultant HU values). For example we are looking whether the mean value of the patient is 100HU, 150HU or 250HU (the real values are in this area) and the difference of this mean value may be around 1 to 10HU, so it is clear that removing the third layer even when it is not necessary introduces insignificant error to the whole computation.

5.5 Parallelization

The algorithm can be parallelized without problems. The only part that must be serialized is the primary and secondary slice segmentation. The final axial slice segmentation can be run in parallel. In case the parent thread that is supposed to clean up the results is well written, the boost in speed with each additional processor is almost linear. See Section 7.3 for actual numbers on a quad core processor.

5.6 Notes

It turned out that the algorithm is not so sensitive to the central point selection. It is, of course, important to have the central point *inside* the femoral head and as close to the real center as possible (we can define the *real center* as a geometrical mean value of the femoral head, that is the best point for our segmentation), but there is a tolerance of about 1cm in each dimension, which does not make much difference in terms of segmentation output quality and it is not difficult to hit this tolerance by just looking at the data.

The *result* is, as mentioned, a *set* of vertices in polar coordinates that are transformed to a *set* of voxels in original data. At first sight you may think that it can happen that the result will not be connected, which would prevent us from using the simple floodfill algorithm in the end to complete the result mask. In general it would be good to use some more robust representation, such as connecting the result voxels with lines. But in our case it turned out that we need good enough precision for the polar cut construction, so that all relevant voxels are represented. This resolution must be so good, that if we have a connected path in the polar cut, its transformation into the spatial domain would not bring us non-continuous path, but even many voxels are duplicated. So for our case it was not necessary to utilize some more robust path representation.

Chapter 6

Implementation

The application implements the features mentioned in previous chapters and several other features, that were there added for various reasons during the development of our method. Some of them were added on request from MUDr. Horák and MUDr. Vaculík, who were constantly informed about the progress and helped to shape the functionality of our application. Other features are for debugging purposes or to show some properties of the implemented methods. A detailed description of the actual application implementation follows in this chapter.

Implementation is done in C++, compiled with compiler *Msvc* version 8 from *Visual C++ 2005*. Final application is designed to run under Windows system. We have found no functionality problems on various 32-bit MS Windows versions. Thorough testing has been done on Windows XP SP2 32-bit and Windows Vista Business Edition 32-bit. Porting for Linux system is not planned, although the program runs fine under the *Wine* emulator.

We used *Windows API* for user interface and some basic disk operations and *DICOM Toolkit* ver. 3.5.4 by *OFFIS* for DICOM data loading and processing.

6.1 Application structure

The application can be subdivided into four main parts (see fig.6.1):

DICOM interface – Utilizes the library OFFIS DCMTK (DICOM Toolkit, [2]) for loading CT datasets stored as a set of DICOM files and provides a way to store them in the program.

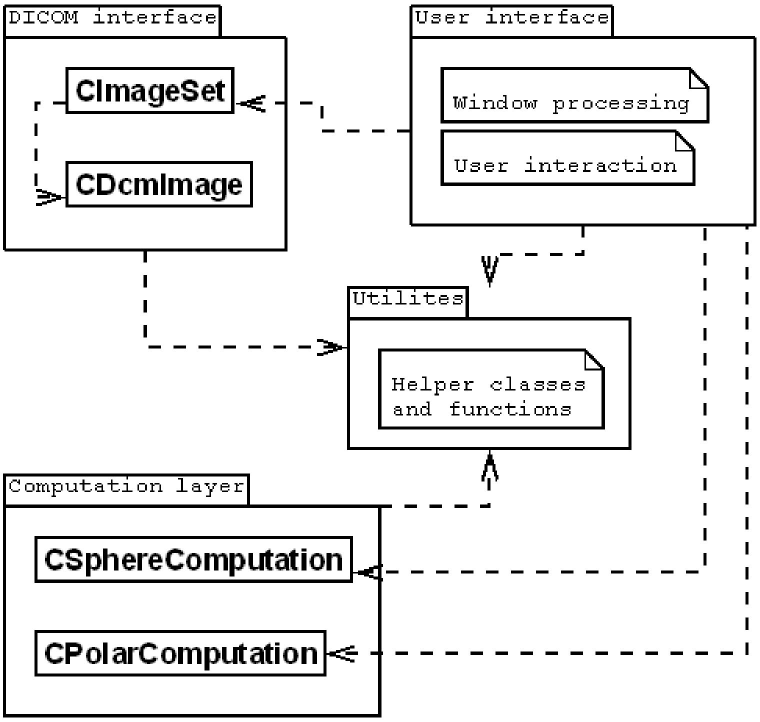


Figure 6.1: Program structure. A brief overview.

User interface – Provides a way for the user to interact with the program and displays results.

Data processing – Implements segmentation algorithms.

Utility classes – Provides transport facilities between other parts of program and some additional helper classes.

We are working with quite a lot of data (for today's computers), because one standard CT study has hundreds of megabytes. It is not such a big amount that would need some special caching, it can all fit into the main memory. But for some quick processing, there has to be some order and optimizations of the data processing and transfer.

For the sake of clarity of how the data moves in the program, we will explain first the utility classes, then continue with DICOM layer, that loads for us the data from disk and stores them. Next is the user interface, that centralizes the whole program and provides input and feedback between the user and the program. Computation layer will be explained last with details and issues of our method of implementation.

6.2 Utility classes

6.2.1 Volumetric data transport

We know that the size of the femoral head is not bigger than about 6cm (usually it is around 4cm), we do not want to process the whole volume, because we have some initial information from the user about where approximately the femoral head is. Actually one of the input parameters is the approximate center of the femoral head. So we can estimate that all of the segmentation would be done inside a limited area around this point. This leads to an idea of making an interface for accessing and transporting a subvolume of the whole CT scan. The data has to be volumetric, so a box with a parametrically adjustable size comes to mind as the simplest representation.

Implementation of this interface is in the class `CVolumeSet<T>`. This template class has an ability to store many different types of data. This class has a set of basic methods for setting and getting its size, ability to remember and provide the original position from which it was taken in the original DICOM dataset, method for finding out whether a given point lies inside this volume and many others (discussed later). The data is stored as a `std::vector<T*>`, which is a vector of 2D slices, that are internally saved in an allocated 1D field.

The class provides methods for getting a constant pointer to all three basic slices. Slice in the XY plane is just a pointer to an adequate plane in the mentioned vector. Slices in the XZ and YZ planes are first loaded from the original data stored inside an internal structure and then a constant pointer on this structure is returned.

If we want to access individual voxels, it is possible both for reading and writing. If we want some global access, we can set the whole volume to zero or some given value. For some global access to change individual voxels, but assign each voxel a different value, we can get a non-constant pointer to the slice in XY, but not the other two (this is clear from the point of view how the slices in XZ and YZ are constructed).

The class is *thread safe* for accessing different individual voxels, or for reading/writing into the XY plane. Most other operations, such as resizing, accessing slices other than XY, accessing the same voxel, etc. are *not* thread safe.

This class is as a template quite incomplete concerning its abilities and algorithms. All other processing of this class is available through specialized functions, that are not members of this class. One such function that falls into the data transport topic is the `copyVolume` function. This function has several overloaded variations for all needed conversions between different types. Other functions have more processing power in them and so will be discussed in Section 6.5.

The primary use for `CVolumeSet<T>` is for transporting data between the *DICOM interface* layer and the *data processing* layer. But it is also used for storing the data inside the data processing layer and performing the *preprocessing* step.

6.2.2 Planar data transport

Basic planar data transport is done through the class `CGeneric2DData<T>`. This acts as a standard 2D rectangular image with the ability to store arbitrary data. Only common methods for resizing and data accessing are provided.

Another class that stores 2D data is the class `CPolarCut`, but this class implements also quite sophisticated transformations to and from polar coordinates, so it will be further discussed in Section 6.5.

6.2.3 Helper classes

Representations of points in 2D and 3D are structures `SPoint2D<T>` and `SPoint3D<T>`. These structures have some basic operators defined to simplify operations with them. These two are defined as general as possible and are used for many operations, both during the *data processing* and in the *user interface.*

`CTimer` is a very simple class for measuring time. This is for the purposes of this thesis to present the results of our algorithm. The implementation is done with standard Windows API function `GetTickCount`, that implies that the actual precision is limited to around 15ms, but we are measuring seconds or hundreds of milliseconds, so that precision is good enough for us.

Two functions are for `std::vector` processing, `getVectorSum` is used for returning the sum of all members from the whole `std::vector`. This is used for Gaussian filter construction and for some bone quality estimations. The function `buildGaussFilter` creates an approximation of a Gaussian curve by creating a row from Pythagoras triangle. This is used for the separable Gaussian filter during preprocessing.

The function `createTestData` has been used during development for creating artificial test data which resemble the input but are still somewhat controllable. The data can be deliberately damaged by a variable amount of white noise to make them more representable. This function is not currently used in the code, but could be enabled by a simple change in the user interface. It has been used during our method planning.

6.3 DICOM layer

The DICOM layer is used primarily for loading and providing the 12-bit data from DICOM files. It is not its only purpose, it also stores all 12-bit data, utilizes masks obtained from various computation algorithms and performs also a basic statistical analysis of the masked data.

The DICOM layer basically consists of only two classes. `CImageSet` and `CDcmImage`. Both are highly specialized to work on the task given, no generalization except for the most needed one was done. Class `CDcmImage` is the class that works directly with DICOM data. It is our interface for one slice of CT scan. On the other hand, class `CImageSet` is a sort of container that represents the whole CT scan and contains an array of `CDcmImage` to store the data.

6.3.1 DICOM slice

First we will discuss the class `CDcmImage`. As already mentioned, this is our interface to one DICOM slice. It is designed to provide access to 12-bit 2D data, so we can represent it as an image. Unlike normal dynamic 2D images, such as `CGeneric2DImage<T>`, this class does not support explicit resizing. This is because it is not prepared (nor designed) to hold arbitrary data, but only data loaded from disk. The resizing operation is done only during loading.

Other methods of a 2D image are present, such as accessing individual pixels, accessing the whole data, getting size in pixels, etc. There are some additional methods for getting information about the DICOM data, such as voxel size, getting bits per pixel and getting information about a patient (name and id).

There are some methods for measuring (usually their name contains the word for measuring). These methods are for debugging purposes, they are the leftovers from the time of method design. Currently they are not used, but are left in the code for the purposes of debugging.

An important feature of the `CDcmImage` is the loading process. The loading is done by calling a method `loadImage`, but this method has an ability (in the form of a parameter) to postpone the actual data transfer from disk to memory. This way when we process the whole directory of DICOM slices, we only assign filenames to the `CDcmImage` objects. The actual data loading can be forced by calling `forceLoad`. Subsequent forcing is ignored once the data is correctly loaded. This way we can very quickly initialize the application after the user selects a directory as a source of DICOM data for us. All data are loaded at the moment they are needed, so this way we can skip the whole parts of volume the doctor does not need to examine.

Another feature that is implemented along this postponed loading is some memory management in case of very large datasets. We can specify a *radius* of slices that must be kept in the main memory around the actual slice, but all slices that fall outside this radius are discarded by calling `releaseLoad` and their memory is freed. They can be loaded again if they are needed.

6.3.2 DICOM datasets

The class `CImageSet` is a kind of container for our slices. It is one of the most complex classes in the whole project. It is also one of the biggest. The primary purpose is to hold a `std::vector` of pointers to `CDcmImage` objects. Because the user interface layer has rarely a chance to access the images themselves, a lot of methods are provided to just hand over the information from `CDcmImage`.

Loading is done by calling `loadDirectory`. At this point we do not support the *DICOMDIR* standard, that is planned for future improvements. So at this moment we need the DICOM data from one dataset to be stored in one directory and sorted alphabetically (the DICOM file header provides information about individual slices position, but we have not yet incorporated this information to the program, it is also planned for future). The `loadDirectory` goes through the given directory, finds all DICOM images and for each of them assigns an internal object `CDcmImage`. The data are not forced to load immediately, the actual data loading (see postponed loading in Section 6.3.1) is done when the image is needed. It is also planned to do this loading transparently in a separate thread, the program design is prepared for this but it has not been done yet, it is also a planned feature.

As has been already mentioned in Section 6.3.1, we have provided a facility to limit the memory usage. Class `CImageSet` provides a possibility to set a radius around the actually displayed image that should be kept in the main memory and everything else should be released until it is needed.

Next feature is providing the data, both for computational and display purposes. Getting data for computation in the form of a `CVolumeSet<T>` is very simple, the caller provides an existing object of the type `CVolumeSet<T>` with defined dimensions and the position in the complete volume data (in terms of voxel indices) and our `CImageSet` fills the provided object with defined data and if the requested volume extends beyond our data size, all such voxels are given the value representing clean air, that is -1023.

Providing data for display purposes is a bit more tricky and creates the most of the complexity of this class. The class `CImageSet` does *not* directly display the data. It only processes the data into three orthogonal slices through the dataset (axial, sagittal and frontal) and provides transformed 8-bit data obtained by applying a so called *window* function on the 12-bit data.

In short the `CImageSet` gets a center of projection (a point in the whole volume that represents the position where the user is looking), finds the appropriate axial slice and computes two orthogonal (sagittal and frontal) cuts through the volume, all this in 12-bit data. When this is done, a conversion with user-defined *window* function is performed to get 8-bit data, which are returned to the application as a 24-bit color DIB (Device Independent Bitmap). Both types of slices (8-bit and 12-bit) are stored locally, only a pointer to them is given when the user interface needs it to display the images.

There are some options on how to exactly compute the two cuts, for example whether the sagittal and frontal slices should always follow the image chosen in the axial direction, or whether they should be centered somewhere else. Constructing the two slices proved to take some time (about hundreds of milliseconds), so it is better to leave the following turned off and center the slices only on explicit user request. This way the user can sweep through the data more interactively and does not have to wait till each image is computed and displayed. Even several hundreds of milliseconds are too much if you want to display about ten images per second.

The *window* function (fig. 6.2) has a form of linear transformation from 12-bit data to 8-bit data. Many radiologists incorrectly call this function *contrast*, but it encompasses both contrast and brightness transformation and reduction from 12-bit to 8-bit. The function has two parameters: win-

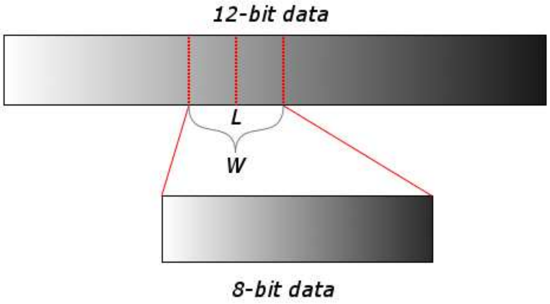


Figure 6.2: Example of a window function transformation

dow *level* and window *width*. *Level* is the central point that should be transformed to the half of the target interval (to value 127 in case of 8-bit target data) and *width* is the diameter of the source interval. So for example for parameters $L = 50 \quad W = 100$ we will transform the interval 0 to 100 to values from 0 to 255 (the target interval is almost always 0 to 255, because modern graphics cards and monitors have 8-bit color processing or at least 8-bit input and output – except for some specialized and very expensive equipment that work with 10-bit colors). See fig.6.2 for a hint how it works.

The window function is not the only transformation that is done on the data. `CImageSet` also displays the mask as voxels with different colors (in our application it is the green color). Mask is there for two purposes:

1. Visually present the result of the segmentation
2. Perform some basic statistical analysis on the masked data

Displaying is very simple. During the last step of the window function transformation there is an additional condition that checks whether the processed voxel is marked in the provided mask. If it is marked, then output its color partially changed to visually distinguish it from the other voxels.

Statistical analysis (in `getMeasurementResult`) finds all voxels marked in the mask and returns their mean value and standard deviation. The *histogram* can be obtained from `getHistogramFromMask`. It works almost the same way, only the result is a `std::vector` of quantity values.

6.4 User interface layer

6.4.1 Window

User interface is based on 32-bit Windows API. So the basis for the whole program is a `WinMain` function that creates the main window and `WndProc` function that processes the window messages sent to the main window.

The `WndProc` does not contain all the code for processing window messages, for most of the messages there are special functions `On`... (for example `OnCreate`, `OnPaint`, etc.). Class `CGlobals` holds various state information about the user interaction (for example which tool is currently used, etc.). If no files (DICOM data) are opened, then most of the functionality is either disabled or ignored. All of the functionality can be accessed in the program menu, the most important features are accessible also through the toolbar. Usually the user selects a tool in the toolbar, that sends a menu ID message to `OnCommand` and if it is a command to change a tool, then the actual tool is selected in the `CGlobals` object. `OnCommand` also processes activation of various dialog boxes, showing or hiding information displayed in the main window client area, opening files/directories, starting computations and doing some simple tasks on the result data, such as eroding or dilating.

The rest of the main functionality that is not handled through the `WM_COMMAND` message goes through the `WM_KEYDOWN`, `WM_LMOUSEDOWN` and `WM_LMOUSEUP` and their respective copies for the right mouse button. The function `OnKeyDown` processes mostly movement through the dataset with cursor and PgUp, PgDown keys and changes some predefined parameters for the *window transformation function* (see Section 6.3.2 for explanation). All four `OnLMouseDown`, `OnLMouseUp`, `OnRMouseDown` and `OnRMouseUp` process the input from mouse events. These are usually dependent on which tool is selected from the menu/toolbar and then according to the tool, they perform operations such as getting original value at a specified voxel, running spherical mask computation (explained in Section 6.5), adding or deleting various control points for computations, etc.

Function `OnPaint` paints the client area of the main window. It finds out what should be drawn and what not. The actual drawing begins with displaying transformed data from `CImageSet` and then on top of it draws various other information, such as the position of the other two orthogonal slices, location of border points for computation, location of user-defined cutting plane, etc. The *mask* is not displayed explicitly here, it is already imprinted in the data obtained from `CImageSet`.

Other functions for processing the window messages are not so important and are usually self-explanatory, so a look into the code should be enough for finding out their functionality.

6.4.2 Coordinates transformation

For the sake of user interface we need to be able to transform from screen coordinates to volume coordinates and back. This is not a difficult transformation and is used very frequently. So for this we have several functions.

Both overloaded versions of `Get3DPositionFromScreen` do the same – compute which point on the screen corresponds to which position in the volume data. This is used for example when the user clicks somewhere and we do not care about which slice was hit, but just the final position in the 3D data.

Functions `Get2DScreenPosFrom3DMain`, `Get2DScreenPosFrom3DCutXZ` and `Get2DScreenPosFrom3DCutYZ` compute the backward transformation. We have a point in the volume and we want to find out where on the screen it will project. Each version projects onto a different slice and each has also an optional output variable that tells us whether the point would project on the given slice at all.

6.4.3 Dialogs

There is a number of dialog boxes, mostly for debugging purposes. All dialog boxes have methods for hiding, showing, destroying and some parameters depending on the type of the dialog.

The most apparent dialog box that is visible almost all the time is `CInfoDialog`. This is a simple window with read-only editbox, that displays various formatted information, such as the size of voxels, results from computations or some information related to current processing (such as how many images remain to be loaded or which part of the segmentation is the application doing right now).

`CParamsDialog` serves for setting various parameters of the application or computation methods. Again, it is a small window that contains a number of edit boxes that show and alter the values of some internal variables.

The remaining dialogs contain basically only canvas, where they output their information, no controls are present. `CGraphDialog` displays a graph of some function $f(x)$, that is quantized in discrete locations with uniform distribution. It is currently used to display the behavior of data when the user draws a line with the *Get Value* tool. An example of such graph is in fig.2.2.

`CHistogramDialog` is very similar in the sense that it displays some property of the data. This time it displays a histogram of values marked by a mask from segmentation. See 3.2 for such a histogram of femoral head.

`CVolumeDialog` and `CPolarDialog` both show some data from the segmentation algorithm. This is used for debuging purposes and are usually used for displaying the cost function in some degree of its creation. `CVolumeDialog` displays a small `CVolumeSet<T>` that contains processed data during computation, `CPolarDialog` is capable of displaying a polar cut (used several times throughout this thesis to present the polar cuts). They are both unused in the final application, but could be very easily incorporated by providing them with relevant data.

6.5 Computation layer

This is the layer, which gets some parameters (depending on which computation are we performing) and returns a `CVolumeSet<unsigned char>`, which is a mask to be used for further display/computation inside the `CImageSet` (see Section 6.3.2 for detailed explanation of how the masks are used). We have two types of computations:

- Spherical computation
- Channeling segmentation

There is also a number of various helper functions, usually specialized functions for `CVolumeSet<T>` processing for some known `T`.

6.5.1 Helper functions

There is a lot of functions beginning with `vol...`, for example `volMultiply`. All these functions work with volumetric data, usually saved as floats (that means `CVolumeSet<float>`), but some of them are defined for arbitrary data (for `CVolumeSet<T>`). They have usually one reference to a target volumeset, one or more constant references to source volumesets and optionally some other parameters.

`volTreshold2DFloodfill` is a function with 2 overloads. One is for filling `<unsigned char>` masks (this is actually used after segmentation, when we have the borders of the femoral head and we need to fill the borders on each slice). The second function with the same name is for creating a mask based on `<float>` data. The input is a float volumeset, a seed and two float limits. All values between these limits are filled with a floodfill algorithm, so any value below or above these limits is considered as border voxel. A similar function is `volTreshold3DFloodfill`, which acts the same way as the 2D version that works on float data, but this time the floodfill algorithm is flooding in all three dimensions. This has been used during our method development to check, whether this kind of filling would be enough after our preprocessing step. It turned out that it is not enough, so we had to use our current segmentation.

`volGaussfilter`... are functions for performing the separable Gaussian filter. Versions with `X, Y, Z` at the end are 1D filters performed on the whole data with given mask size. The final `volGaussFilter3D` calls subsequently all these functions to obtain a final gaussian filtered volume.

`volClampAndStretch`, `volMultiply` and `volLinearCombination` perform simple operations on one or two volumesets. `volClampAndStretch` takes only one volume and transforms linearly all voxels from source interval to destination interval. All values outside the source interval are transformed to minimal, resp. maximal value of the destination interval. `volMultiply` takes two volumes with the same sizes and multiplies each voxel from one volume with its counterpart with the same coordinates from the other volume. And finally `volLinearCombination` takes two volumes with the same sizes and three parameters A, B and C. Each voxel from the destination is computed as follows:

$$dest(x, y, z) = A \cdot src1(x, y, z) + B \cdot src2(x, y, z) + C$$

Next two functions are `erodeMask4` and `dilateMask4`. They work on mask data (i.e. `<unsigned char>` volumesets) and perform an *erosion* or a *dilation* on a 3D equivalent of a 2D 4-neighborhood (in 3D that may be called a 6-neighborhood). Erosion just makes the mask smaller, volume outside its boundaries is defined as 0's, so voxels right at the border are cleared and the whole mask is shrunk by one voxel on a 6-neighborhood. Dilation performs the opposite, the mask grows by one voxel around any nonzero voxel. However, the physical dimension of the mask volumeset stays the same, so in the end it fills the whole volume of the volumeset, but does not continue further nor enlarge the volumeset.

6.5.2 Spherical computation

Our whole work was based on the need of MUDr.Horák and MUDr.Vaculík, who were working on a research project concerning the *femoral neck fracture* treatment (more in Chapter 1). They provided us with the needed data and information and we tried to come up with an application that would help them in their bone density estimation. The ultimate goal was, of course, complete and precise segmentation of the femoral head. But the medical team was under time pressure and needed some approximate results as soon as possible.

So the first idea was to come up with an application, that would have already some basic user interface and an ability to measure the average density in a sphere with a user-defined radius. This is simple to code and also simple to use. The only user input needed is one click representing the center of the sphere and optionally a change in the sphere diameter.

To unify the computation with our *channeling* method, the computation is divided into two parts:

1. Spherical mask preparation
2. Statistical analysis

The second part is the same as with our *channeling* method, it is done inside the `CImageSet` class. The first part is implemented in the class `CSphericalComputation`. So the class `CSphericalComputation` gets parameters such as voxel dimensions and sphere radius and returns a mask in appropriately sized `CVolumeSet<unsigned char>`. It is then transported through the user interface to `CImageSet`, its *original position* is adjusted according to where the user clicked (for explanation see Section 6.2.1) and the results are computed.

Advancements have been done to allow some arbitrary and continuous changing of the diameter. The internal structure for the mask is allocated a little big larger than needed and then if the diameter is changed, the mask size remains the same unless the sphere is bigger or by a defined tolerance smaller than the mask size. This is to prevent many allocations/dealocations if the sphere has to be changed very frequently. In the end it showed up that this measurement is usually done on predefined sphere diameters, such as 20mm and 30mm, so a continuous change was not needed. But it was pointless to remove this functionality so if there is need, then it can be used without any alterations.

6.5.3 Channeling segmentation

This is the final implementation of our method described throughout this thesis. Again, it consists of two parts:

1. Segmentation and mask preparation
2. Statistical analysis

The *statistical analysis* is done exactly the same way as in spherical computation, only the *original position* of the processed volumeset does not have to be manually adjusted, it is obtained during the data reading from `CImageSet` and travels through the whole computation along the various processed volumesets.

The segmentation itself can be further roughly subdivided into three parts:

1. Parameter setting
2. Data preprocessing
3. Slice-by-slice segmentation

All this is implemented in the class `CPolarComputation`. This is probably the most complex class in the project, even more than for example `CImageSet`.

The user first sets parameters, such as border voxels, user cutting plane and computation center. Each change of critical parameter that may change the preprocessed volumeset data (cutting plane and computation center) invokes data preprocessing before the first attempt to do segmentation. Each subsequent attempt to do segmentation without changing mentioned critical parameters ignores preprocessing and uses old preprocessed data.

Preprocessing (if done) consists of several operations on source `CVolumeSet` data:

1. Gaussian filter with a Gauss curve approximation kernel of size 5
2. Computing of a gradient
3. Cost function computation
4. Adjusting the cost function to fall into the interval 0-255

5. Gauss filtering of the cost function
6. Combining filtered cost function with a gradient size approximation
7. Another adjusting to interval 0-255
8. Multiply of the result with an artificial channel created by a user-defined cutting plane

The functions used for this are described in Section 6.5.1. A flag is set that the data have been correctly prepared and unless the input parameters change, no other preprocessing will be necessary during subsequent segmentations.

The segmentation itself is started by calling a `run` method. There are two optional parameters, one defines whether to do all three phases of segmentation (primary control slices, secondary control slices and complete segmentation) or whether to do only the 1^{st} and 2^{nd} phase (only the control slices) and if all phases should be done, then if the result should be floodfilled or should stay only as a border (which is used for debug purposes to see the actual segmentation result). `run` then checks whether all needed parameters of the computation are set (that means at least one primary border point and the center of computation). Then it clears the result mask and checks whether the preprocessing step has been done and is up to date (and runs it if necessary).

When the preprocessing is done, it transforms all three types of border voxels (primary, secondary and correcting) from global coordinates of the whole volume to local coordinates inside the source `CVolumeSet<float>`. If we have less than one primary border voxel, then no computation is done and `run` returns with an error, because we have no starting point.

The segmentation itself is divided into three parts, each part corresponds to one phase:

1. Segment one or more primary sagittal slices
2. Segment one frontal slice in the middle of the volumeset (the slice that contains the user selected central point)
3. Segment all axial slices that contain at least one border voxel

The first phase is segmenting the primary control cuts in sagittal slices. We count all primary border voxels and find all slices that contain at least one border voxel. For each of those slices we run `computeDijkstraInCut`, that segments the slice and returns all voxels that represent the path along the femoral head corticallis (see Section 6.5.4). To the voxels we have obtained from the segmentation, we add the original voxels and then filter them, so that each voxel is unique (it may happen that two or more pixels from the polar transformation may project back to one voxel, so for each such voxel we want only one representant). During the filtering all voxels are written into the result mask.

With the newly created set of border voxels we run the second phase, which is segmentation of one central slice in the frontal plane (frontal slice that contains the central point selected by user at the beginning of parameter setting). Again, we run the method `computeDijkstraInCut` on the slice with all known border voxels excluding correcting voxels for third phase. Those are primary and secondary user selected border voxels and all result voxels from the first phase. When this segmentation is done, all voxels are again merged together and filtered to keep only each unique representant (and they are also written into the result mask).

After the second phase the segmentation can be stopped (by a parameter to `run`), so that the user can check whether these segmentations were done correctly and correct them if needed. If the user is sure that the computation of these control slices is alright, the computation is not interrupted and we continue with the third phase.

The input is all merged and uniquely filtered voxels, both user selected (all three – primary, secondary and correcting) and those obtained from the first two phases. The third phase is a little bit more complicated, because it utilizes parallel processing. The user selects (in Parameters Dialog) how many threads he wants to use. Array of `CPolarComputation::SThreadData` structures is created, which is a structure for communication between the main thread and the segmentation threads. On each field of this array, we run a separate thread.

One `while` cycle runs until all slices are processed and results returned. It runs as follows:

1. Find the first task that needs computation. If found, mark it.
2. If we have a marked task, find an idle thread and run it. If no idle thread is found, keep the task for future computation (just forget that we have found it).
3. If all threads are full or no task needs to be computed and there is at least one working thread, wait till some thread has finished and store its result into the result mask.

4. If there is nothing to compute and all threads are idle, set a flag for all threads that they should die, else continue with step (1).

When the `while` cycle has ended, wait until all threads have ended and then delete the array of communication structures. This is the end of the computation, just the result mask is provided to the user and end.

The threads themselves are primitive. They are implemented in a static method `DijkstraProcessingThread`. The thread sleeps until it gets either data or a flag that it should die. If it gets the flag, it informs the main thread that it has successfully ended and returns. If it gets data, it runs `computeDijkstraInPolarCut`, which is a slightly modified version of `computeDijkstraInCut`. More in Section 6.5.4.

6.5.4 Polar Dijkstra

First let's discuss the polar transformation. For it we have a class `CPolarCut`. This is basically a class for containing a 2D image enhanced with methods for transforming from and to polar coordinates. The image itself is a polar transformation of some 2D image. We can create it either from an array of `float` values or from a specified slice from `CVolumeSet<float>`. The parameters of the image transformation is a resolution of the source image, a resolution in the polar direction, a resolution in the radius direction, starting angle, angle direction (clockwise or counterclockwise) and whether we want the result to use nearest filtering or linear filtering from the source image. The center of the transformation is in the middle of the source image and radius is so big so that the source for the polar image can be inscribed within the source image. See fig.6.3 for an example how it works. All the user defined parameters are stored and can be used for transformation of points to and from polar coordinates, such as control voxels to polar coordinates and the result back to spatial coordinates. All the transformations are available as `public`.

The *Dijkstra's* shortest path search algorithm is then performed on these polar transformed data. There are several methods that do this. In Section 6.5.3 two of those methods were mentioned (`computeDijkstraInCut` and `computeDijkstraInPolarCut`). They are functionally the same, only the data source for them is different. `computeDijkstraInCut` takes volumeset as a source. From one specified slice creates a polar cut and calls the `computeDijkstraInPolarCut` method. This method then selects all control border voxels that are present on the specified slice and transforms them to polar coordinates. The voxels are then sorted according

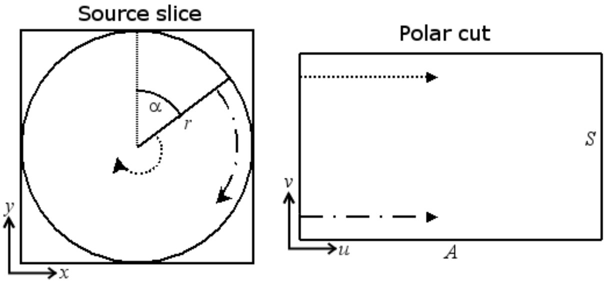


Figure 6.3: Example of a polar transformation(α – starting angle, r – radius of transformation, A – angular resolution, S – samples per radius)

to their angle in spatial coordinates (on fig.6.3 it is the u coordinate). For each consecutive pair (the last voxel is paired with the first) we run `computeDirectionalDijkstra`.

`computeDirectionalDijkstra` works with the linearly interpolated polar transformation of the cost function and two points – start and end. It has two parameters that specify which way it should search for the path, it is able to search along the positive or negative or both directions of the u axis. We need to search only along the positive u axis, because this is the way we have sorted our control points. It starts to *spread* along all the shortest paths from the starting point and stores two types of results: the actual shortest path to each visited voxel and a predecessor for each visited voxel from which was the shortest path. The algorithm can cross along the u coordinate from one end of the image to the other, but cannot cross the v borders. When we hit the end point, we finish the computation and backtrace along the predecessor map back to the starting point and return this path as the final result.

All such obtained paths are then put together and give us a complete path around the original slice, which is then returned back to the segmentation algorithm as the segmentation border.

Chapter 7

Results

7.1 Result utilization

Our program is used by MUDr. Horák and MUDr. Vaculík on Bulovka Hospital to perform a study project that aims at finding a possible link between the femoral head bone density and the treatment success. They use the mean value and standard deviation of our segmented data and try to find a relationship between these values and the successfulness.

At this moment, no new patients are examined with it, the data they process are the data of already treated patients and the success and problems of the surgical operation are already known.

However, MUDr. Horák came with an idea to further utilize the segmented data. When we have the segmentation finished, we can use the data further for another purposes, not only bone quality analysis. On fig.7.2 we can see another application of our algorithm. We have performed standard segmentation as usually, but this time instead of *erosion* we used *dilation* to encase the whole femoral head and removed this data from the volume. Next we can with some 3D visualization software inspect the inside of pelvic acetabulum (for example look for possible fractures, etc.).

7.2 Results showcase

A complete segmentation can be seen for example in fig.3.1. For a more clear representation some sort of 3D data visualization technique is needed. Fig.7.1 shows how a femoral head can look like. For this image we have used

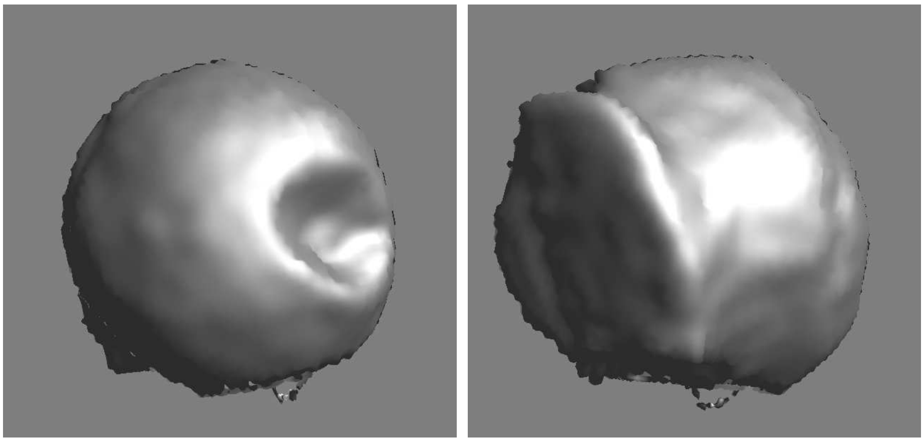


Figure 7.1: A 3D isosurface visualization of a segmented femoral head. Medial side depression is visible on one side, on the other is clear the user-defined cutting plane. Rendered with Simple Volume Renderer (2005, S. Stegmaier, [24]).

an isosurface representation.

7.3 Time complexity

The number of control points does not change the time complexity almost at all. Adding one control point to a slice segmentation is almost for free, sometimes it can even make the segmentation a little bit faster (but not much, see Section 5.2.2 for a detailed description of slice processing).

We have performed a number of tests with one dataset segmentation on a computer with Core2Quad Q6600 2.4GHz. The algorithm is done so that multiple threads can do the computation at the same time. The times in tab.7.1 are the times of all slice segmentations excluding the preprocessing. The graph 7.3 shows the same results.

The preprocessing step is always done in roughly 1.9 seconds on the Q6600 machine. It has not been done in parallel yet, it is planned as a future improvement. The process is always the same for any patient, so the time is almost constant (differences only due to the processor scheduling if some background processes are running). However, preprocessing is done only after changing the center of computation or changing the user defined plane, so it is usually done only once. After that any subsequent segmentation is done without the preprocessing, that is why we did not include the preprocessing step in the timings.

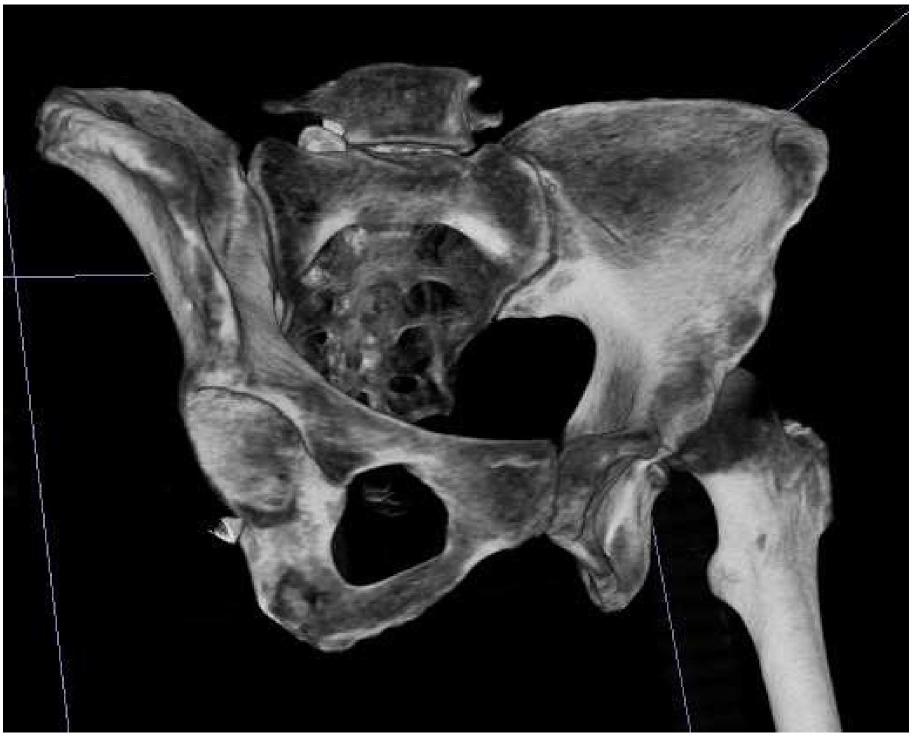


Figure 7.2: Example of another utilization of our algorithm – removing the femoral head for a virtual pelvic acetabulum inspection. Rendered with Pre-Integrated Volume Renderer (2001, K.Engel, [6]).

You can notice that the optimized code does not benefit from additional threads. This is mainly because our implementation had a parent thread that was taking care of storing results and the child threads that performed slice segmentation performed much faster that the parent. So when we used 3 or more threads, the child threads were waiting for the parent thread to clean up after them. This can be even more enhanced by subdividing the result storing to individual child threads. This is also an idea for future work.

	Type	Threads	Time.min	Time.mean	Time.max	Time.sd
1	NoOpt	1	18.64	18.69	18.72	0.03
2	NoOpt	2	9.98	10.02	10.05	0.02
3	NoOpt	3	6.93	7.14	7.33	0.16
4	NoOpt	4	5.80	5.85	5.94	0.04
5	NoOpt	5	5.63	5.68	5.72	0.03
6	Opt	1	7.36	7.38	7.42	0.02
7	Opt	2	4.03	4.07	4.15	0.04
8	Opt	3	3.57	3.60	3.65	0.03
9	Opt	4	3.62	3.66	3.73	0.03
10	Opt	5	3.63	3.66	3.68	0.02

Table 7.1: Times needed for compiler optimized and unoptimized segmenation to run with various thread settings. Preprocessing is not included (always takes 1,9 seconds). *Type* – differentiates optimized code (Opt) and unoptimized code (NoOpt). *Threads* – number of threads used for segmentation. *Time.min* – minimal time needed for segmentation, in seconds. *Time.mean* – average time needed, in seconds. *Time.max* – maximal time needed, in seconds. *Time.sd* – standard deviation of measured times, in seconds.

7.4 Segmentation results

We have segmented 30 different patients. If the fracture was very clear, we segmented only the fractured femur, if the fracture was barely visible or we could not find it at all, we performed segmentation of both legs. The actual results are displayed in table 7.2.

We can summarize that 35% of femoral heads were segmented using only 1-3 control points (see for example fig.7.4). 14% of them need 11 to 20 control points, 6% needed more than 20. 45% of femoral heads needed 4-10 points. Only one patient was not correctly segmented (that result is included between those that needed more than 20 control points).

Patient number 27 needed around 20 control points and even then we did not reach a precise segmentation (fig.7.5 shows the beginning of the segmentation process). This patient had very unclear corticallis over a large area, so we were basically painting it again with control points and even six segmented primary slices were not enough. The segmentation algorithm had a tendency to stick with pelvic corticallis. This fact fortunately implies that we cannot find the corticallis because it is too similar to the trabecular bone, so that including this part of the corticallis in the statistical computation

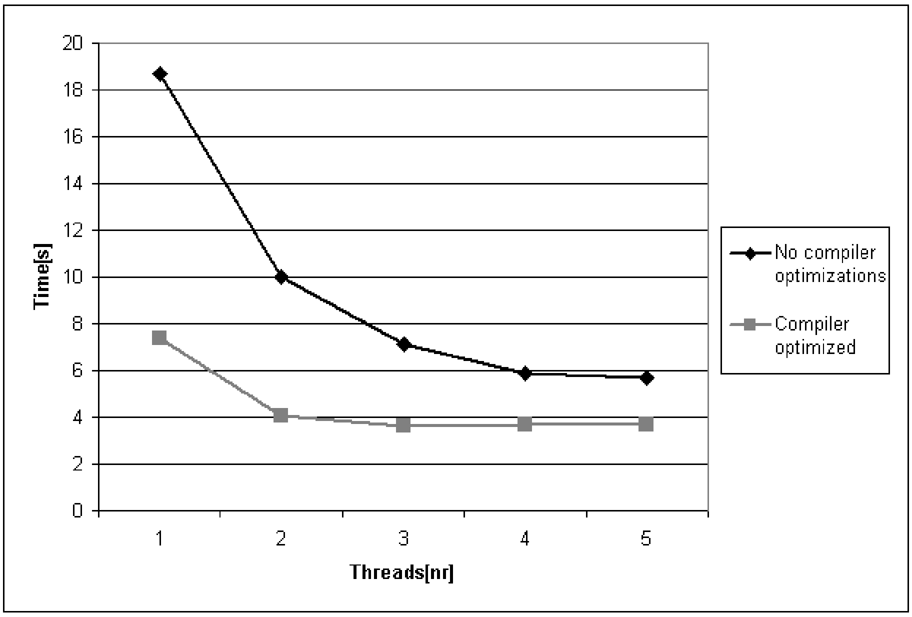


Figure 7.3: Times needed for compiler optimized and unoptimized code to run with various thread settings on a Core2Quad Q6600 2.4GHz. Times used are from table 7.1.

does not spoil the result by more than a few HU units (which is good enough for us).

However, the opposite problem when we segment less than we should segment introduces a more significant error. One slice that leaves out part of the bone introduces a hole in the final segmentation twice as big as the erosion diameter (because of the erosion). So it is important to look for these places and correct them, because such a volume may already bring a significant error to the final bone quality analysis. All femoral heads we have segmented either did not suffer this problem or that problem was easily corrected by one or several correcting control voxels.

It is possible for each of those patients to do much worse segmentation by choosing inconvenient primary slices and control points. The right ones are more or less guessed by experience, the user needs to have some practice with the program to make the right decision. We have not measured the time needed for a user that has enough computer and medical knowledge, but is inexperienced with our method and application to become familiar with it. But a rough guess is to do several (5-10) segmentations and the user will be able to guess the right points. This process may take about half an hour to one hour. The more segmentations the user does, the better the

Patient	Hip bone	Time[m:s]	Pri	Sec	Corr
1	Left	3:45	10	0	2
2	Left	2:09	2	1	10
3	Left	1:18	1	0	2
3	Right	1:57	2	1	3
4	Right	1:30	2	0	0
5	Right	2:55	6	0	0
6	Left	3:09	9	2	0
6	Right	1:35	4	0	0
7	Right	2:02	5	0	0
8	Left	2:37	5	0	5
9	Right	1:25	2	0	0
10	Right	2:14	4	0	0
11	Right	2:12	7	0	0
12	Left	2:20	1	2	5
13	Right	2:52	8	0	0
14	Left	2:42	2	1	2
15	Left	3:22	2	1	8
16	Right	2:31	3	0	5
17	Left	1:16	1	0	0
17	Right	1:40	2	0	0
18	Left	1:20	2	0	0
18	Right	1:11	1	0	0
19	Left	1:11	1	0	0
20	Left	1:36	1	2	0
21	Left	3:12	8	1	0
22	Left	6:00	19	0	4
23	Right	2:35	3	1	2
24	Left	2:45	9	0	0
25	Left	1:45	2	0	0
26	Left	3:22	4	1	3
26	Right	2:34	4	1	5
27	Left	2:56	20	0	0
28	Right	2:25	2	1	0
29	Right	1:36	1	1	0
30	Left	2:25	3	1	7

Table 7.2: Summary of segmentations of 30 patients. *Patient* – ID of a patient. *Hip bone* – shows whether the left or right leg was segmented. *Time* – time in minutes:seconds needed for complete segmentation including data loading and checking the result. *Pri* – number of primary control voxels for sagittal plane segmentation. *Sec* – number of secondary control voxels for frontal plane segmentation. *Corr* – number of correcting control voxels for axial plane segmentation. (*Note:* The right leg of the patient is displayed on the left part of the screen and vice versa. This is because the axial CT slices can be imagined as being looked at from below.)

probability of successful segmentation in a few clicks.

The timings in table 7.2 of the segmentations are measured roughly from the moment the user selects a directory with DICOM files to the moment a final result is obtained by performing final morphological adjustments. So the time *includes:*

1. Initial loading
2. Quick examination of the patient and finding the broken bone
3. Selecting central point and user-defined cutting plane
4. Performing initial segmentation of primary and secondary slices
5. Performing complete segmentation of the whole femoral head
6. Adjusting and adding additional primary and secondary slices and repeating complete segmentation until it is correctly segmented
7. After each segmentation quick examination and error searching
8. Final morphological adjustments

Segmentation that needed only one control point can be seen in figure 7.4. Segmentation of hard to segment data is in 7.5. Even though the segmentation looks almost fine after a few clicks, even after more than 20 clicks there are still errors. The average segmentation process can be seen on figures 7.6 and 7.7. Figure 7.6 shows segmentation of a primary and secondary slice (in this case there is a problem with the structure of the trabecular bone that confuses the algorithm, so more than one control points are needed for a successful segmentation). Figure 7.7 shows additional primary slices added in places with the biggest error. Several additional primary slices (in this case two) are usually enough to provide an amount of information needed for a successful segmentation process in axial slices. The axial slices can be corrected as well, but in this case segmenting two primary slices was enough to lead to a successful segmentation.

Note: The colors in the segmentation images are as follows:

- Greyscale – Original CT data
- Green – Mask representing segmentation result
- Yellow line – User cutting plane

- Yellow cross – Primary slice control border points
- Dark blue – Control points of the user cutting plane
- Orange – Central point of computation
- Red – Position of user point of view

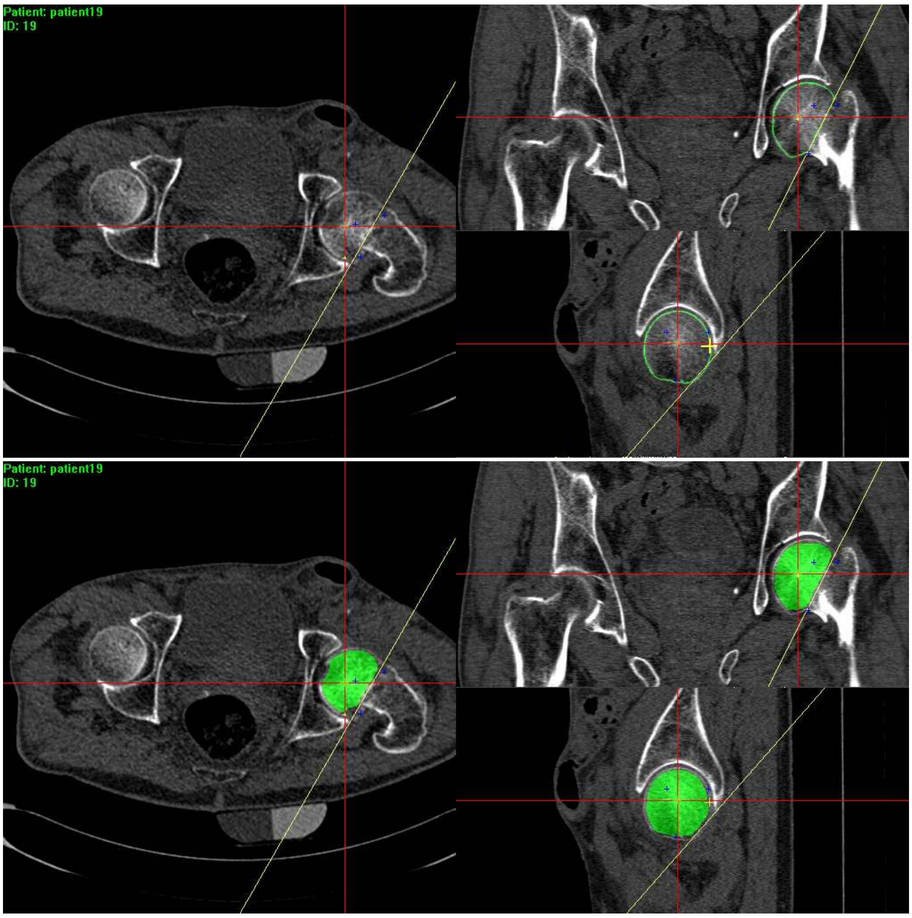


Figure 7.4: Complete segmentation in one click. This is one of our best and fastest results. The corticallis structure is well visible. Upper image shows the primary border voxel placement in the sagittal plane (yellow cross in lower right part) and both sagittal and frontal control segmentations. Lower image shows segmentation after the third phase with slice-by-slice segmentation in axial plane, postprocessing has been applied (3x erosion).

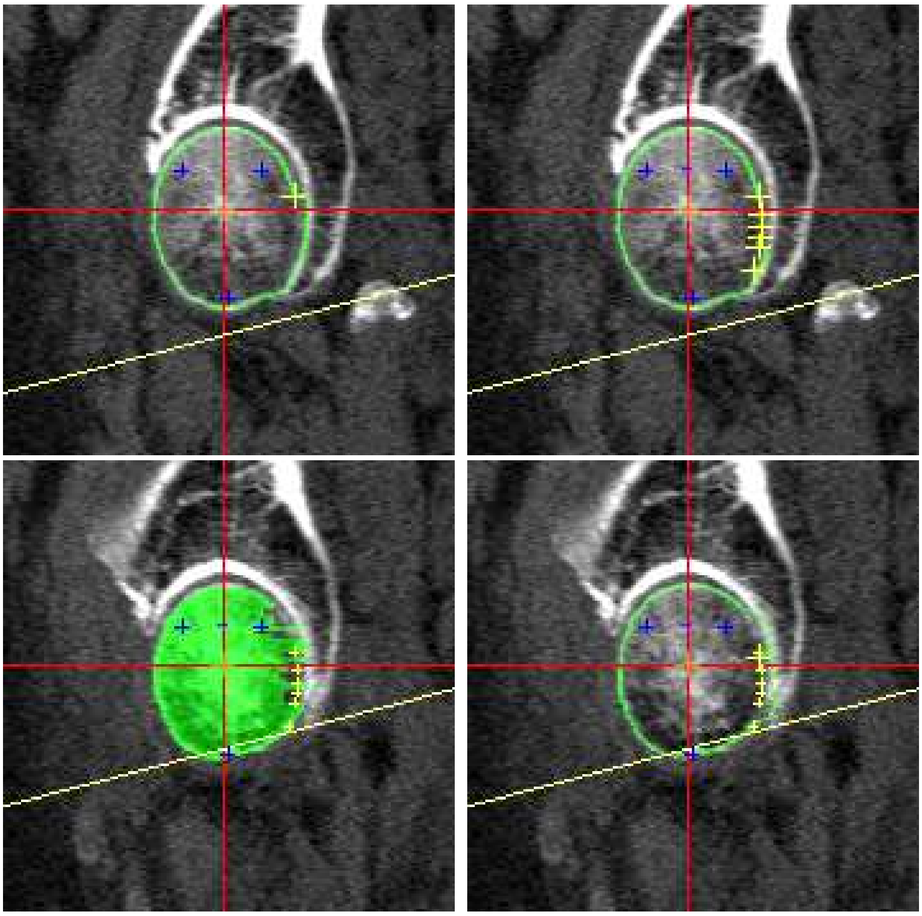


Figure 7.5: Example of hard to segment data and its segmentation. This femoral head was not segmented correctly even after using around 20 control points. You can notice that the algorithm on this dataset has a strong tendency to stick with the pelvic corticallis in the sagittal slice, because the femoral corticallis is locally almost invisible. In axial slices the algorithm escapes in both directions – to the trabecular bone or to the pelvis. Images from top left to bottom right: (a) After setting the first control point – the algorithm escaped to pelvis. (b) Needed control points to correct the slice segmentation. (c) Sagittal slice only a few voxels away from the first one – notice the holes in the segmentation, these are the most dangerous ones, because they form considerable errors after erosion. (d) Trying to add another primary slice for the segmentation of (c) – small yellow crosses are control points from another slice, the control point added in (d) is the bigger one (one of the topmost). For a successful segmentation of this slice we would need another 5-10 points, such as in (a). This correction must be done many times either in sagittal or axial plane to do a correct segmentation for this patient.

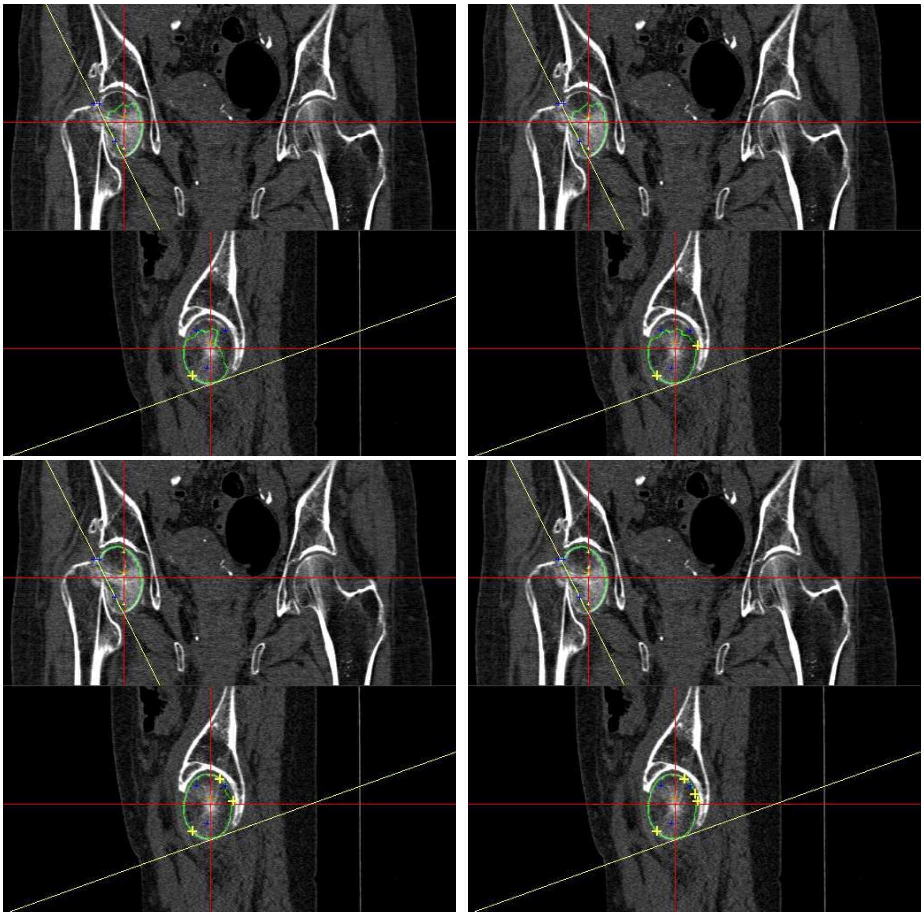


Figure 7.6: Example of an average segmentation process, first control slices. It is normal that there is a significant error after setting the first control point. However, placing 1-3 additional points usually converges to an almost perfect result. Images from top left to bottom right: (a) One click in sagittal plane. (b) Two clicks in sagittal plane – still with significant error. (c) Three clicks in sagittal plane – only a minor error. (d) Four clicks in sagittal plane – correct result for this slice.

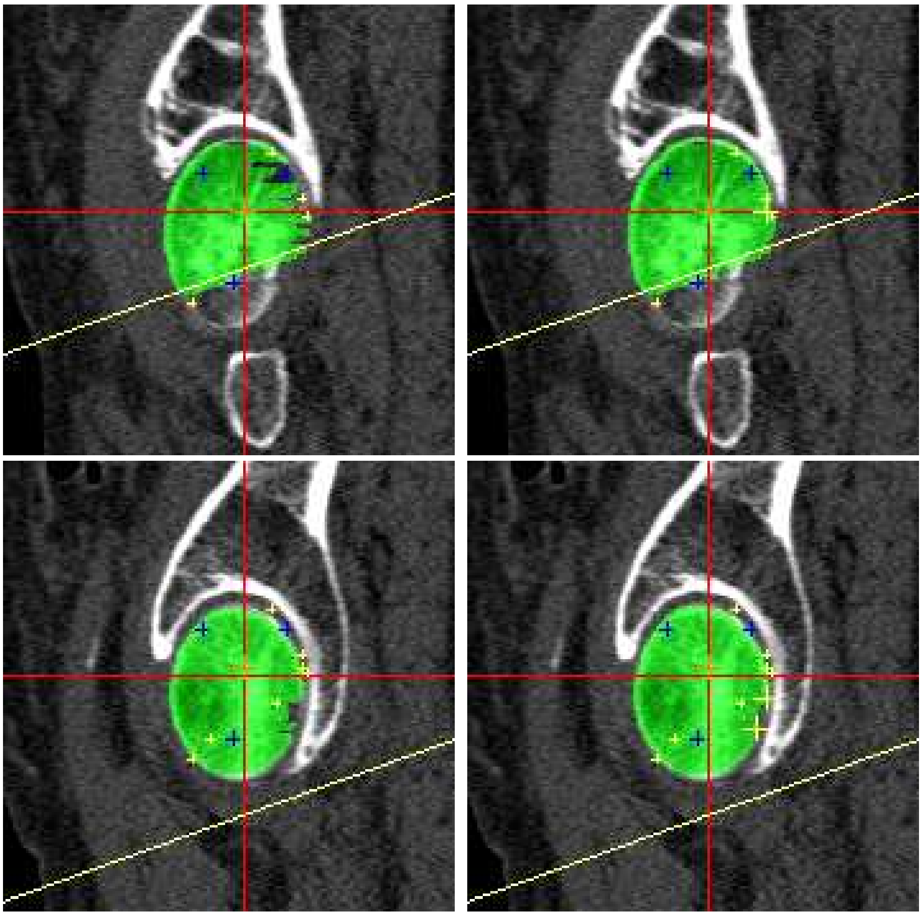


Figure 7.7: Example of an average segmentation process, correcting segmentation errors by introducing one or more additional primary control slices into segmentation from fig.7.6. In this case two additional primary control slices are needed for a correct segmentation. Images from top left to bottom right: (a) Found a sagittal plane that intersects many wrong axial segmentations. (b) Repaired by adding another primary sagittal segmentation with one border voxel. (c) Found another error, this time smaller. (d) Repaired with 2 new primary control voxels.

Chapter 8

Conclusion

We have presented a novel method based on a combination of several existing methods with some advancements in preprocessing of our own. The most difficult part for the implementation (if we consider only the segmentation method, not the user interface) is definitely the Dijkstra's shortest path search algorithm, which is not overly complicated. In this sense we can state that we have found a working method, that is much easier than many existing algorithms, both in the sense of actual program implementation and debugging (here we can compare it to various deformable models algorithms) or in the sense of the first-time data preparation (compared to registration and atlas-based segmentations).

A significant advantage of our algorithm is that we are able to do segmentation without previous estimation of bone positions. That enables segmentation of dislocated femur as well as some complicated fractures. That would be very difficult using some sort of algorithm that relies on the estimated structure (such as atlas-based segmentation).

The time complexity of the algorithm is quite low. Although it is not a realtime algorithm, it runs in a reasonable time of around several seconds (for the *whole* segmentation), one slice segmentation can be considered as a realtime performing algorithm) on a mid-level modern PC. With the advance of modern multicore processors that enter the price level of a normal home PC component this algorithm can be parallelized to utilize this architecture and becomes almost realtime.

The algorithm performs quite well for an average CT dataset, it takes about 3-10 control points to completely segment a femoral head with broken femoral neck and the segmentation is done in about 1,5 to 3 minutes including loading and visual error checking. There are also many patients that can be segmented in 1-3 control points, although less common.

During testing segmentations 97% of patients were correctly segmented (only one was unsuccessful, but the error was not significant for the final statistical analysis).

Our application was used in the RTG clinic of the Bulovka hospital by MUDr. Horák for bone quality estimation during a research project that tried to find a relationship between trabecular bone density and suitability of a patient for screw treatment of a femoral neck fracture.

8.1 Discussion and future work

There are patients that cannot be segmented so well. A major problem for this algorithm (and also many others) is a barely visible cortical bone of the femur and very significant pelvic corticallis. This leads all algorithms that do not incorporate the information about pelvis into confusion. The algorithms have tendency to consider the pelvic corticallis as the femoral corticallis. This may be a major problem for usage that needs precise segmentation (for example pelvic acetabulum inspection, see fig.7.2). But for statistical bone analysis this is not such a big problem, it introduces an error to the computation, but this error is in the order of several Hounsfield units and the smallest difference we are interested about is in the order of tens of HU. This small error is an implication of the reason why this error happens – the corticallis is too similar in density to the trabecular bone. So we cannot precisely find it, but on the other hand it does not spoil the result too much.

Another problem (although not so common), is when the corticallis is too thick and does not create a *bump* in the data, but rather a step between two almost homogenous areas (low density – internal organs, fat, muscles. . . , high density – homogenous bone). This inhibits the ability of our preprocessing step to highlight the corticallis. Fortunately this happened in our data only at about two patients and even then the bone is not so homogenous as to completely hide the cortical part, so one or two additional control points lead the algorithm successfully along the corticallis. So the algorithm may exhibit errors along this part of the bone, but they can be corrected by additional control points. Of course it happens also in the case of some other patients as well, but in a less significant way and many times no additional information is necessary to lead the segmentation correctly, because the neighborhood of the corticallis is much less like our model of corticallis.

During the error inspection it is important to look for places where the segmented region does not contain the whole bone, because even one slice that makes a hole in the segmentation introduces a very significant error after the erosion during postprocessing. This error may influence the final statistical analysis because a large part of the bone is missing. On the other hand, solitary slices that segment much larger area than the bone are simply discarded after the first or second erosion, so it is important to correct them correctly only in the case more of them are close to each other.

If we do not want to change the algorithm at all, there is definitely some space for algorithm parameters optimization. Tweaking the equation for corticallis enhancement function and selecting proper coefficients for the final linear combination of corticallis enhancement method and gradient size. However, this preprocessing step is usually good enough, so it may be advisable to spend effort rather on the segmentation step.

The segmentation step can utilize an information about the smoothness of the path (we chose not to use this information, because there are some places like the small depression on the medial side of the femoral head that need sharp and correct segmentation, but on the other hand smoothness may help us keep the path on the corticallis in cases it has tendency to escape to pelvis.

Next topic is to exhibit some more coherency between individual slices. The final segmentation in axial slices has usually good results, but in the case of about 1 to 5 slices from the usual number of 50 to 80 slices per femoral head, there is tendency to escape the corticallis path and follow some other locally significant feature and the doctor has to correct these results. Maybe some fuzzy channel addition to slices that stick out of the others may help this.

Another idea is to completely replace the segmentation step with something more robust. As has been mentioned, our polar cut shortest path search algorithm was chosen because of its nice properties as easy implementation and quick debugging. The preprocessing step in our method has also some interesting properties and is able to transform very unclear and hard-to-segment data into quite clear data that can be used for many other segmentation methods. So utilizing our preprocessing in a completely different segmentation algorithm is possible.

Also the parallelism of our algorithm was not completely exhausted. There is still a lot of room for parallelization of the preprocessing step and improvement of the parallelization of the segmentation step.

Appendix A

User manual

A.1 Program overview

The program implements our method described in this thesis and several other features. The goal was to present our method to a radiologist that has already a good knowledge of the DICOM file format, is able to work with some other program for CT data examination and has enough knowledge about for example the *window function* and other techniques needed for this type of work.

The overall layout of the program is not completely polished to look as a professional commercial program, but we tried to do it as user friendly as possible. On the other hand effort to do the user interface was not given at the cost of the actual method implementation. The basic process of both *spherical* segmentation and *channeling* segmentation can be done without entering the program's menu. All necessary controls are on the toolbar (fig.A.1). The program's menu duplicates the function of the toolbar and adds some other features (such as for debugging).

The toolbar is divided into five separate groups. The first group contains only the command *Open...*. The second group contains all necessary *tools* for working with DICOM data. The third group contains commands for starting *channeling* segmentation. The fourth group is only a button for eroding the surface of the segmented data by one voxel over a 6-neighborhood and finally the last group contains a button for exiting the program.

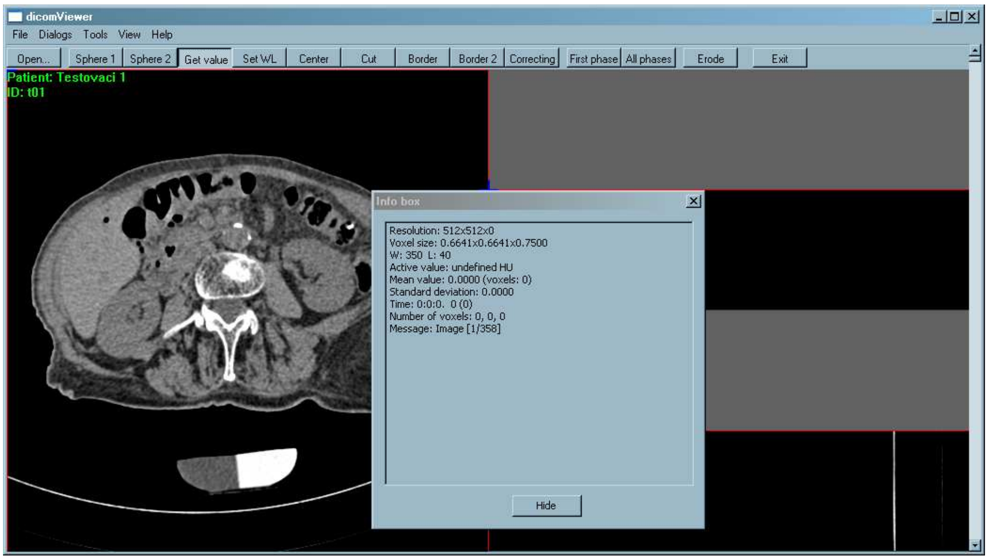
Figure A.1: Program toolbar and menu.

A.2 Study examination

A.2.1 Opening the study

The basic need for the function of the program is that the study is stored in *one* directory, the DICOM files themselves are alphabetically sorted and no other files are present in that directory. This is the way most of the CT scanners save these files when exported, but if it happens that more studies are merged into one directory, then some means of separating them is needed (we used for example *DicomWorks*, see [4]).

For opening a study, select *Open. . .* on the toolbar. A dialog box appears to load a file. Go into the directory where the actual *.dcm* files are stored and select any of them and press *Open* (or an equivalent in your system language). An *Info box* appears and the program now parses the whole directory and looks for all files that can be loaded as DICOM files. The screen after loading may look like fig.A.2.



```
Resolution: 512x512x0
Voxel size: 0.6641x0.6641x0.7500
W: 350  L: 40
Active value: undefined HU
Mean value: 0.0000 (voxels: 0)
Standard deviation: 0.0000
Time: 0:0:0.  0 (0)
Number of voxels: 0, 0, 0
Message: Image [1/358]
```

Figure A.2: Program layout after loading a DICOM directory.

The *info box* in the middle of the screen displays all information that can be described as text. That means various information about the properties of the loaded data, results from computations, active *window* function, time needed for last *channeling* segmentation or a context information that displays a line of text dependent on current activity (for example number of currently viewed axial slice, phase of data processing or segmentation, etc.).

Behind it there is the client area divided into three parts. The left half is the *axial* slice. The right half is divided into upper and lower quarter, upper quarter is the *frontal* slice and lower quarter is the *sagittal* slice. All these show 12-bit data from the DICOM set processed by a *window* function into displayable grayscale 8-bit images and various other information with colored lines and pixels displayed on top of it.

Note: all interaction with mouse and keyboard must be done when the main window has focus. When another window has focus, for example the info or other dialog, all mouse and keyboard interaction goes to that dialog and is usually discarded.

A.2.2 Toolbar

For examination, we need only three buttons from the toolbar: *Get value*, *Set WL* and *Center*.

Get value is a tool for getting a value in Hounsfield units at the point where the user clicks with left button. Choose this tool in the toolbar and then click anywhere in the client area (on any of the three slices). The result is displayed in the info area as *Active value.* If you click outside the defined area, active value displays *undefined HU.*

Set WL is a tool for manual setting the *window function* (there is also a possibility for setting one of preset window functions, see Section A.2.3). Click on the *Set WL* tool, then left click and drag the mouse inside the client area. Dragging along the X axis changes the *width* of the window function, dragging along the Y axis changes the *level* of the window function (parameter explanation is in Section 6.3.2).

Center is a tool for setting the center of all three slices. Select *Center* in the toolbar, then click anywhere in the client area (any of the three slices). The program automatically finds out the 3D coordinates where did the user click and centers all three slices on this position. For example if you roll through the axial slices to the femoral head and then select the *center* position inside the femoral head, then both frontal and sagittal slices will recompute as centered on the femoral head. This tool also sets the center for the *channeling* computation.

A.2.3 Keyboard

Keyboard commands are very simple and are only for examination purposes.

Arrow Up, *Arrow Down*, *PageUp* and *PageDown* are for movement through the axial slices. Arrow up/down are for moving to the previous and next axial slice. PageUp and PageDown jump over 10 slices in the same direction as arrow keys.

F2 - F9 and F11 - F12 are for selecting preset window functions:

F2 – W:256 L:127 – debug for viewing the values 0-255

F3 – W:350 L:40 – chest

F4 – W:350 L:1 – abdomen/pelvis

F5 – W:1200 L:-600 – lungs

F6 – W:80 L:40 – brain

F7 – W:2500 L:480 – bones

F8 – W:350 L:90 – head/neck

F9 – W:1500 L:400 – bones

F11 – W:800 L:250 – angio

F12 – W:800 L:100 – angio

The only keys left are *Escape* for exiting the application and *I* for displaying and hiding information about the examined patient.

A.2.4 Mouse

Mouse is used for almost all interaction, the whole application can be controlled only with mouse. It is important that the main window has focus, otherwise all mouse interaction goes to another window (for example dialog windows).

Basic interaction is with *mouse wheel*. For mouse wheel is important also the position of the cursor. If the cursor is above the axial slice or outside the main window, then mouse wheel controls the active axial slice. Rolling the wheel up goes to previous slices, rolling it down goes to next slices. If the cursor is above the frontal or sagittal slice, the wheel controls the respective slice the same way (wheel up goes to previous, wheel down goes to next slices in the series in given direction). In this sense the mouse wheel duplicates the functionality of up/down arrow keys and PageUp and PageDown keys.

Next possibility to move even faster through the data is by using the *vertical scrollbar*. The scrollbar represents the stack of axial slices. So intuitively by dragging the scroll box we move to the respective axial slice.

A very fast way to look at the femoral head may look like this:

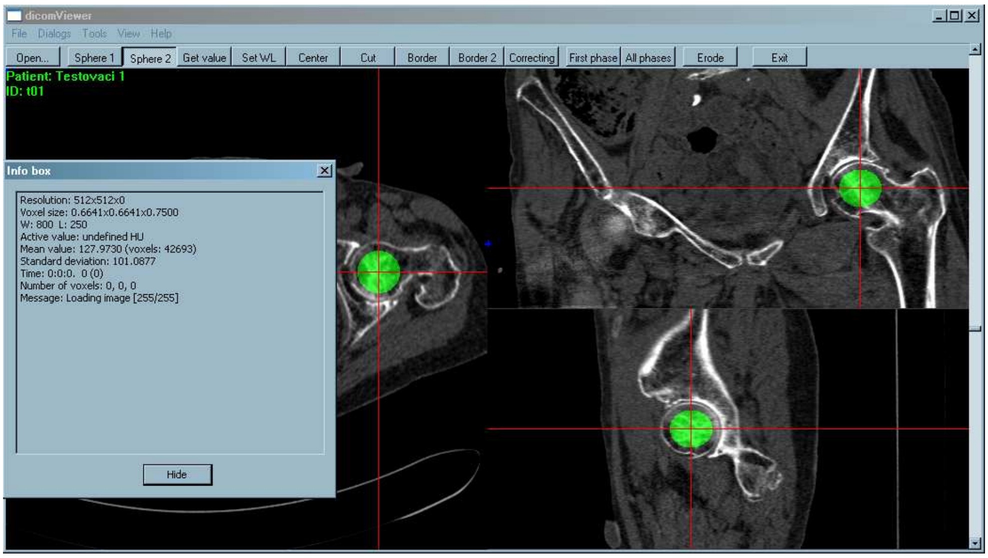
1. Open DICOM series
2. Drag the scroll box to about $\frac{2}{3}$ length of the scrollbar (that is approximately the position of the femoral head in most of the studies)
3. More precisely hit the center of the femoral head with rolling the mouse wheel when the mouse is hovering above the axial slice
4. Select *Center* tool in the toolbar
5. Click in the middle of the femoral head displayed on the axial slice
6. Now all three slices should display a slice through the femoral head

A.3 Spherical segmentation

Spherical segmentation was the first feature implemented in this program, because MUDr. Horák and MUDr. Vaculík needed at leat a rough approximation of the complete segmentation as soon as possible. It is also very simple and since the first version the user interface for this changed a little bit to better suit the doctor's needs.

There are predefined two diameters of spheres – 2cm and 3cm (they are in Parameters Dialog given as radii 1cm and 1.5cm, see Section A.5.3 for more reference on parameters).

Spherical segmentation is done by selecting tool *Sphere 1* or *Sphere 2* in the toolbar, then clicking anywhere in the client area. Sphere 1 is usually 2cm in diameter, sphere 2 is usually 3cm. On the clicked spot a spherical



Patient: Testovaci 1
ID: t01

```
Resolution: 512x512x0
Voxel size: 0.6641x0.6641x0.7500
W: 800  L: 250
Active value: undefined HU
Mean value: 127.9730 (voxels: 42693)
Standard deviation: 101.0877
Time: 0:0:0.  0 (0)
Number of voxels: 0, 0, 0
Message: Loading image [255/255]
```

Figure A.3: Example of a spherical segmentation.

mask of given size is created and also displayed. All voxels that fall within this mask are computed and their *mean value* and *standard deviation* are written in the Info Dialog under their respective names.

Both spherical measurements also change the point of view in the data, that means that they center all three slices on the sphere. You can see how the program centers the data and a spherical segmentation example on fig.A.3.

A.4 Channeling segmentation

Channelling segmentation is the main part of the whole program and is the actual implementation of the method described in this thesis. The method consists of three phases, two of which are performed at the same time. The first phase segments several primary slices in the sagittal plane, second phase segments one secondary slice in the frontal plane. These two phases are performed at once. The third phase segments each axial slice that has some seed points from the first two phases and fills the result, so that the final volume is a solid, not only a border. For details read Chapter 5 for theoretical background and Section 6.5.3 for implementation details.

For a successful segmentation look into Appendix B for a quick guide with images on how to do the channeling segmentation. If the segmentation went well and was correctly done, after eroding you can see the result of the segmentation in the Info Dialog as the *mean value* and *standard deviation.*

A.5 Other program features reference

A.5.1 Patient info

Patient info is written in green letters in the upper left corner of the client area. You can notice this in most example figures throughout this thesis. We are using anonymized data, so the name is represented by a number. The second line is the ID of the patient. In our case it is the same number as the "name" of the patient. But in case the doctor uses this on real data, both these lines display actual information about the patient. This feature was incorporated on request by MUDr. Horák.

A.5.2 Menu

Menu contains mostly the same functionality as the toolbar and adds a number of various debugging features not directly needed for the segmentation process done in a hospital.

File menu

- **Open ...** – Opens a directory containing a DICOM study – the same as *Open...* in toolbar.
- **Export 3D data to RAW ...** – This is an additional feature, that takes the currently loaded DICOM study and sets all voxels behind the user plane to 0, all voxels marked by a mask to 0 and all other voxels converts with the current window function to 8-bit values and stores it in the current directory in a format suitable for Preintegrated volume renderer, [6]. This is in a state of debugging, so no dialog box for storing location is opened and the data are stored as two files in the directory we were loading the data from. This feature is not considered final.
- **Exit** – Exits the program.

Dialogs menu (more about each dialog box in Section A.5.3):

- **Show info box** – Shows an info dialog box that displays various information and segmentation results.

Show parameters box – Shows a dialog box for editing parameters such as the diameter of the spherical segmentation, number of computational threads, etc.

Show graph – Shows a window with a graph representing a line between two points during dragging the *Get Value* tool.

Show volume set – Shows a window that displays a `CVolumeSet`, currently unused

Show polar cut – Shows a window that displays a `CPolarCut`, currently unused

Show histogram – Shows a window that presents a histogram computed from the most recently segmented data (histogram represents only the segmented data).

Tools menu:

Get value – Selects tool *Get Value* for immediate 12-bit value getting.

Set computation center – Selects tool *Center* for setting point of view in the data and at the same time setting the computational center for *channeling* segmentation.

Adjust window function – Selects tool *Set WL* for setting *window* transformation function.

Set cut plane – Selects tool *Cut* for setting user-defined cutting plane for the femoral neck.

Get sphere 1 – Selects tool *Sphere 1* for spherical segmentation, default diameter 2cm.

Get sphere 2 – Selects tool *Sphere 2* for spherical segmentation, default diameter 3cm.

Set primary border points – Selects tool *Border* for setting and removing primary border points for sagittal slices segmentation.

Set secondary border points – Selects tool *Border 2* for setting and removing secondary border points for frontal slice segmentation.

Set correcting border points – Selects tool *Correcting* for setting and removing correcting border points for final axial segmentations.

Compute all phases – Invokes computation of all segmentation phases (sagittal, frontal and all axial slices) and final filling of the segmented volume.

Compute first phase – Invokes computation of control slices segmentation (sagittal and frontal slices segmentation).

Erode mask – erodes (reduces) the segmented mask by one voxel over a 6-neighborhood.

Dilate mask – dilates (enlarges) the segmented mask by one voxel over a 6-neighborhood.

View menu:

Show cut plane – Shows/hides dark blue points representing the user-defined cutting plane control points.

Show cut halfspace – Shows/hides yellow line representing the user-defined cutting plane.

Show display position – Shows/hides red lines representing the position of other two slices in each displayed slice.

Show mask – Shows/hides segmented mask (works on mask from both spherical and channeling segmentation).

Show dragged line – Shows/hides a line between which was dragged the mouse using the *Get Value* tool (values on this line are then sent to the Graph Dialog).

Show info – Shows/hides basic information about the patient (name, ID).

Help menu:

About – Shows a dialog box informing about the author and current version.

A.5.3 Dialog boxes

Except for Info Dialog (which is displayed automatically), all other dialog boxes are only for debugging purposes. They show various information about the data properties or are able to display intermediate results of various computations.

Info Box is the most important dialog in the program. It displays the following information (given by the order they are written in the dialog):

Resolution – The resolution of the slices, a typical value is 512x512x12, that means one slice has resolution 512^2 and each voxel is stored with 12 bits.

Voxel size – The size of one voxel, given in millimeters, typically around 0.6x0.6x0.5.

W – Window function *width*.

L – Window function *level*.

Active value – Value stored in a voxel where user clicked with the *Get Value* tool.

Mean value – Mean value of all voxels from a segmentation.

Standard deviation – Standard deviation of all voxels from a segmentation.

Time – Time needed for last *channeling* segmentation excluding the preprocessing step.

Number of borders – Gives the numbers of primary, secondary and correcting control voxels used for the last *channeling* segmentation.

Message – Displays various information, either about the loading or an information about the segmentation process, etc.

Parameters Box is a dialog for changing several program parameters. These parameters mean:

Measurement radius in cm – Sets the radius of tool *Sphere 1* for spherical segmentation.

Measurement 2 radius in cm – Sets the radius of tool *Sphere 2* for spherical segmentation.

File cache size – Number of axial slices that are kept in memory. Do not change this value unless you know what you are doing, may lead to program instability.

Number of threads – Number of threads that run in parallel during the *channeling* segmentation. An optimal value is number of processors plus one (3 for dual core, 5 for quad core processors)

Graph window shows a graph of actual values obtained from dragging the *Get Value* tool in the axial slice. Bottom of the window represents -1023, top of the screen represents 3095. This is for debug purposes only.

Volume set dialog can display a small volume from the program. It is filled with data only during specific actions inside the program, mostly displays nothing. If some data are displayed, then *mouse wheel* is for moving through these data, left click switches between XY, XZ and YZ cuts through these data.

Polar cut dialog is for debug displaying of some specific polar cut from the computation phase. Currently unused.

Histogram window displays the histogram of segmented data. This is for bone quality analysis described in Section 3.4. A very simple way to present this window is: open this window in menu, then select tool *Sphere 1* or *Sphere 2* and click anywhere in the CT data. The histogram window should display the histogram of the segmented data. Try experimenting with sphere position. This histogram also displays the result of our *channeling* segmentation the same way as spherical segmentation.

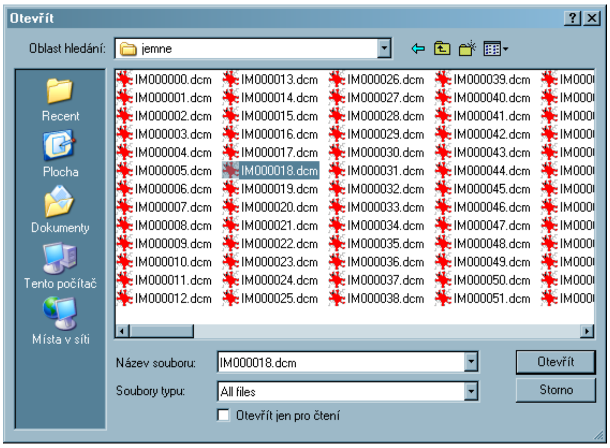
Appendix B

Quick guide

This chapter is for easy and quick explanation of *channeling* segmentation process in our application.

B.1 Open DICOM study

Run the program and then select *Open...* in the toolbar. In the Open dialog box (fig.B.1) go into the directory of target patient pelvis study, select any file and press Open.



IM000000.dcm
IM000001.dcm
IM000002.dcm
IM000003.dcm
IM000004.dcm
IM000005.dcm
IM000006.dcm
IM000007.dcm
IM000008.dcm
IM000009.dcm
IM000010.dcm
IM000011.dcm
IM000012.dcm
IM000013.dcm
IM000014.dcm
IM000015.dcm
IM000016.dcm
IM000017.dcm
IM000018.dcm
IM000019.dcm
IM000020.dcm
IM000021.dcm
IM000022.dcm
IM000023.dcm
IM000024.dcm
IM000025.dcm
IM000026.dcm
IM000027.dcm
IM000028.dcm
IM000029.dcm
IM000030.dcm
IM000031.dcm
IM000032.dcm
IM000033.dcm
IM000034.dcm
IM000035.dcm
IM000036.dcm
IM000037.dcm
IM000038.dcm
IM000039.dcm
IM000040.dcm
IM000041.dcm
IM000042.dcm
IM000043.dcm
IM000044.dcm
IM000045.dcm
IM000046.dcm
IM000047.dcm
IM000048.dcm
IM000049.dcm
IM000050.dcm
IM000051.dcm

Název souboru: IM000018.dcm
Soubory typu: All files

Figure B.1: Open dialog. Find directory with patient study, select any file and press Open (or equivalent in your language).

B.2 Set center of computation

When the data is loaded, press F7 or F9 to adjust the window function for bone inspection (for better clarity). Then scroll through (either by mouse wheel or by dragging scroll box to around two thirds of the scrollbar length) to find the femoral head. Then choose tool *Center* in the toolbar and click approximately in the middle of it on the axial slice (fig.B.2). Then correct the position on sagittal and frontal slices, so that it is approximately in the geometrical center.

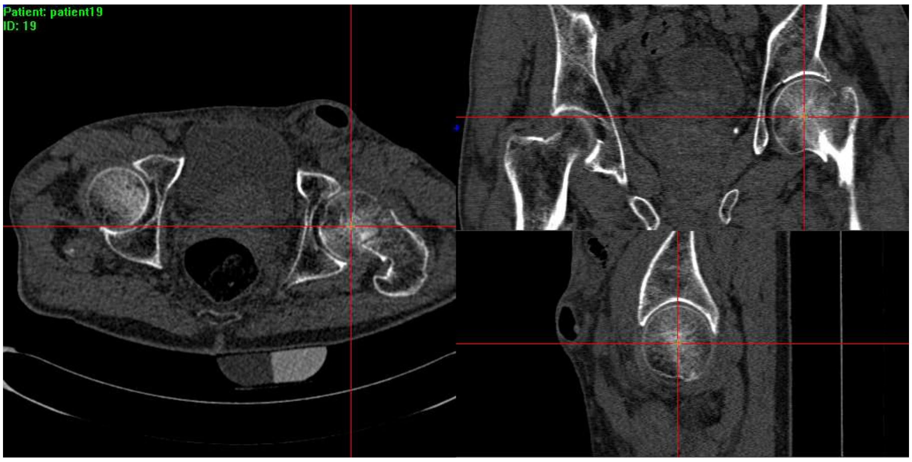


Figure B.2: Choose center of computation on axial slice. Then correct the position on frontal and sagittal slices.

B.3 Set user-defined cut

Scroll through the data to upper part of the femoral head and find the place where the femoral neck ends. This is an ideal place to set the first two control points of the user-defined cutting plane (see fig.B.3). Then scroll to lower part of the femoral head, and place the third control point of the cutting plane (fig.B.4).

B.4 Segmenting the primary slices

Go back approximately to the center of the computation. The primary segmentation is in the sagittal plane, several sagittal slices can be segmented.

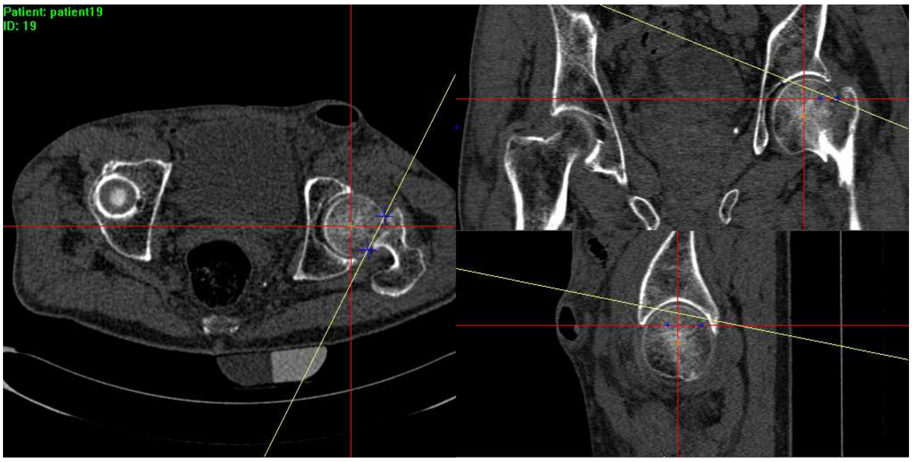


Figure B.3: Setting the first two control points of the user defined cut plane.

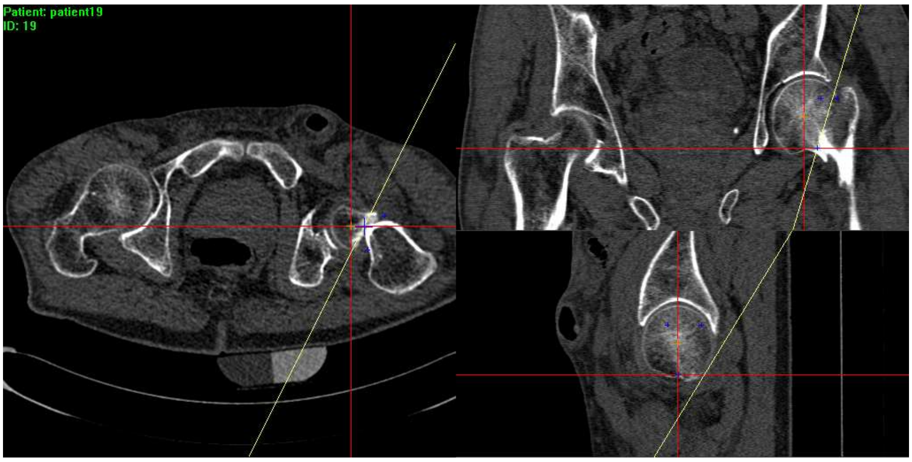


Figure B.4: Setting the third control point of the user defined cut plane.

So select *Border* tool in the toolbar and click once on the femoral corticallis in the sagittal plane and click on *First phase* in the toolbar to check, whether the result is alright. If it is wrong, correct it with additional border points (fig. B.5). You can always check the result by running *First phase*.

When the first primary slice is done, check whether the secondary segmentation went correctly – it is done always after the primary segmentation. The only secondary slice has such position that it contains the center of computation. If there are errors, you can correct them by selecting *Border 2* and clicking on the wrong parts (fig.B.6). Again, you can check the result with *First phase* tool in the toolbar.

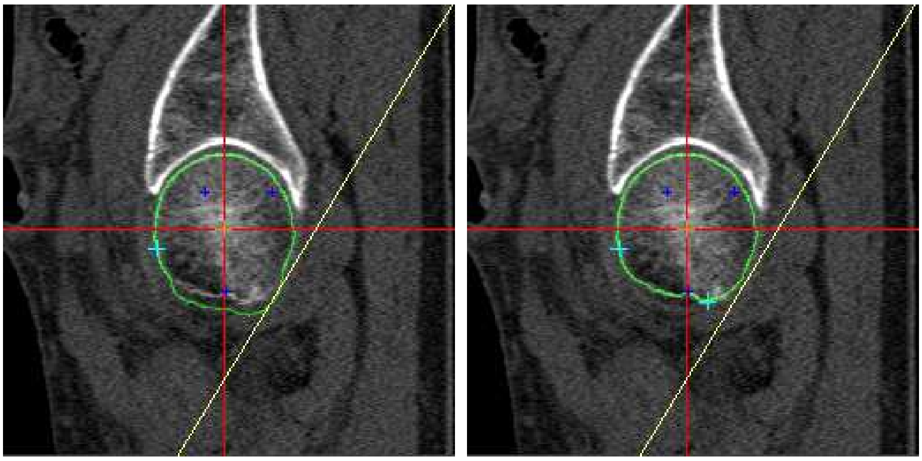


Figure B.5: Primary segmentation in sagittal plane. Done with tool *Border* from the toolbar.

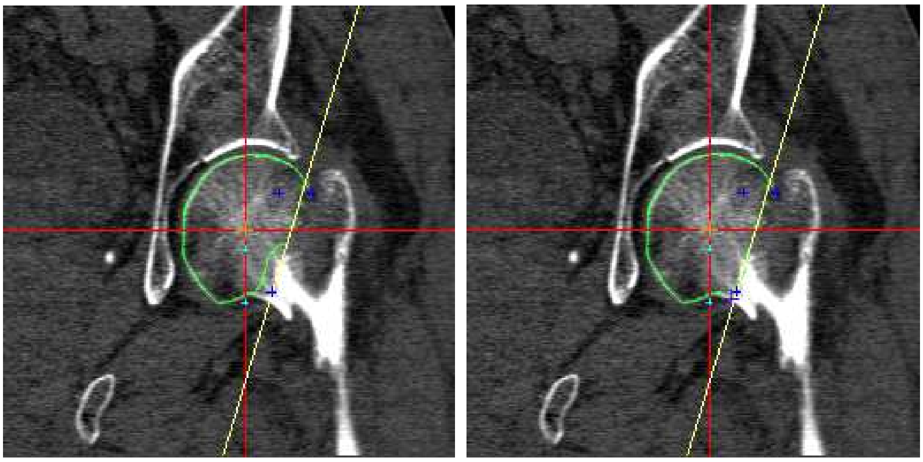


Figure B.6: Secondary segmentation in frontal plane plane. Done with tool *Border 2* from the toolbar. This is only for correcting errors, secondary segmentation is run always after the first and mostly without errors or with only minor corrections needed.

B.5 Complete segmentation

When the primary segmentations are done, you can proceed with the whole segmentation. This is simply done with clicking on *All phases* in the toolbar. The result can be seen in fig.B.7. Now it is important to check on axial slices each segmented slice of the volume whether it has been done correctly. Quickly scroll through all the axial slices containing the femoral head and

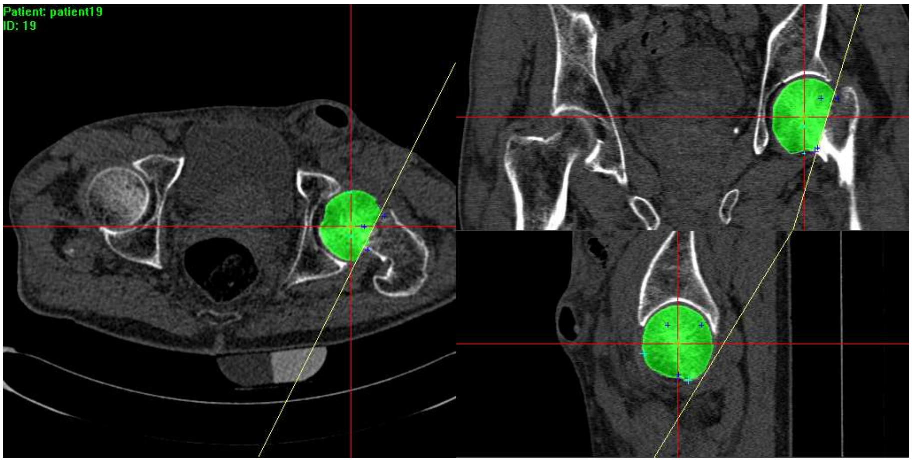


Figure B.7: Final segmentation. In this case correct, but if needed, can be corrected on individual axial slices with tool *Correcting*.

you should be able to visually distinguish correctly segmented slices from the ones with errors.

In our case no errors were found, but it may happen (and it mostly does) that there are errors. They may origin in a strong trabecular bone structure or weak corticallis. In that case, check how many slices are affected. If it is only a singularity in one or two slices, select tool *Correcting* and click on each of those slices on places where it needs correction (in axial plane) and then check result with *All phases* whether it is enough.

If the error propagates along the sagittal direction into more slices, it is advisable to use another sagittal segmentation. Easily move the mouse pointer over the sagittal slice and then scroll with mouse wheel the sagittal plane into the position of those errors. Then perform another primary segmentation as described in Section B.4 and shown in fig.B.5. Then again check with *First phase* until this new primary segmentation is correct, then perform full segmentation with *All phases* and continue this way until the whole volume is correctly segmented.

B.6 Postprocessing

When the femoral head is correctly segmented, press several times the *Erode* tool to remove the corticallis (around 2x-3x). When this is done, the final result is displayed in the *Info Box*. If you eroded too much, just press *All phases* to return to the original segmentation.

Appendix C

Contents of DVD

bin – Executable version of our program

data – Samples of pelvic CT scans saved in DICOM format

doc – PDF and PS versions of this thesis

src – Source files needed for compiling the application

  - **libs** – Library files of DCMTK 3.5.4, both debug and release versions
  - **res** – Resource files
  - **src** – Application C++ source code including headers for DCMTK 3.5.4
  - **testing** – Debug directory for testing

Bibliography

[1] Čihák, R.:*Anatomie 1*, Grada Publishing, 256-266, Praha, 2001

[2] DCMTK: *DICOM Toolkit*, http://dicom.offis.de/, 2007

[3] DICOM: *Digital Imaging and Communications in Medicine*, http://medical.nema.org/, 2008

[4] Puech, P., Boussel, L.: *DicomWorks*, http://dicom.online.fr/, 2008

[5] Dungl P., a kolektiv: *Ortopedie*, Grada Publishing, 917-949, Praha, 2005

[6] Engel, K., Kraus, M., Ertl, T.:*High- Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, Los Angeles, 2001

[7] Ferda, J., Novák, M., Kreuzberg, B.:*Výpočetní tomografie*, Galén, 2002

[8] Gonzales, R.C., Woods, R.E.: *Digital Image Processing, Third edition*, Prentice Hall, Upper Saddle River, NJ, 2008

[9] Gonzales, R.C., Woods, R.E., Eddins, S.L.: *Digital Image Processing Using MATLAB*, Prentice Hall, Upper Saddle River, NJ, 2004

[10] Haralick, R.M., Shapiro, L.G.:*Image segmentation techniques*, Comput. Vis. Graph. Im. Proc., 29, 100-132, 1985

[11] Kass, M., Witkin, A., Terzopulos, D.:*Snakes: Active contour models*, International Journal of Computer Vision, 1(4): 321-331, 1988

[12] Krajíček, V.:*Měření objemu v 3D datech*, Diploma thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2007

[13] Krajíček, V., Pelikán, J., Horák, M.: *Measuring and segmentation in CT data using deformable models*, WSCG, 15:149, 2007

[14] Lakare, S.:*3D Segmentation Techniques for Medical Volumes*, State University of New York at Stony Brook, 2000

[15] Liu, H.K.:*Two and Three Dimensional Boundary Detection*, Computer Graphics and Image Processing, 6: 123-134, Apr. 1977

[16] Maintz,J.B.A.,Viergever,M.A.:*A Survey of Medical Image Registration*, Medical Image Analysis, 2: 1-36, 1998

[17] Pettersson, J., Borga, M., Knutsson, H.:*Some Issues on the Segmentation of the Femur in CT Data*, Linköping University, Department of Biomedical Engineering & Center for Medical Image Science and Visualization, Linköping, Sweden

[18] Pettersson, J., Knutsson, H., Borga, M.:*Automatic Hip Bone Segmentation Using Non-Rigid Registration*, Linköping University, Department of Biomedical Engineering & Center for Medical Image Science and Visualization, Linköping, Sweden

[19] Pham, D.L., Xu, C., Prince, J.L.: *A Survey of Current Methods in Medical Image Segmentation*, Annual Review of Biomedical Engineering, Baltimore, 1998

[20] Pohle, R., Toennies, K.D.: *Segmentation of medical images using adaptive region growing*, Otto-von-Guericke University, Magdeburg, 2001

[21] Rohlfing, T., Brandt, R., Menzel, R., Maurer, C.R.Jr.: *Segmentation of Three-dimensional Images Using Non-Rigid Registration: Methods and Validation with Application to Confocal Microscopy Images of Bee Brains*, Medical Imaging 2003: Image Processing, San Diego, 2003

[22] Sahoo, P.K., Soltani, S, Wong, A.K.C.:*A survey of thresholding techniques*, Comput. Vis. Graph. Im. Proc., 41, 233-260, 1988

[23] Sezgin, M., Sankur, B.:*Survey over image thresholding techniques and quantitative performance evaluation*, Journal of Electronic Imaging 13(1), 146-165 (January 2004), 2004

[24] Stegmaier, S., Strengert, M., Klein, T., Ertl, T.:*A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting*, Proceedings of Volume Graphics 2005, Stony Brook, New York, USA, 187-195, 2005

[25] Szeliski, R.:*Bayesian modeling of uncertainty in low-level vision*, International Journal on Computer Vision, 5: 271-301, 1990

[26] Terzopulos, D., Fleischer, K.:*Deformable models*, The Visual Computer, 4(6): 306-331, Dec 1988

[27] Xuan,J., Adali, T., Wang, Y.:*Segmentation of Magnetic Resonance Brain Image: Integrating Region Growing and Edge Detection*, Proceedings of the 1995 International Conference on Image Processing, 1995

[28] Zhang, Y., Matuszewski, B.J., Shark, L.K.:*A Novel Medical Image Segmentation Method using Dynamic Programming*, Medical Information Visualization – BioMedical Visualization, 2007

[29] Zhang, Y., Qu, H., Wang, Y.:*Adaptive Image Segmentation based on Fast Thresholding and Image Merging*, Proceedings of the 16th International Conference on Artificial Reality and Telexistence–Workshops, 2006