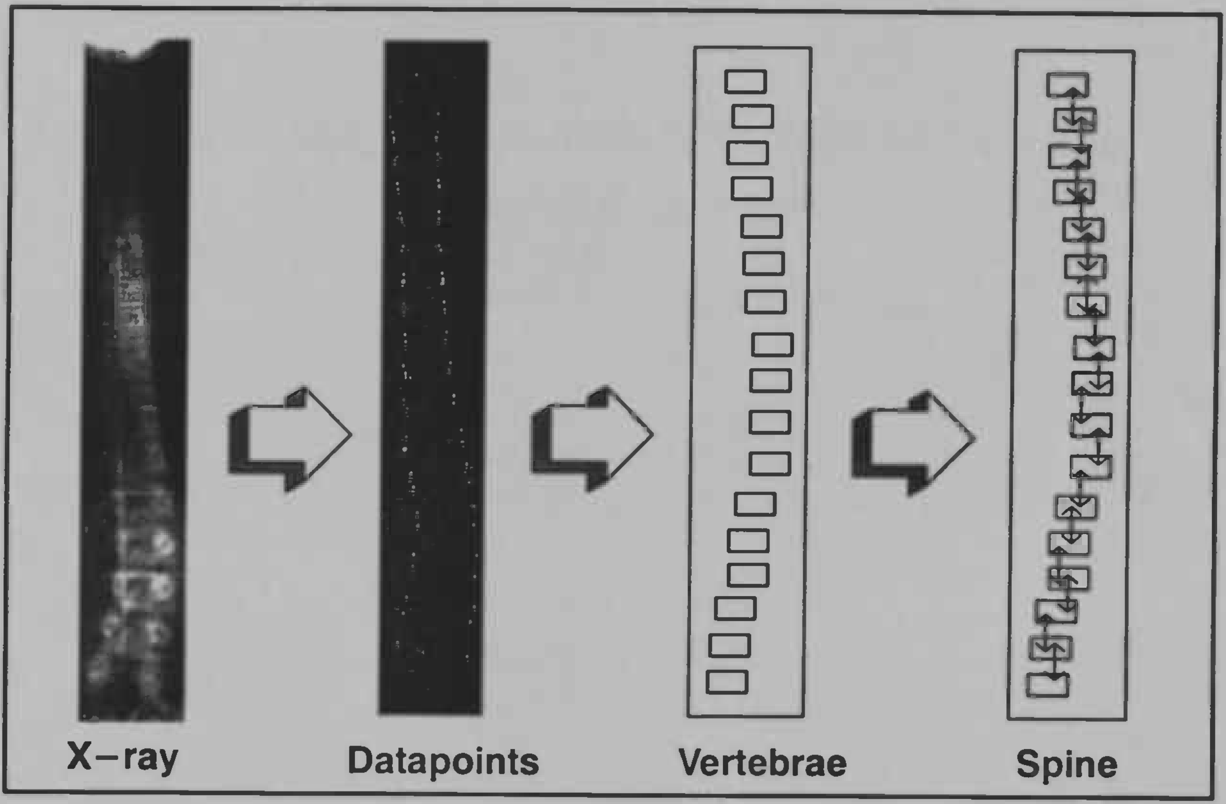
University of Groningen **Faculty of Mathematics and Physical Sciences**

Department of Computing Science

Fuzzy Merging Techniques for Creating 3D Models of the Spine



Harm P. Brouwer

begeleiders: J.A.G. Nijhuis
L. Spaanenburg

augustus 1995

RuG

Fuzzy Merging Techniques for Creating 3D Models of the Spine

Master's thesis, department of Computer Science, University of Groningen

Harm Popke Brouwer
Student Computer Science
University of Groningen

Supporters RuG:
dr. ir. J. A. G. Nijhuis
prof. dr. ir. L. Spaanenburg

External supporters:
ir. C. M. de Bakker (Holec Projects)
dr. A. G. Veldhuizen (AZG)
E. Bakker (Orthin)

Abstract

Scoliosis patients are children in the age of seven to sixteen, who suffer from a curvature of the spine, along with a torsion component (a rotation to the left or the right). In order to be able to treat these patients a 'brace' has to be constructed, this is a corset that forces the body to attain a better posture. Currently this brace is being constructed using a series of (two dimensional) X–ray images of the spine and a plaster cast of the back of the patients. This method is however still far from perfect, and can be improved on many fronts using, amongst others, fuzzy techniques.

The first part of this paper describes the search for new imaging techniques that can possibly be used for improving the method. The mean selection criteria have been: the patient must endure minimal stress and minimal risc, the imaging time must be fast and the resulting image of the spine must be accurate.

There appeared to be no technique that could meet all the demands, accuracy of the techniques being the main stumbling block. Therefore an attempt was made to try and find a solution by combining data from several promising imaging techniques, namely: ultrasonografy, rastering stereografy and possible in a later stage thermografy and optic tomografy in order to accomplish the required accuracy.

In order to construct a syntactic model of the spine, I used data from X–ray images of the spine and foreknowledge on the shape of vertebrae and the spine. Doing this I defined a fuzzy model for 'approximate vertebrae', which models the vertebrae as 'approximate rectangles', and using the structural relations between the vertebrae, I constructed the fuzzy model for 'approximate spines'. This model is shaped like a fuzzy binary tree and can be parsed by fuzzy root–to–frontier tree automata. Both these concepts are explained thoroughly in the paper.

Because this model is tailor made for interpreting data that represents different views of the spine emerging from the X–ray images, merging different models in order to construct new fuzzy binary tree models has become very simple and elegant. As an example of this ease of combining, I combined two different models representing two different (two dimensional) views of the spine to a new model representing a three dimensional view of the spine, along with the new corresponding three dimensional 'approximate vertebrae' ('approximate cubes') and 'approximate spines'. Results are visualised using a viewer that can interpret three dimensional structures.

Preface

From: Te veel vertering, Herman Finkers

Ik kocht voor de grap wat patat.
De prijs was een gulden exact,
en als ik iets overdrijf
was de prijs één gulden vijf...

Even Herman Finkers knows how to use fuzzy reasoning. Help!

This thesis report is the result of research performed at the 'University of Groningen', the Netherlands, in partial fulfillment of the Master's Degree in Computational Science. The project was part of a joint research project by the 'Academic Hospital of Groningen', Sissing Electotechnique/ Holec Projects B. V., Orthin (orthopeadic instrument makers), and the University of Groningen, and focusses on developing and implementing new methods for imaging the human spine under the influence of scoliosis. Software is written in C++ and runs on a Unix platform that supports X–windows and X–Motif.

During the time I worked on this project I learned more than I could have ever imagined to be possible, and had a great time both at the same time! Therefore I want to thank some people. First of all I want to thank Jos Nijhuis, my supporter at the RuG, for his never failing optimism and his great ideas. I want to thank Ben Spaanenburg, my other supporter at the RuG, for his ability of pin–pointing every weak part in a way of reasoning (never failing pessimism?). I also want to thank the people on 'the other end' of the project, C. de Bakker from Holec Projects, A. G. Veldhuizen of the AZG, and E. Bakker of Orthin, not only for giving me every possible cooperation and freedom, but especially for taking me and my ideas very seriously thus making me a fully–fledged partner in the project. I want to thank my fellow students at the RuG for all the discussing, laughing, helping, debugging and relaxing. Special thanks go to Hans Seinhorst and Erik Grave for letting me use their viewer and assisting me in adapting it for my special purposes.

Finally I want to thank you, Krista, for being there and supporting me, even in the last few weeks, when a day might have lasted longer than only 24 hours.

Drachten,
juli 1995,
Harm P. Brouwer.

Table of Contents

Problem Description

At the orthopeadic section of the Academic Hospital in Groningen patients with spinal–deformities are being treated. A mayor part of these patients consists of scoliosis patients. Scoliosis patients are children in the age of seven to sixteen or seventeen years old, who are still growing. These patients suffer from a curvature of the spine along with a torsion component (a twist to the left and/ or right).

In order to be able to treat the patients, a special kind of corset has to be constructed, the so called 'brace', which forces the body of the patient to attain a better posture. In time this will result in a correction of the deformation. In order to construct this brace for a certain patient currently a series of two–dimensional X–ray pictures are being made. Then these pictures are used to make an optimal functioning and fitting brace. However, a number of problems still arise:

- The large number of X–ray pictures that are needed result in a higher chance of cancer. It has to be taken into consideration that in the case of scoliosis–patients the brace sometimes has to be adjusted monthly on the basis of a new series of pictures.
- Patients with serious spinal deformities sometimes are not able to stand still for more than a few seconds, which causes movement artifacts in the X–ray pictures. Still, these pictures are needed for constructing the brace and the artifacts can therefore result in errors in the brace.
- The X–ray pictures are the input for a large number of technical calculations, eventually resulting in the optimal functioning and fitting brace. The extension from two dimensional images to the three dimensional brace is far from perfect in the current settings.
- The current imaging method, where X–ray pictures sometimes have to be taken monthly and from different angles (this is time– and physically intensive) is bad for the patients.
- The costs that go with taking the large number of X–ray pictures are enormous. On top of that the technical calculations and drawings which are necessary for construction the brace go to great expense as well.
- In the current settings it is hardly possible to efficiently measure the progress in the correction and curing of the patient.

In order to find a solution for these problems the Academic Hospital has decided to consult the companies Sissing and Orthin which has resulted in a combined research project. The focus of this project will be on the replacement of the disadvantaging X–ray technology by other methods. Fuzzy techniques may be used for image improvement, pattern recognition and classification. In this report these problems will be investigated, possible solutions will be suggested, and after a choice has been made, an implementation will be presented.

CHAPTER 1:

Choosing an Imaging Technique

1.1 The selection criteria

The executive company Sissing Electrotechnique, part of Holec Projects, has divided the project into several parts. The first part consists of searching and investigating new techniques, then a choice is made amongst those techniques and the final part is making an implentation that actually uses the technique. For choosing this new technique the search mainly extends towards 'novel' applications of imaging techniques. This means that more commonly known techniques such as (N)MRI, PET, MEG and CT, which may be able to solve the problem adequately, are not considered candidates.

In order to choose which new technique can possibly be used for making a three dimensional image of the spine, a few selection criteria have to be stated. Major candidates are those techniques that are already in use for imaging purposes, and are not or hardly harmful. The next selection criterion is the applicability of the technique for our specific problem. Because Sissing has neither the time nor the facilities to engineer a complete imaging system it is important that there exists an application of the technique in the form of an apparatus that can be purchased. This application does not necessarily have to produce the required results exactly, but should produce enough information to reconstruct an image of the spine. The speed of the method is also important. This speed criterion concerns the time necessary for scanning the patient. The shorter this time, the less physically intensive the technique. This last criterion will probably, together with economical aspects, turn the scale when there are several techniques that can possibly be used. Summarising the goal is to find a method that is better than the currently used one: For the construction of a brace, a plaster cast of the back of the patient is made twice a year. This provides the instrument–maker with a reference model for the three dimensional shape of the spine. This model is corrected by studying the X–ray scan. The position of the pelvis, the *C7* point (position of the seventh cervical vertebra) and the form of the spine play an important role in this construction. The required accuracy of these positions regarding the construction of the brace is an error smaller than 1 – 1,5 centimeters.

This chapter describes the most important candidate techniques; how they work, their advantages and disadvantages and how they are put into practice. Eventually a technique has to be chosen. The best case solution would be a technique that is completely harmless and preferably very fast and cheap as well. The worst case would be that no harmless technique can be found. In this case we would have to use an X–ray technique that minimalises the amount of harmful radiation used. The techniques investigated in this chapter are: optic tomography, thermography, radar, ultrasonography, raster stereography and (soft) X–ray tomography.

1.2 Optic tomography

As is often the case with new concepts there exists some confusion what optic tomography exactly is. Actually optic tomography is a collective noun: optic means in this context ‘having a relation with light beams or working with them’ and tomography means ‘reconstruction from projections’ (derived from the Greek *τόμοσσ*–cut and *γϱάφειν*–writing respectively). Regarding the medical applications one can say that light is used for making cross–sections of an object without actually having to cut or damage the object. There exist several tomographical techniques that can be considered optic:

- scanning with shutter cameras or stroboscopic light
- laserpulstechnology

Because of optic tomography ‘seeing through someone’ has at last gotten a literal meaning. That is, it has become possible to investigate the interior of the body using ordinary light. A common example used to demonstrate this is the torch which is held against the hand, resulting in a pink glow. Because the light is strongly scattered, the inside of the hand is hardly visible. However, apart from all the scattered light, there are always a few rays of light that travel through the hand nonscattered. Those rays give us information on the interior of the hand. The fast part of the beam of light changes direction quite oftenly, thus obscuring the silhouette of the bones.

When trying to solve this problem one could add a strong light–absorbing substance to the tissue. This makes use of the knowledge that scattered light must travel a longer way than light that is not scattered, and therefore becomes more absorbed. A combination of strong beams of light and highly sensitive detectors can then be used to make sure that the sporadic rays of light that pass through the tissue can be detected. A different solution would be making use of the fact that light that is not scattered needs a shorter time to move through the hand than scattered light. This time difference is very small, because the photons travel at the speed of light (3.0 x 10^9 *m/sec* in air) through only a few centimeters of tissue. This is the reason why very short laserpulses are being used, of about a few picoseconds long (10^{-9} *sec*), and detectors that open at exactly the right moment to register the first puls of nonscattered photons.

Advantages of optic tomography:

- Tissue and thin bones like the skull transmit (infrared) light rather well.
- The equipment used for optic tomography (lasers) is small, portable and cheap (compared to MRI– and PET–scanners).
- Light can be endured in large doses, this in contrast to X–rays.
- The scanning method is fast, therefore there is less chance on artifacts caused by movement of the patients. There is even the possibility to keep track of a dynamic process, such as the oxygen grade in the brain.
- Timing the delays of the pulses gives a possibility to calculate the optic density of the tissue; using this density it is even possible to detect different tissues (organs).

Disadvantages of optic tomography:

- Thicker bones are transmit very little light, resulting in a very bad quality of the scans. This is a mayor disadvantage!

The advantages of optic tomography could very possibly solve the problems existing with the currently used imaging technique. The disadvantages to overcome are however considerable.

The question remains if the quality of the acquired optic images is sufficient for using them as a base for the calculations needed to construct the brace. The state of affairs is that a permeating depth of about 11 centimeters has been registered; this is not enough for our purposes.

1.2.1 Optic tomography in practice

Though optic tomography is a relative young imaging technique, there already exist several usable applications. Optic tomography has already successfully been used in the field of opthalmology ([1], [2]), the detection off small tumors in breast tissue ([3], [4]), the scanning of the bones in the hand ([5]), the transillumination of small mammals ([6]) and the (dynamic) detection of the oxygen grade in the brain ([7]). Also the further possibilities of optic tomography are still intensively investigated by several researchers ([8], [9], [10], [11], [12]). Overall can be concluded that optic tomography has the advantage that many visualisation techniques initially developed for use with other imaging techniques such as CT, MRI, PET and MEG ([13]) can easily be adapted to interpret the optically generated information.

1.3 Thermography

Thermography is a technique that is used for making the temperature of several objects visible. It does so by making use of the fact that every object emits light. The wavelength of that light is dependent on the temperature of the object.

This dependency is described in the radiation law of Planck:

$$E = \frac{2\ \pi\ \ c^2}{\lambda^5 \left(\exp\left(\frac{h\,c}{\lambda\,k\,T}\right) - 1\right)} \tag{i}$$

with:

E: emitted radiation (*W/m*^{2} *mm*),
T: absolute temperature (*K*),
c: speed of light (*m/s*),
k: Boltzmanns constant (1.380662 10^{-23} *J/K*),
h: Plancks constant (6.626176 10^{-34} J_q),
l: wavelength of the emitted light (*mm*).

After measuring the radiation (light emitting from the object), calculating the temperature of the object is rather simple. The formula implies that, for normal temperatures, most of the emitted light is in the infrared spectrum. When we integrate (i) along the wavelength *l* we find the well known law of Stefan–Boltzmann:

$$q = \int_0^\infty E\ d\lambda\ = \omega\ T^4 \tag{ii}$$

with: ω: Stefan–Boltzmann's constant (5.67032 10^{-8} *J/m*^{2} *K*^{4}).

A simplified way to describe a thermographic camera would be that it is a videocamera that operates in the infrared spectrum. There are however some differences:

- Because one measures the infrared light emitted by the object itself, a thermographic image will not show shadows. This in contrast with an ordinary videocamera that registers the (sun)light that is reflected by the objects.

- An ideal thermografic camera will measure q as described in the law of Stefan–Boltzmann. The cameras that are actually used only detect a limited wavelength interval, therefore the integration should be along that interval because the calculated the temperature is dependent on this interval. The larger this interval, the more accurate the registration of the temperature is.
- Because different objects can have different emission–characteristics this has to be taken along in the the law of Stefan–Boltzmann (ii):

$$q = \varepsilon\ T^4 \tag{iii}$$

with: ε the emessitiveness of the material.

- Instead of lenses made of glass, expensive lenses made of semiconductor materials like Selenium, Germanium or ZincSelenium have to be used. This is because glass is not penetratable for infrared light of wavelengths higher than 2 millimeters, while the semiconductor materials are.
- The lens of a thermographic camera has to be cooled. This is a direct result of the fundamental laws of thermodynamics: Heat transport, also in the form of radiation, always travels from a hot to a cold source. In order to be able to measure the radiation, the temperature of the detector has to be much lower than the temperature of the object that emits the radiation. This cooling is done with liquid nitrogen (–180 $^o C$) or with Peltier cells (–70 $^o C$).

1.3.1 Thermography in practice

Using thermographic techniques does not seem something that is done in everyday life. However, appearances are deceptive, our normal videocamera actually acts on the same principle. Videocameras can do without cooled lenses however, because the temperature of the source of radiation, the sun, is very high (5727 $^o C$), so a detector at room temperature would receive enough radiation in order to be able to construct an image.

Already in the sixties people started applying thermography for preventive maintenance, to prevent interruption or breakdown of energy suppliers, and for localisation of recalcitrant materials. Precondition is that the defects reveal themselves by small or sometimes drastic changes in the operational temperature. A mayor advantage of infrared thermography is the fact that measurements can be taken from a distance, so production processes do not have to be stopped but can be monitored in progress.

Less known, but for our purpose more interesting are the medical applications of thermography. These can be divided into human and vetenary medicine. All the applications use the fact that several physical diseases and inflammations are the cause of a change in temperature which can be detected with thermography. Some examples are: The detection of tumors in the breast that lie just under the skin; because of the many veins that lie in the breast differences in temperature can be made visible. The detection of the cause for bad healing bone fractures. The course of the disease of rheumatism. The detection of the size of burns and frosts.

Advantages of thermography:

- Thermography is a fast method, recordings can be made within fractions of a second.
- It is a safe method, only emitted radiation is detected, the method is therefore completely harmless.
- Measurements can be made without having to make physical contact with the object, the camera can be placed in every desired position.
- Thermography is relatively cheap.
- It can be used for preventive and predictive purposes.

Disadvantages of thermography:

- It only visualises the temperature at the surface of an object; this is a mayor disadvantage because we want to determine the position of the spine which lies beneath the skin.
- It is not a diagnostic medium.

The main stumbling–block for applying thermography for the detection of the position of the spine is the fact that thermography only visualises surface temperatures. Take for instance a thermographic image of the hand, the vessels near the surface will be clearly visible because their temperature is higher than that of the rest of the skin. The underlying bones will however not be visible because in normal conditions they will have the same temperature as the surrounding tissue. This also goes for a thermogram of the back, the spine does not show itself because it does not have a temperature different from that of the surrounding tissue.

Things would be different if the assumption holds that bones react different on heating than the surrounding tissues do. At the Free University of Brussels, prof. J. Cornelis has done some research that may lead to a solution ([14]). A special technique has been developed where patients are irradiated with microwaves, resulting in a slight heating. Instead of taking one thermographic image, a time series of thermographic images is made. When tissues react differently to this heating this should show in the time series, and already this has proved to work, resulting in the detection of cancerous tumors. Research in the thermographic field still continuous, attemps are made to visualize other tissues. Hopefully the technique will also reveal the internal structure of the spine in a harmless but accurate way.

1.4 Radar

The letters **'radar'** are short for **radio detecting and ranging**. A radar is an device that by means of transmitting radiowaves and the detection of the reflection of those radiowaves by an object can determine direction and distance of that object. Conventional radar techniques are not well suited for an visualisation of the spine. A practical aspect is that the wavelength of the radiowaves that are usually used is simply too large to acquire sufficient spatial resolution to see the spine. The result, if any, would be very unreliable. Another disadvantage is that the interpretation of the data, produced by recording the reflections, takes a lot of time–consuming calculations. The so called *FMCW*–technique can be used for making frequency spectrums and uses radiowaves of wavelengths short enough to be able to detect different tissues. The computational load makes this technique too expensive to use for our purpose (costs can add up to millions).

1.5 Ultrasonography

Ultrasonic sound consists of soundwaves with a such a high frequency that they cannot be heard by the human ear, thus frequencies higher than 20 *MHz*. As is the case with radar an ultrasonic imaging device consists of a transmitter and a receiver, used for detecting and ranging an object. The waves used this time are ultrasonic soundwaves. Because the speed of sound is much lower than the speed of light, it is still possible to acquire enough spatial resolution when sound of high frequencies is used. Simple mathematics show us that waves of a frequency of 20 *MHz* traveling at the speed of sound (340 *Km/H*) have a wavelength of approximately 4.7 *mm*, this in contrast with waves of the same frequency travelling at the speed of light which have a wavelength of approximately 1.5 kilometers! As will become clear below, ultrasonography proves to be very promising for the visualisation of the spine. However there does not yet exist an application that can be immediately used.

Advantages of ultrasonography

- The technique is cheap, especially the use of simple transducers is quite cheap, an array of transducers costs more but does more preprocessing.
- The technique is harmless, the sound waves used have very little energy, therefore they can do very little damage passing through tissue.
- Measurements can be done without making contact when using a laserbeam as the carrier of the sound waves. This results in a higher recording rate and a much higher accuracy in a series of recordings because the relative distance between the positions where two adjacent scans are made and the orientation of the scans is known.

Disadvantages of ultrasonography

- Speed is a problem, although ultrasonic recordings can be made rather quickly (about twenty images a second) this is not yet fast enough to reduce the total scanning time for patients to an acceptable one (this should be less than 2 seconds).
- Measuring without making contact using a laserbeam cannot be applied for our purpose, because it may damage the skin of the patient. Another possibility would be to place the patient in a small water filled bath, the water then supplies the medium needed to transport the soundwaves. However, this method is considered to be too strenuous for the patients, and therefore not applicable.
- Ultrasonic waves that travel through the air hardly penetrate the body, but are mostly reflected by the skin. Sensors therefore have to be held against the skin, or travel through a medium that lets the soundwaves penetrate deep enough to be able to tell something about the interior, in our case the position of the spine. When holding the sensors against the skin this probably has to be done by a human operator, moving the sensors along the back. The result would become very much dependent on the skills of the operator and would thus be much less reliable. Scanning by human operators would also not be fast enough.

1.5.1 Ultrasonography in practice

Medical applications of ultrasonography are commonly used in the field of gynaecology, for prenatal investigations of the foetus and the ovaries. There are also other applications known, for instance in the field of opthalmology where ultrasonography is used to examine the eyes, or when ultrasonography is used for constructing images of the intestals, these images can be used for detecting digestive diseases.

The ultrasonic methods still are very promising, further research should tell us if the disadvantages really can't be overcome. The speed disadvantage could be dealt with by using multiple sensors, making it possible to take multiple measurements at the same time and thus reducing the total scan time. The problem of finding a way to measure without making contact, or find another solution that is acceptable for the patients and still gives enough reliability, must also be solved.

In this research TNO could possibly take a leading role. They have the technology and the experience to research different ways of using ultrasonography that show potential in solving the problems stated above:

- Making a vertical scan of a patient that is lying in a small water filled bath using a linear array of ultrasonic transducers. This should result in a (quasi) three dimensional image of the spine with an estimated scanning time of seven seconds.
- After attaching a referential lattice made of material that reflects ultrasonic waves, the same method as above is used. The resulting image would then show a (quasi) three dimensional image of the spine and the referential lattice. This has the advantage that movements that patients make during the scan can be corrected afterwards.
- Replacement the water bath with another medium that also can act as a carrier of the ultrasonic waves but is less strenuous for the patient.

1.6 Raster stereography

A stereographic method is a method that simultaneously makes two two dimensional pictures of an object from two different angles. These two pictures can then be viewed binoculair, resulting in a three dimensional image. Rastering stereography uses a lattice of known dimensions which is projected onto the object to show the three dimensional structure of the object on a two dimensional medium (often a terminal), this can be compared to moiré topografy ([15], [16]), where light due to interference projects shadows at known distances from the light source.

Rastering stereography has enough advantages to already be applied in practice: Using two binocular images of the back a three dimensional image is constructed. This three dimensional information is usually visualised using a raster and by shading the image. The recording of the images is done in a fraction of a second, the method is completely harmless, and the information about the shape of the lower back is very accurate. The shape of the back gives a lot of information about the location of the spinal column, the back part of the spine is even visible. The existing applications use this information to calculate the position of the spine. This position is however not very accurate. The reasons for this inaccuracy can be muscular tensions suggesting for instance a lateral deviation of the spine that does not exist. Another reason is that the position of the spine can be determined from only one side. Information about torsion in the spine is therefore not very accurate. This is a mayor disadvantage, because when trying to accomplish an optimal functionality for the brace this information is indispensable. The method is usable to give quickly and safely an indication of the approximate three dimensional position of the spine. This reference may then be completed and made more accurate by using another method that is able of supplying the missing information. This is in fact what the orthopeadic instrument maker does, using the plaster cast of the back as the initial model of the brace, followed by the use of an X–ray image of the spine for further refinement of the brace.

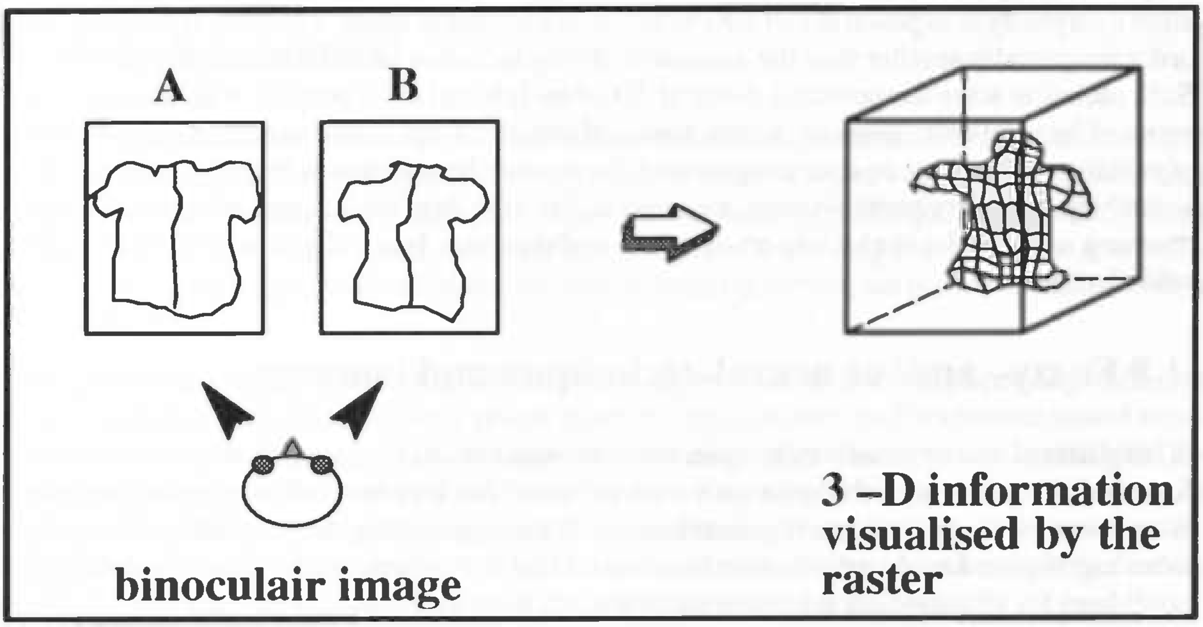


Figure 1: Stereographic vision

1.6.1 Raster stereography in practice

As stated above, raster stereography can provide us with information on the position of the spine. This takes us a step further; how to retrieve this data. Quantec is a company that manufactures and sells systems for rastering stereography also known as computerised surface topography. When testing their systems they have compared their calculations of Cobb–angles and their own Q(uantec)–angles, measures that are used by experts to determine the shape of curved spines, to the ones calculated from X–ray images. This means they can possibly provide us with a combination of stereographic images and X–ray images, that can be used for the development and the testing of a new imaging system.

1.7 Soft X–ray tomography

Soft X–ray tomography is mainly applied in the field of safety–systems at airports for checking people and luggage, and in the food–industry for quality controls ([17], [18]). As with normal X–ray computed tomography (CT), the method uses the electromagnetic waves of short wavelength that were discovered by Röntgen in 1895. These rays are produced when fast moving electrons hit a solid target and can be visualised on fluorescent screen or film. The difference with the normal X–ray tomography is that the radiation used is much ‘softer’, that is, the intensity of the used X–rays is much lower. Though X–rays were of immediate clinical value, the hazardous side–effects of the radiation caused the ever ongoing development of the technology in terms of more detailed information while using less radiation. The use of soft X–ray tomography would be the logical next step in this process.

With the equipment used for the X–ray investigations nowadays, safety comes first. The radiation source only generates X–rays when and where they are needed. This minimalises the chance on radiation leakage and contamination. The amount of radiation coming from an ordinary soft X–ray device is about 100 *mRem/h*. Comparing this with the natural background radiation everybody is exposed to (10 m*Rem/h*) this is acceptably small. The dose is for instance six order magnitudes smaller than the amount of X–ray radiation used for the sterilisation of food. Still, radiation stays dangerous: a doses of 400 *Rem* is lethal in 50 percent of all the cases as the result of internal destruction of organs, tissue, blood, etc. Frightening as this may be, the danger of radiation should not be over exaggerated. It can even be said that radiation is a natural thing, something that every human being is exposed to, day after day (e.g. because of traces of radiation reaching us from the sun). Only an overdose is dangerous. Ironically enough the same can be said of vitamines.

1.8 Fuzzy– and/ or neural–techniques and imaging

Using fuzzy– and or neural–techniques for imaging purposes isn't new. Practice has shown that fuzzy– and / or neural–techniques are very well suited for, amongst others, pattern recognition, morphologies, classification and generalisation. When researching the possibilities for applying these techniques, I made a distinction between related techniques, applications that are possible candidates for a fuzzy–neural approach and already existing imaging applications using fuzzy–neural techniques.

An example of an application that could very well be implemented using fuzzy techniques is described in ([19]). In the second part of this masters thesis a morphometric algorithm is described that calculates the position of the spine and the separate vertebrae. This algorithm is then used to further calculate the amount of *osteoporosis*, using the areas of the vertebrae and the bone density.

A comparison of different fuzzy– and neural–techniques is given in ([20]). Three different applications for segmenting MRI–scans are compared here. The segmentation is used to automatically detect different tissue–groups and to artificially color these groups. This results in a better visual interpretation by the specialist. The fuzzy techniques used to accomplish this segmentation are described more extensively in ([21]). A comparable application of neural–techniques for recognising and classifying so–called discrete 3D scenes (a series of two–dimensional cross–sections) is given in ([22]). Here it is attempted to imitate the human visualisation system in order to be able to interpret the scenes. Especially remarkable, but quite usual for neural networks, is the performance given by the network for higly distorted images.

The articles ([23], [24], [25], [26]) provide us with a good overview of how fuzzy–techniques can be used all the way from image improvement to the interpretation of the images. They concern automatic recognition of the maturity of the skeleton from an X–ray image of the hand. In the first article a number of fuzzy image improvement techniques are described and implemented. These improved X–ray images are then input for the techniques described in the following articles: edges of bones are detected, the results are fuzzy strings describing the contours of the bones in the hand. Using this description a fuzzy grammar is used to parse these strings for reducing them to a class of maturity of the skeleton. The four sub–problems solved in this way are:

the interpretation of an X–ray image, the detection of specific bones and their location using fuzzy clustering techniques, preprocessing of X–ray images in order to attempt to extract the edges of the different bone– and tissue–structures using fuzzy edge detection techniques (with morphological operations), translation of the detected borders into fuzzy primitives (dot, line, curve, sharp curve, etcetera) using a fuzzy language generated by a fuzzy grammar, and ultimately the syntactical classification of these primitives into one of the different stages of maturity of the skeleton by parsing the fuzzy language to determine the syntactic class of the shape.

1.9 Which technique?

It will be clear by now that none of the techniques described can meet all the demands we had imposed upon them. Fortunately this does not mean that our problem can't be solved, only that the answers must be found in a different solution; If one imaging technique does not provide us with enough information to image the spine accurately, and improving it does not yet give sufficient accuracy, then the *combination* of two or more imaging techniques is a next possible option to examine.

The global idea behind combining different techniques, or more generally speaking, different images, is that this combination may reveal more information than each technique would separately. An example of this phenomenon is binocular vision, where two almost identical two dimensional images can be combined to generate a three dimensional image of the same scene. In our case, the goal of combining images would be to obtain a more accurate and more specific image of the spine. In the next chapter I will research this possibility extensively, using the data that is already available (X–ray images of the spine) as a base for the research.

CHAPTER 2:

Modelling the Spine

Searching amongst the different imaging techniques revealed several promising techniques. These techniques are rastering stereografy and ultrasonografy and, within a longer term, possibly optic tomografy, thermografy or other newly developed imaging techniques. The greatest advantage of these techniques over the X–ray imaging is that they have proven to be completely harmless or at the least much less potentially harmless then X–rays. The main disadvantage of using these techniques is, again compared to using X–rays, most of the time a loss of accuracy. This is a severe disadvantage because one of the main goals of the project is to get rid of this accuracy where and if possible. Fortunately these inaccuracies are not structural but partial inaccurate; they are inherent to the imaging techniques used, and can oftenly very well be predicted. Rastering stereografy for instance can be quite inaccurate in imaging rotated spines. The positions of the dorsal protuberance of the vertebrae [processus spinosus] however can very accurately be computed from a stereografic image. These protuberances give a very good reference for the dorsal position of the spine. Even X–ray images know some inaccuracy, especially the problem of superimposing, i.e. the problem that all the objects in the body appear on top of each other limit its value (CT X–ray imaging solves this problem at the cost of a larger computational load and longer scanning times). The claim we wish to make here is the following: different imaging techniques have different inaccuracies. This suggests a possibility for combining different partially inaccurate techniques, in order to acquire an accurate result. This way of combining the data of two or more different imaging techniques into one image we refer to as *merging*.

When combining two or more different imaging techniques there arise several problems. One has to know which information is accurate enough to include and which information is not. Another important problem to solve is the orientation of the data. When one technique tells a lot about the rotation of the spine and another technique pin–points the dorsal protuberances these ought to be 'translated' into an image of the spine. The first problem can be solved by using foreknowledge on the imaging techniques that are used. This knowledge concerns, simply put, knowing what is accurate and what is not. When this is known the information can be interpreted and combined to a maximal use. This information has been gathered in chapter one. The second problem can also be solved by using foreknowledge, this time the knowledge is knowledge about the shape of the spine. This knowledge can be captured in a model, describing the syntaxis of the spine. This model gives an upper– and lower bound of the possible shapes of the spine. The model must off course be constructed in such a way that it contains enough information for computing an image of the spine, and has to be able to be combined with other models or data in order to provide a well founded base for orientation of different information and images.

2.1 A system for merging different imaging techniques

The system proposal shown in Figure 2 is designed for merging different imaging techniques, or in this case, different models of the spine. The input for the system is the data emerging from the different imaging techniques. In order to interpret this data an interface has to be given, this interface must provide the translation step that is needed to construct a mergable model of the spine from the imaging data. Because in our case this translates 'hard' data into a fuzzy description of the spine this step is called the fuzzification of the data. The next step is the merging of different fuzzy models into a new fuzzy model of the spine, that is better than the the separate models. A fuzzy model can be called better when there exists less uncertainty about the correctness of the model or, to use the correct fuzzy approach is less vague. Because the resulting model is also a fuzzy model another translation is needed, this time the fuzzy data has to be translated into data that is suited for visualisation. This step is called the defuzzification. Finally this translated data is visualised using a viewer that can visualise three dimensional structures constructed from line segments. See also Figure 2.

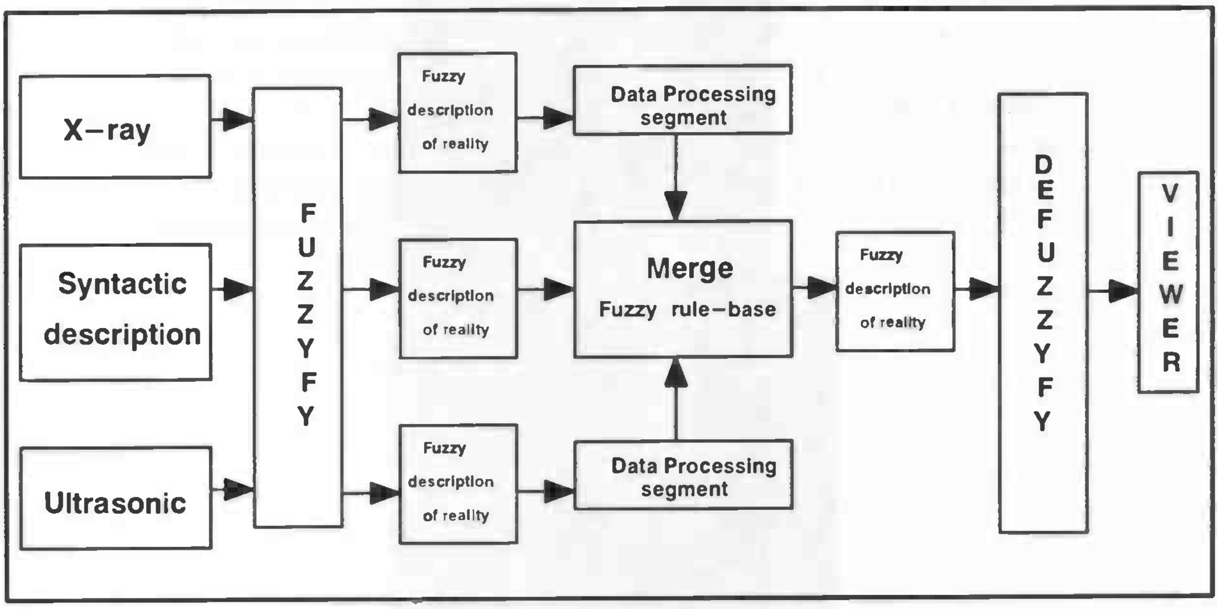


Figure 2: System outline

2.2 Foreknowledge on the spine

Using foreknowledge is of crucial importance when using fuzzy techniques. The success of using fuzzy techniques stands or falls with a good use of this foreknowledge. The advantages and philosophy behind fuzzy logic will be explained in the next chapter. In order to select and interpret data knowledge is needed. For a layman this knowledge can best be captured by simple means: asking, reading, brainstorming or even just guessing. Doing this in combination with an implementation in fuzzy techniques has the mayor advantage that this knowledge can be stated in fuzzy terms, i.e. the terms experts commonly use. When searching this knowledge I focused mainly on what is known of the shape of the spine and the effect of scoliosis on that shape, and the resulting X-ray images, see also ([27], [28] and any medical encyclopaedia e.g. [29]);

- The number of vertebrae in a spine is known. In the classical combination there are seven vertebrae in the neck, the so called cervical vertebrae. Beneath them are the twelve dorsal vertebrae of the back, the thoracic vertebrae. Then there are five lumbar vertebra, five vertebrae that grown together form the sacrum and a variable number (three – five) of vertebrae forming the coccyx.

- The part of the spine that is used by the experts for determination of the shape of the spine with X–rays of patients with scoliosis, runs from T1 to L5 (1th Thoracic vertebra until 5th Lumbar vertebra). This gives a total number of: *12 thoracic vertebrae + 5 lumbar vertebra = 17 vertebrae.* This means that when constructing a model of the spine only this part has to be taken into account.

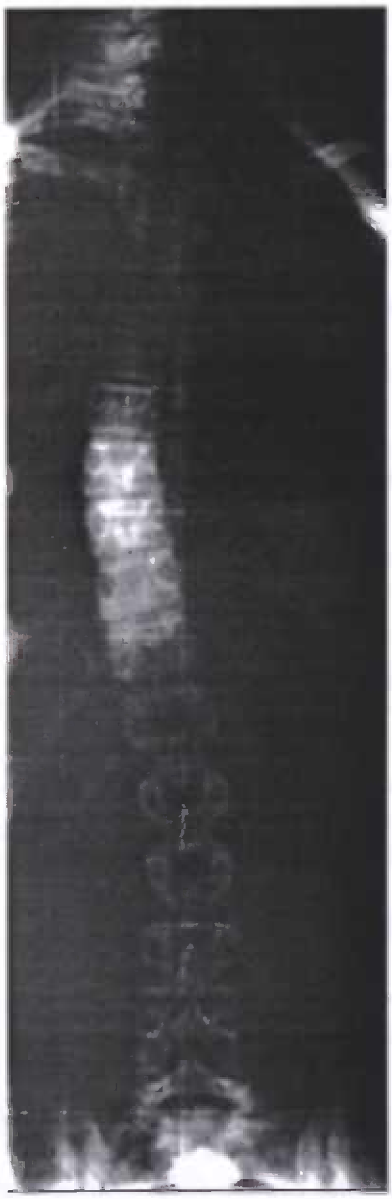


Figure 3: Ventral view of a spine with S shaped scoliosis

- The shape of the separate vertebrae is known. Knowing this shape and how this shape is reflected in the X–rays is important for modelling the vertebrae. When looking at the top view and the back view of the vertebrae their shape is quite capricious. The shapes of vertebrae as they appear in the X–rays that are currently used in imaging the spinal deformations are much more regular. Here a frontal and lateral view of the spine are used, in which the part of the vertebrae that reflects the position of the spine the best, the vertebral bodies, appear as almost rectangular shapes. Expanding this to a three dimensional view one can say that the area of interest is an almost cylindric shaped part of the spine, mostly made up of the vertebral body. There however do exist some serious problems. The first problem is the quality of the X–ray images. In the upper part of the image, the vertebrae are not very well visible because the are partly obscured by the shadows cast by the longs and the organs. This is is especially the case with the lateral views of the spine, where also the ribs cause extra problems. The second problem is the orientation of the different images; in order to be able to merge a ventral view and a lateral view into an accurate 3–D model of the spine there (relative) position in a common world coordinate system has to be known. Currently this is not the case, however when we assume a coordinate system where the y–axis is runs perpendicular to the length axis of the images, the x–axis is the width of the ventral view, and the z–axis the width of the lateral, one can say that: the y–axis in the images are *approximately* the same, and that the rotation angle between the two images when rotating around the y–axis is *approximately* 90^{o}. See also Figure 12.

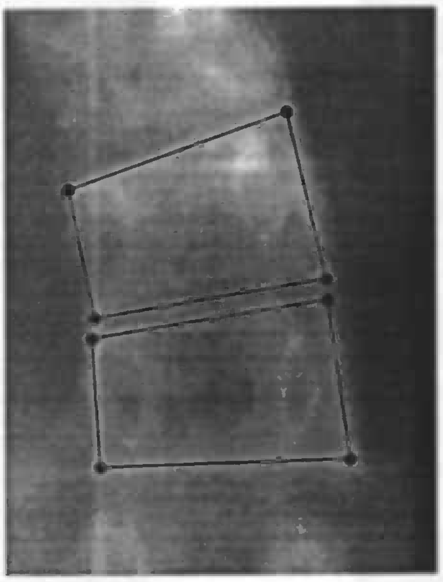


Figure 4: Wedging vertebrae

- The three dimensional relations between successive vertebrae must be known in order to be able to model them. The vertebrae are connected by strong collagen ties called ligaments and intervertebral discs. These discs are slightly transformable causing the spine to be flexible in stead of fixed. This can result in different possible angles, distance and possible even torsion between successive vertebrae. A model for this intervertebral disc should be able to capture this knowledge, and correctly connect the vertebrae. This would result in some kind of 'stacking element', with possible sizes, angles and shapes, these variables are off course limited by anatomic possibilities but these limits can best be interpreted widely because the spinal deformations may very well reach these limits.

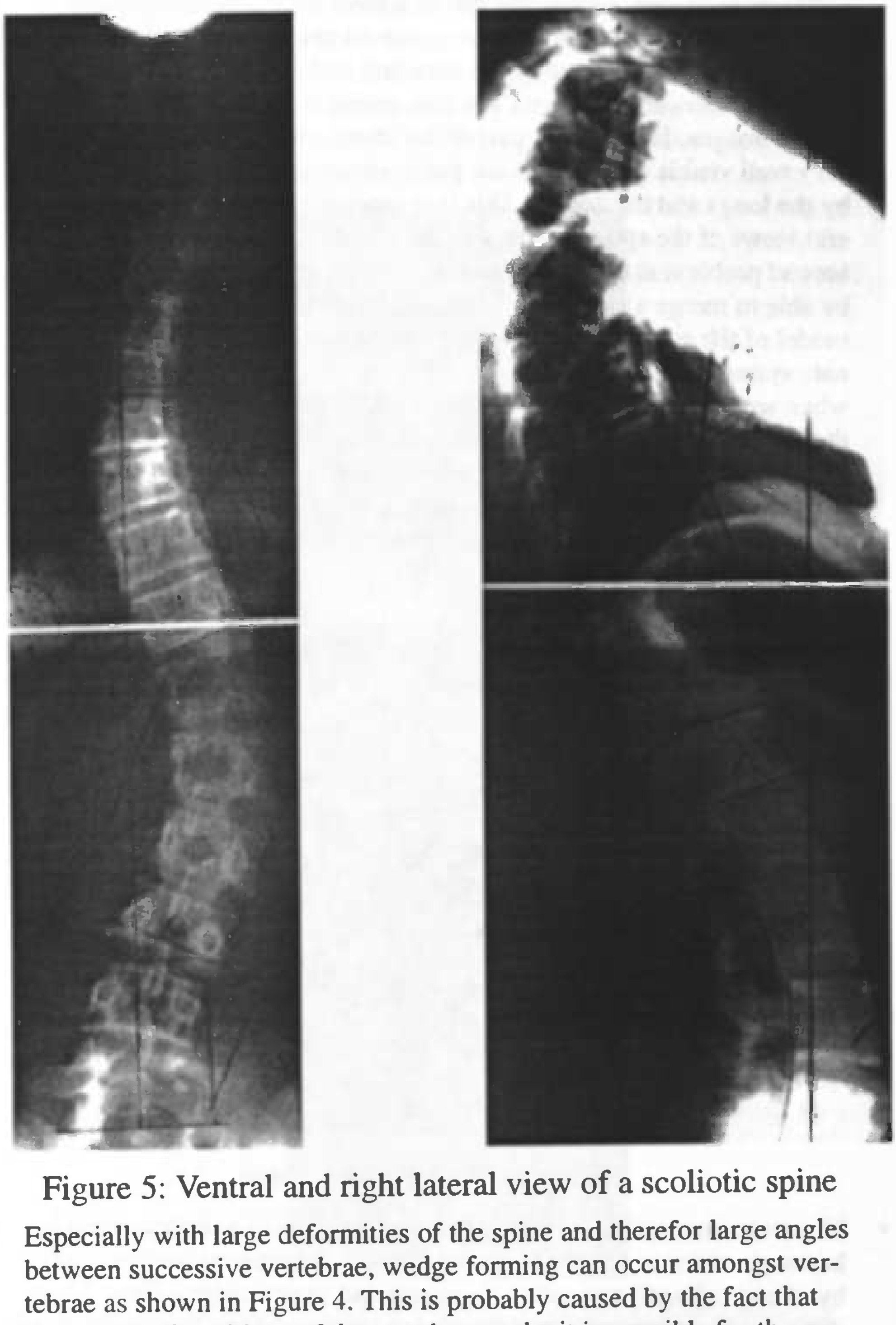


Figure 5: Ventral and right lateral view of a scoliotic spine

- Especially with large deformities of the spine and therefor large angles between successive vertebrae, wedge forming can occur amongst vertebrae as shown in Figure 4. This is probably caused by the fact that the unnatural positions of the vertebrae make it impossible for the vertebrae to grow properly. It is not yet known for certain if this process is irreversible, if it is then correction in the shape of the spine would not lead to correction in the shape of the vertebrae. However it probably is, especially with somewhat older patients, where the growth of the spine has, or has almost, come to an end (20 – 25 years). This means that, when constructing a model for the vertebrae it has also be taken into account that a vertebra also can have a somewhat wedged shape. This can be dealt with in the model of the vertebrae or when connecting the successive vertebrae, fitting the wedge into the connecting element.

- The overall shape of the spine is known. A normal spine is not straight, but has a few gradually merging ventral (frontal) and dorsal bends, with one sharp bend between the lumbal and the sacral area. In the cervical and lumbar area the hollow side of the bend is directed backwards or dorsal; this is called lordosis. In the thoracic and sacral part this side is directed forward or ventral; this is called kyfosis. Because the left and right part of the body are not completely equal the spine can also have one ore two minimal reversable lateral bends in order to maintain balance; this is called functional scoliosis.
- The shape of the spine under the influence of scoliosis also shows some regularities. As stated above a normal spine is almost straight in the ventral or dorsal view. With scoliosis these views show a deviation that is *almost always* shaped like an **S**. When the deviation is **C** shaped the bend is *almost always* to the right (in the ventral view), there is no explanation for this phenomena. The **C** shaped as well as the **S** shaped deviations show the largest lateral deviation *usually* at the height of the seventh thoracic vertebra (T7).
- The X–ray images of patients which suffer from scoliosis show in the lateral views that the lordosis and kyfosis of the spine get smaller; or in normal English, the spine seems straighter. This is usually the caused by the torsion in the vertebrae.

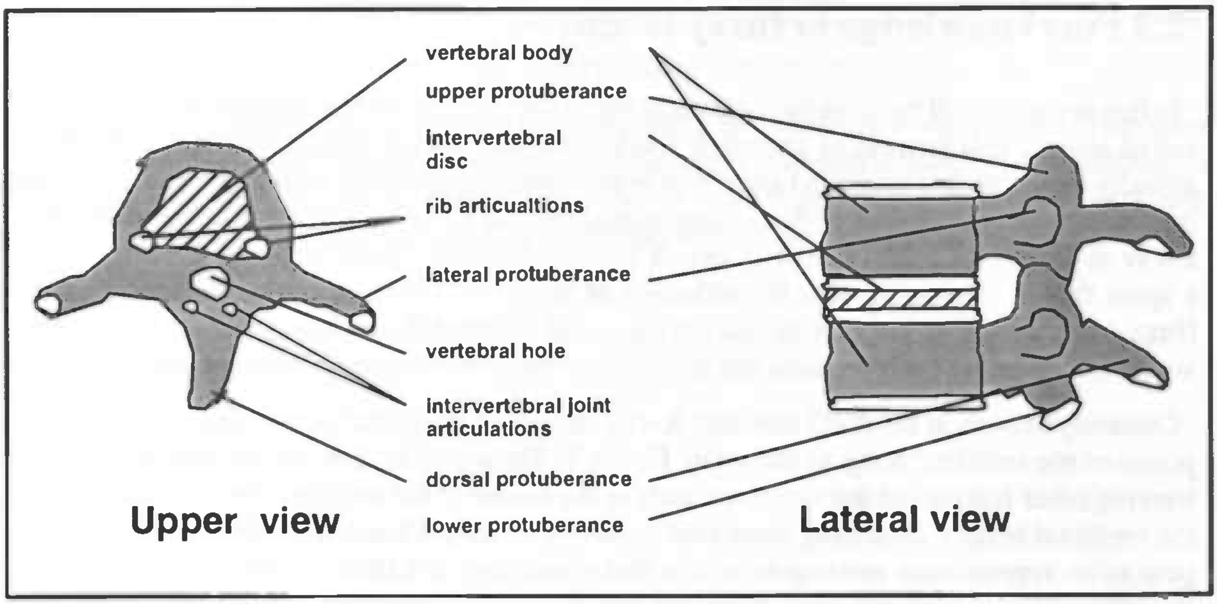


Figure 6: The shape of vertebrae (anatomical)

- When interpreting a ventral view of the spine, experts from the AZG currently mark the four points at the corners of the vertebrae (Upper-

Left, UpperRight, LowerLeft and LowerRight). These points can be used for constructing a model of a vertebra. Because of the possible vagueness in the X–ray pictures that are used to determine this points these points are not necessarily the correct ones.

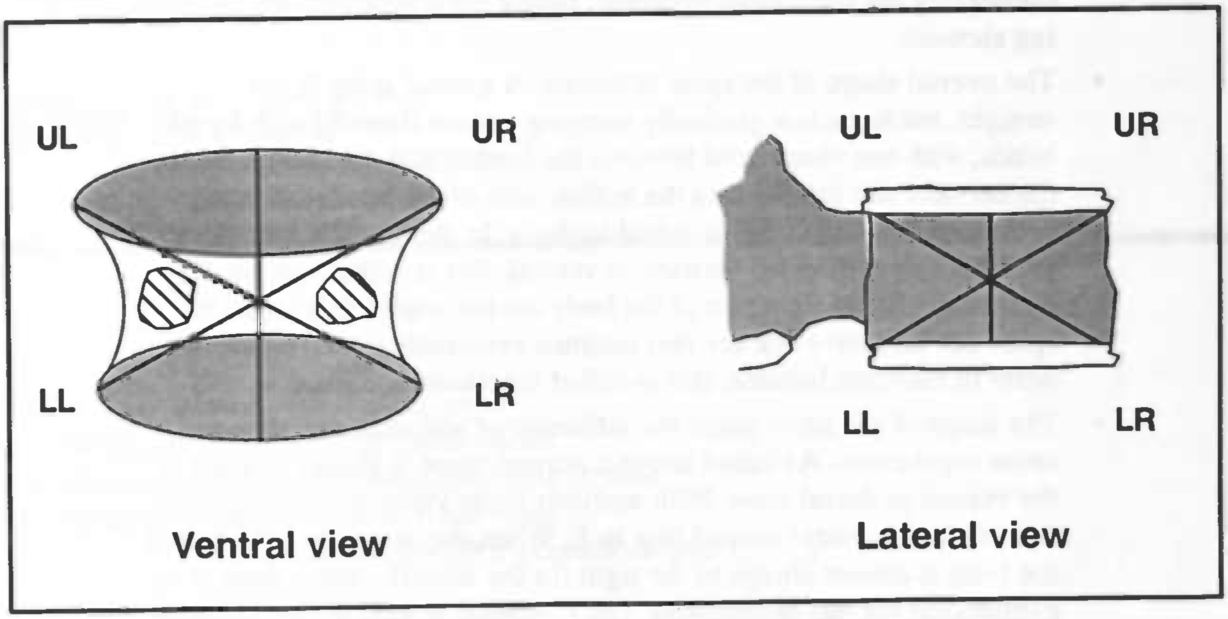


Figure 7: The shape of the vertebrae (schematically)

- When the X–ray scans are made, the spine is forced into a position that already is as straight as possible, using the natural friction in the spine. This is done in order to provide the instrument maker with a good hunch of how much correction can possibly be achieved.

2.3 Foreknowledge in fuzzy terms.

In this section I will try to extract the main fuzzy parts of the foreknowledge that has been collected above. This process of gathering foreknowledge and extracting the important parts has actually been one that consisted mainly of brainstorming, guessing, trying and lots of researching, is focussed on making a (three dimensional) model of the spine that describes the spine in a way an expert (or in absence of an expert a 'smart layman') would describe a normal spine or a spine that is deformed under the influence of scoliosis. This description will be in terms of (fuzzy) three dimensional objects, but not the actual shape of these objects, but the way an expert would interpret and describe them and the relations between these objects are of importance here.

Currently experts at the AZG interpret X–ray images of the spine by marking the four extreme points of the vertebral body as shown in Figure 7. These points then are the base for further retrieving other features of the vertebrae such as the center of the vertebral bodies and the area of the vertebral body. Connecting these four points by a straight line makes the vertebral body appear as an approximate rectangular shape that sometimes is a little wedged. We could say that this interpretation makes the vertebrae look like *'approximate wedged rectangles'*, which would be an usable fuzzy description of the vertebrae as an abstract object.

Marking and recognizing the vertebrae in this way is usually done starting from the upper vertebra that is used to determine the shape of the spine (T1) and then moving downwards repeating this until the last vertebra is reached (L5). The relation between two successive vertebrae can be described as some kind of concatenation, where the connecting element is the intervertebral disc. Because of the fact that these discs are also approximate rectangular shapes but far less deformable than the vertebrae the actual shape of the discs is not important. The fuzzy relation between two successive vertebrae is then simply that the vertebrae are situated *'beneath each other'*, where the bottom of the upper vertebra and the top of the lower vertebra are the referential points.

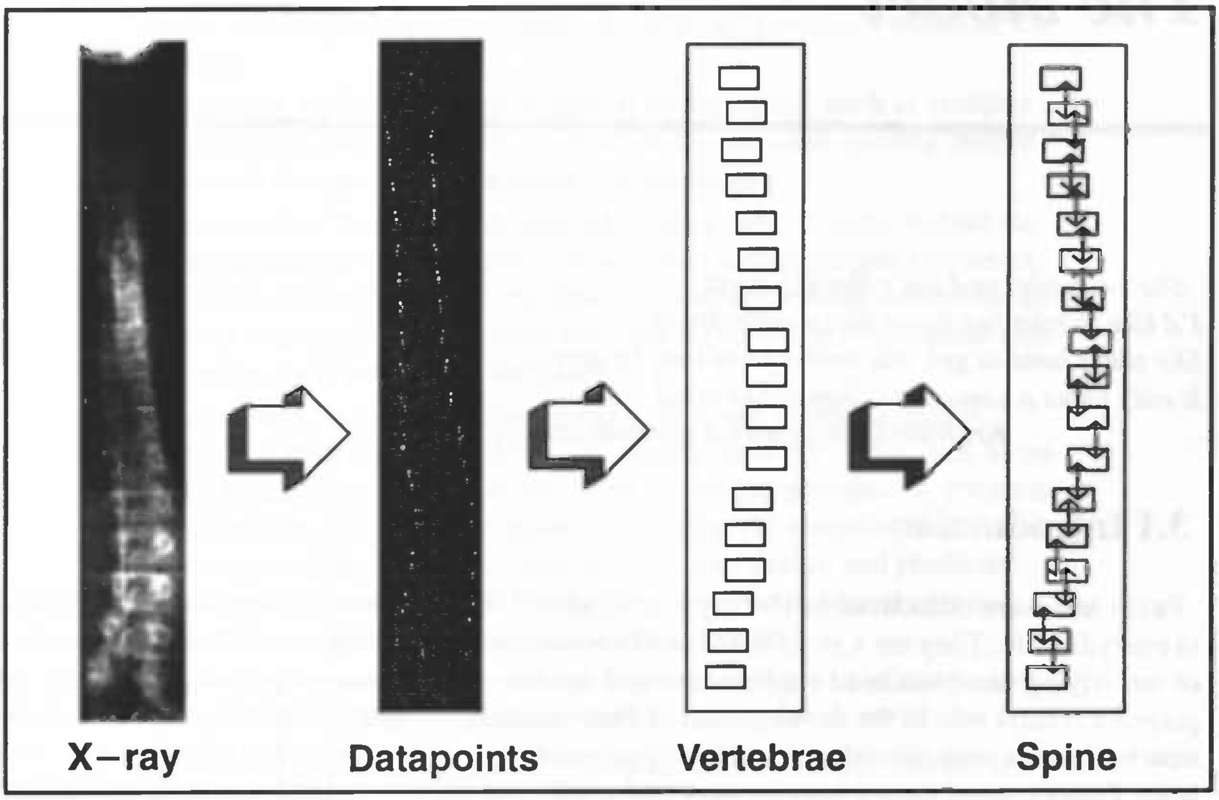


Figure 8: Interpreting the spine

Summarising, a model that can be used for interpreting the data from X–ray images of the spine can be constructed by first modelling the shape of the separate vertebrae using a fuzzy description of this shape and then modelling the relation between these fuzzy shapes by using a fuzzy relation. A technique well fit for this kind of modelling will be presented in the next chapter.

CHAPTER 3:

The Model

She's a model and she's looking good
I'd like to take her home that's understood
She plays hard to get, she smiles from time to time
It only takes a camera to change her mind..
Kraftwerk, sung with a german accent

3.1 Introduction

Fuzzy sets were introduced in 1965 by Lotfi Zadeh ([30]) as a new way to represent vagueness in everyday life. They are a generalisation of conventional set theory, one of the basic structures of underlying computational mathematics and models. Computational pattern recognition has played a central role in the development of fuzzy models because fuzzy interpretations of data structures are a very natural and intuitively plausible way to formulate and solve various problems. Fuzzy control theory has also provided a wide variety of real, fielded system applications of fuzzy technology.

3.1.1 Fuzzy logic and pattern recognition

A very basic definition of pattern recognition is that it is *the search for structure in data.* With this definition it is easy to make a case for the position that pattern recognition is in fact the basis for almost every line of scientific inquiry that humans pursue ([31]). Pattern recognition is an inexact science and thus admits many approaches to the approximate solution of a given problem. Pattern recognition is a current research area because of the need to process data and information obtained from the interactions between scientists, technologists, and society in general. Possibly the most important motivation for study in this field is that scientists and engineers are concerned with the idea of designing and making automata (intelligent machines) that can carry out certain tasks with skills comparable to human performance.

3.1.2 Developing a pattern recognition system (PRS)

When developing a pattern recognition system there are some steps one can take in order to dynamically develop and improve the system.

1. Humans nominate data that hopefully captures basic relationships between the apparently important variables of a process.
2. Data is collected from humans and sensors.
3. We search for underlying structure in the data that provides a basis for hypothesizing relationships between the variables governing the process.
4. Hypotheses are formalized by characterising the process with equations, rules or perhaps algorithms; in short, we propose a model of the system.
5. If possible, various theoretical aspects of the model, such as stability linearity, and composability, are analyzed in hopes of gaining insight into both the model and the process it represents.
6. The model is "trained" with labeled training data; in order to find the right parameters of the model it is provided with examples of correct instances. Classification of the labeled data can be done simultaneously with learning (decision–directed learning) or we may postpone classification until learning is complete
7. The model is tested with labeled test data, when available, and is compared with other models of the same process for things such as relative sensitivity to perturbations of its inputs and parameters, error rate performance, and time and space complexity. (The smaller the training or design set, the better the expected classifier design and predicted validity of its performance. Results are, at the least, biased optimistically if the same training set is used as the test set.)
8. We build, test, and place in service a system comprised of hardware and software that implements the model.
9. The model enables us to classify, predict, estimate and/or control elements of the process and its subprocesses.

3.1.3 Why fuzzy?

Basically the problem to be solved is the determination of the boundary or shape of a class (the class of vertebrae or spines) from its sampled points. Conventional approaches attempt to estimate an exact shape for the area in question by determining a boundary that contains (i.e., passes through) some or all the sample points. It is possible that the sample points have a certain impreciseness (or ambiguity) for instance because of an instrumental error or noise corruption. Other sample points may be measured very precise but appear equally precise to the classifier. It would be convenient to use linguistic variables (e.g., small, medium, very, more, etc.) to describe this feature information. It is also possible that certain regions of the shape to detect are obscured and therefor not represented in the sampled points. In this case the boundaries are extended to fit the model, however the position of the extended boundary is less certain than the ones explicitly highlighted by the data points. This leads one to define multivalued or fuzzy (with continuum grade of belonging) shapes and boundaries of certain classes.

The problem of the determination of the shape of the spine can possibly be modelled as a syntactic pattern recognition problem. A syntactic pattern recognition algorithm ought to be able to decompose and reconstitute objects from representations of structural relationships between various parts of the object, much as humans apparently do. Syntactic pattern recognition deals with representations of structure via sentences, grammar, and automata. Searching among such data is done by means of various kinds of parsing. The syntactic approach has incorporated the concept of fuzzy sets at two levels. First, the pattern primitives are themselves considered to be be labels of fuzzy sets, that is, subpatterns such as "almost circular arcs", "gentle", "fair", and "sharp" curves are considered. Second the structural relations among the subpatterns may be fuzzy, so that the formal grammar is "fuzzified" by weighted production rules, and the grade of membership of a string is obtained by min–max composition of the grades of the production used in the deriviations. Inference of a fuzzy grammar is the problem of inferring the productions as well as the weights of these rules from the specified fuzzy language.

3.1.4 Fuzzy sets and membership functions

A few weeks ago I was at the AZG, where Dirk Jan Wever provided me with a few X–ray scans of scoliotic spines and also answered some questions I had about the common shape of the spine and the vertebrae. "That spine seems *pretty severely* twisted", I said upon seeing the first scan. Dirk Jan smiled and said: "Perhaps to you it is, but we consider it a *minor* deformity". When he explained later that a common used measure for the deformation is the Cobb angle and how it was calculated I understood that for that spine the Cobb angle was *pretty small* indeed. Everyday language is one example of the ways vagueness is used and propagated. Imprecision in data and information gathered from and about our environment is either *statistical* (e.g., the outcome of a coin toss) or *nonstatistical* (e.g., The Cobb angle is pretty small). This latter type of uncertainty is called *fuzziness*.

Children quickly learn how to interpret and implement fuzzy instructions ("go to bed *about* 9"). We all assimilate and use (i.e., act on) fuzzy data, vague rules and imprecise information, just as we are able to make decisions about situations that seem to be governed by an element of change. Accordingly, computational models of real systems should also be able to recognize, represent, manipulate, interpret and use both fuzzy and statistical uncertainties. Statistical models deal with random events and outcomes; fuzzy models attempt to capture and quantify nonrandom imprecision.

3.2 Fuzzy logic and syntactic pattern recognition

The idea behind syntactic pattern recognition is that certain pattern classes contain objects, such as geometric figures, that have an identifiable hierarchical structure that can be described by a formal grammar, called the *pattern grammar*. A basic set of pattern primitives is selected and forms the set of terminals of the grammar. The productions of the grammar are a list of allowable relations among the primitives. The pattern class is the set of strings generated by the pattern grammar. The productions of the grammar are a list of allowable relations among the primitives. The pattern class is the set of strings generated by the pattern grammar. However, the concept of a formal grammar is often too rigid to be used for representation of real patterns, which are generally distorted and noisy, but yet still retain much underlying structure.

This is where fuzzy languages come in. Fuzzy languages can handle imprecise patterns when the indeterminancy is due to inherent vagueness. The fuzziness may lie in the definition of primitives or in the physical relations among them. Thus, the primitives become labels of fuzzy sets and the production rules of the grammar are weighted. The membership grade of a particular pattern in the class described by the grammar is calculated using min–max composition, i.e. the grammar is fuzzy.

3.2.1 Fuzzy tree automata

The fuzzy language we use in our case can be very simple. Because the strings we want to recognize are simply concatenations of vertebrae, together forming a partial spine, it is possible to use trees for representing the patterns and then process these trees by using fuzzy tree automata. These trees then represent the fuzzy model we wish to implement, where the leaves represent the different objects we want to recognize, and the internal vertices represent the fuzzy relation between the objects. This method, see ([32]), gives a well structured approach to recognizing patterns that consist of several, well seperatable objects, which are related by fuzzy relations.

3.2.2 Defining fuzzy root–to frontier tree automata (FRFTA)

Definition 1: A fuzzy root–to–frontier tree automaton (FRFTA) over an alphabet Σ is a quintuple $(K, \Sigma, \delta, q_0, F)$, where K is a finite nonempty set of states, Σ is a finite input alphabet, $q_0 \in K$ is the initial state, and $F \subset K$ is a set of final states which may be a fuzzy set over K. The symbol δ denotes a fuzzy mapping from $K \times \Sigma$ to $K \times K$. This means that each pair (q_j, a) in $K \times \Sigma$ defines a pair of fuzzy "next states" in $K \times K$ which is characterized by the conditional membership functions $\mu_1(q_j \mid q_i, a)$ and $\mu_2(q_j \mid q_i, a)$ with arguments $q_j \in K$, $q_i \in K$ and $a \in \Sigma$.

The formation of the state tree can be described inductively as follows:

1. The root of the state tree is labeled q_o.
2. Given that any node of the state tree is labeled Q which in general is a fuzzy set in K defined by a membership function $\mu(q_i)$ and the corresponding node of the input is labeled a, then the two successor nodes of the state tree are labeled with the pair $\{Q_1, Q_2\}$ whose membership function is given by:

$$\mu_1(q_j) = \bigcup_{q_i} (\mu(q_i) \wedge \mu_1(q_j \mid (q_i, a)) \qquad \text{(i)}$$

$$\mu_2(q_j) = \bigcup_{q_i} (\mu(q_i) \wedge \mu_2(q_j \mid (q_i, a)) \qquad \text{(ii)}$$

Where $\cup_{q_i}$ denotes the supremum over $q_i \in K$. When Q is a singleton $\{q_i\}$, the membership functions for next state reduce to $\mu_1(q_j \mid q_i, a)$ and $\mu_2(q_j \mid q_i, a)$.

Definition 2: Let $Q_1, Q_2, ..., Q_m$ denote the fuzzy sets of states labeling the frontier nodes of a Σ–tree, t. Let F be a designated set of final states, which may be a fuzzy subset of K. Then $\mu_A(t)$, the *grade of acceptance* of t by the FRFTA A, is given by the minimal grade in the set of maximal grade in $F \cap Q_1, F \cap Q_2, ..., F \cap Q_m$, the intersections of F and Q_1, F and Q_2, ..., F and Q_m.

Definition 3: Let $A = (K, \Sigma, \delta. q_o, F)$ and $A' = (K', \Sigma, \delta', q_o', F')$ be two fuzzy root–to–frontier tree automata. The direct product $A \times A' = (K \times K', \Sigma, \delta \times \delta'. (q_o \times q_o'), F \times F')$, where $K \times K'$

and $F \times F'$ are the Cartesian product of sets, $(q_0 \times q_0')$ is the ordered pair of q_0 and q_0', and fuzzy mapping $\delta \times \delta'$, which is characterized by a conditional membership function, is defined by the formulas:

$$\mu_1((q_j,\ q_j') \mid (q_i,\ q_i'),\ a) = (\mu_1(q_j \mid (q_{i,} a),\ (\mu'_1(q_j \mid (q_{i,} a)) \qquad \text{(iii)}$$

$$\mu_2((q_j,\ q_j') \mid (q_i,\ q_i'),\ a) = (\mu_2(q_j \mid (q_{i,} a),\ (\mu'_2(q_j \mid (q_{i,} a)) \qquad \text{(iv)}$$

for all $q_j \in K, q_i \in K, q_j' \in K, q_i' \in K$ and $a \in K$.

The fuzzy set of Σ–trees accepted by an FRFTA A is denoted $T(A)$, and a fuzzy set U of Σ–trees is *recognizable* if $U = T(A)$ for some fuzzy root–to–frontier automaton A.

Theorem 1: If A and A' are fuzzy root–to–frontier automata, then $T(A \times A') = T(A) \cap T(A')$, that is $\mu_{A \times A'}(t) = min\ (\mu_A(t), \mu_{A'}(t))$, with t in T_Σ.

Corollary 1: The class of fuzzy root–to–frontier recognizable Σ–trees is closed under intersection.

Theorem 2: The class of fuzzy root–to–frontier recognizable Σ–trees is closed under union.

Theorem 3: The class of fuzzy root–to–frontier recognizable Σ–trees is closed under complementation.

Theorem 4: The class of fuzzy root–to–frontier recognizable Σ–trees forms a Boolean algebra.

A fuzzy tree automaton is just a fuzzy automaton in which labeled trees replace strings as inputs. Because in our case binary input trees are sufficient for modelling the spine, the definition of the automata we will use can be a simplified version, that can parse binary fuzzy trees. Taking the informal definition of a generalized fuzzy finite automaton operating on fuzzy binary trees leads to the following definition of a generalized fuzzy finite automaton operating on fuzzy binary trees:

A fuzzy root–to–frontier tree automata (FRTFTA) consists of a finite set S of states; a fuzzy transition function

$$M : \Sigma \times S \rightarrow S \times S \qquad \text{(v)}$$

With fuzzy transition membership functions m_L and m_R.

$$M[(a,\ f_a),\ S_i] = \left\langle \left(S_L,\ f_a \wedge m_{LS,a}\right),\ \left(S_R,\ f_a \wedge m_{RS,a}\right)\right\rangle \qquad \text{(vi)}$$

with a in Σ, and S_i, S_L, S_R in S. The initial state is denoted by S_0 and a set of final states is denoted by F, where F is a subset of S.

The formation of the fuzzy state tree is described inductively as follows:

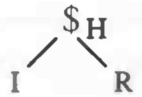
1. The root of the fuzzy state tree is labeled S_0.
2. Given that any node of the fuzzy state tree is labeled $(S_i,\ m_i)$ and the corresponding node of the input is labeled (a, f_a), then the two successor nodes of the fuzzy state tree are labeled with the pair:

$$M\left[(a,\ f_a),\ \left(S_i,\ m_i\right)\right] = \left\langle\left(S_L,\ f_a \wedge m_i \wedge m_{LS_i a}\right),\ \left(S_R,\ f_a \wedge m_i \wedge m_{RS_i a}\right)\right\rangle \qquad \text{(vii)}$$

The fuzzy binary input tree is accepted if every state labeling the frontier of the fuzzy state tree is a final state. The grade of acceptance of the fuzzy binary input tree is equal to the minimum of the memberships of all the states labeling the frontier of the fuzzy state tree.

3.3 Fuzzy tree automata and syntactic pattern recognition

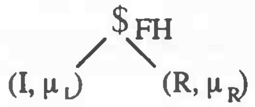
The application of these automata can best be illustrated by a simple example. Consider the simplified version of a "house" to be an isosceles triangle vertically connected to a rectangle. Thus, a house may be represented by a tree as:



(viii)

Where *I* represents an isosceles triangle, *R* represents a rectangle, and the syntactic relation between *I* and *R* is vertical concatenation.

When using the concept of a fuzzy–language an "approximate house" may be defined as an "approximate isosceles triangle" vertically concatenated to an "approximate rectangle". Therefor the tree representation of a "fuzzy house" is denoted by:

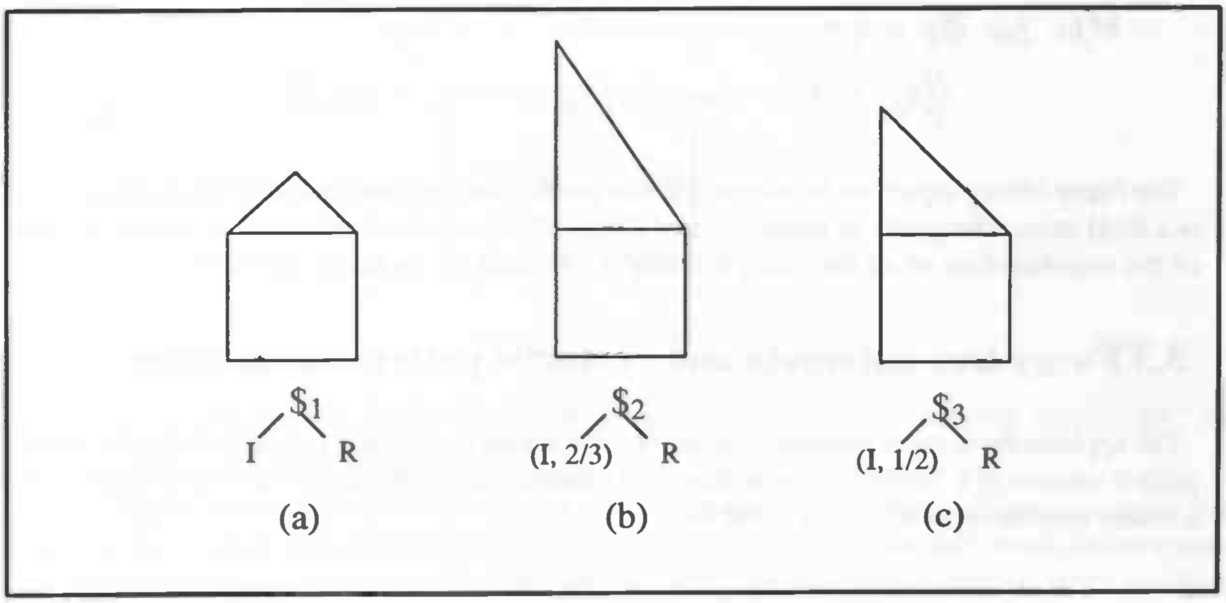


(ix)

Where μ_I and μ_R are grades of membership of "approximate isosceles triangle" and "approximate rectangle" respectively.

For a triangle ΔABC with angles *B* and *C* as the base angles, a quantitative measure of the similarity of this triangle to isosceles triangles may be defined as:

$$\mu_I(\Delta ABC) = 1 - \frac{|B - C|}{90^o} \qquad \text{(x)}$$



(a) (b) (c)

Figure 9: ”Approximate houses”

Example 1: For $A = 30^0$, $B = 90^0$, and $C = 60^0$ as shown in Figure 9(b)

$$\mu_I(\Delta ABC) = \frac{2}{3}$$

For $A' = 45^0$, $B' = 90^0$, and $C' = 45^0$ as shown in Figure 9(c)

$$\mu_I(\Delta A'B'C') = \frac{1}{2}$$

A quantitative measure of the similarity of a quadrangle with angles *A, B, C* and *D.* to rectangles can for instance be defined as:

$$\mu_R = 1 - \frac{|A - 90^o| + |B - 90^o| + |C - 90^o| + |D - 90^o|}{360^o} \qquad \text{(xi)}$$

Example 2: Generating a fuzzy root–to–frontier tree automaton which has the capability of processing the fuzzy representation of ”fuzzy houses” denoted as $TR(\$_{FH})$ as shown in (ix). The generated fuzzy tree automaton is constructed as follows. The set of states is

$$S = \{S_0,\ S_1,\ S_2,\ S_3,\ S_4,\ S_5,\ S_6\}$$

The fuzzy mapping $M\colon \Sigma \times S \to S \times S$ is defined as follows:

$$M[\$_{FH},\ S_0] = \langle S_1,\ S_2 \rangle$$

$$M[(I,\ \mu_I),\ S_1] = \langle (S_3,\ \mu_I),\ (S_4,\ \mu_I) \rangle$$

$$M[(R,\ \mu_R),\ S_2] = \langle (S_5,\ \mu_R),\ (S_6,\ \mu_R) \rangle$$

The set of final states is

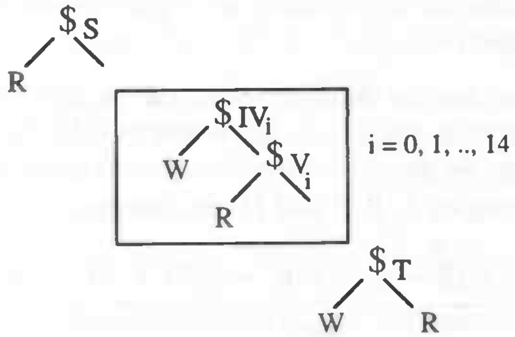
$$F = \{S_3,\ S_4,\ S_5,\ S_6\}$$

Example 3: The grades of membership of "approximate houses" for the three figures shown in Fig. 6 are

$$\mu_H(\$_1) = 1; \qquad \mu_H(\$_2) = \frac{2}{3}; \qquad \mu_H(\$_2) = \frac{1}{2}$$

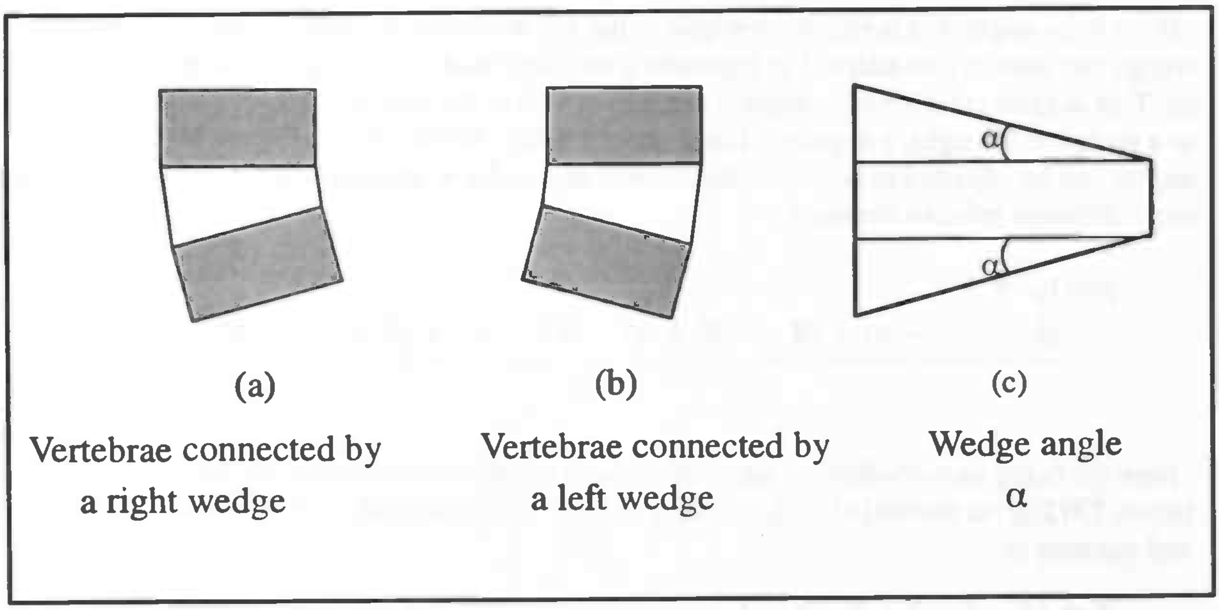
3.4 A fuzzy root–to–frontier tree automaton for the spine

This first base model is going to be a two dimensional description of a spine, where the spine is constructed from seventeen rectangles vertically connected by sixteen four–angled wedging rectangles. The representation of a spine by a tree then becomes the following:



(xii)

Where *R* represents a rectangle, *W* represents a wedged rectangle, and again the syntactic relation between *R* and *W* is vertical concatenation. *IV, V* and *T* are used to represent an intervertebral disk (the wedged four–angle), a vertebra (the rectangle), and the tail section of spine (an intervertebral disk vertically connected to a vertebra), respectively. In this model the connection between spines has been modelled by an intervertebral disc, but also fuzzy relations can be used for defining for instance that the vertebrae are 'above' each other ([33]).



(a)

Vertebrae connected by a right wedge

(b)

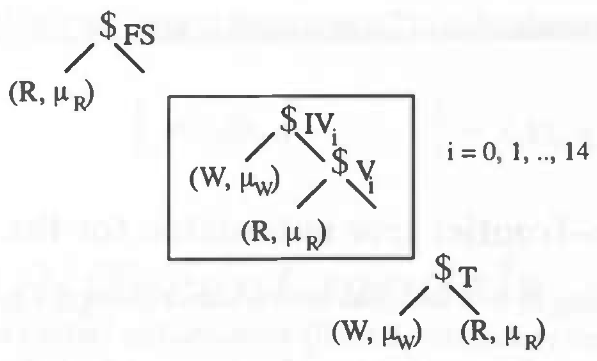
Vertebrae connected by a left wedge

(c)

Wedge angle α

Figure 10: Wedging four–angles connecting the vertebrae

The fuzzy "approximate spine" may now be defined as seventeen "approximate rectangles" vertically connected to each other by sixteen "approximate wedging four–angles". The tree representation of a "fuzzy spine" is denoted by:



(xiii)

Where μ_W and μ_R are grades of membership of "approximate wedging four–angle" and "approximate rectangle" respectively.

The "approximate rectangles" can use the relative measure of (xi), the wedging rectangles need another description. The difference with a rectangle is that a wedge has only two sides symmetry, so two pairs of adjacent corners are approximately the same. A quantative measure for similarity to a four angled wedge with angles *A, B, C* and *D* can then possibly be defined as:

$$\mu_{WR} = 1 - \frac{|A - 85^o| + |B - 95^o| + |C - 95^o| + |D - 85^o|}{360^o} \qquad \text{(xiv)}$$

for a wedge slightly to the right, as shown in Figure 10 or:

$$\mu_{WL} = 1 - \frac{|A - 95^o| + |B - 85^o| + |C - 85^o| + |D - 95^o|}{360^o} \qquad \text{(xv)}$$

For a four–angle that is slightly wedged to the left as shown in Figure 10(b). The angle of the wedge can possibly be adapted to represent a standard model of the spine, the spine as it should be. This suggests that a wedge angle α can also be part of the wedge, a positive angle α then leads to a wedge to the right, a negative α represents a wedge to the left, see Figure 10(c). The wedge angles can be adjusted to represent the correct angles for a 'standard spine', without having to use a different relative measure:

$$\mu(a)_W = 1 - \frac{|A - 90^o - a| + |B - 90^o + a| + |C - 90^o + a| + |D - 90^o - a|}{360^o} \qquad \text{(xvi)}$$

Now the fuzzy root–to–frontier automaton that is capable of processing the fuzzy tree representation, *TR($\$_{FH}$)* as shown in (xiii), of "fuzzy spines" can be defined. The set of states is quite large and consists of:

$$S = \{S_0,\ S_1,\ S_2,\ S_3,\ S_4,\ S_5,\ S_6,\ S_7,\ \dots\dots\dots, S_{127},\ S_{128},\ S_{129},\ S_{130}\}$$

The fuzzy mapping $M\colon \Sigma \times S \rightarrow S \times S$ is defined as follows:

$$M[\$_{FH},\ S_0] = \langle S_1,\ S_2\rangle$$

The $M[(R,\ \mu_R),\ S_1] = \langle (S_3,\ \mu_R),\ (S_4,\ \mu_R)\rangle$

$$\left.\begin{aligned} M\left[\$_{IV_i},\ S_{(i\times 8+2)}\right] &= \left\langle S_{(i\times 8+5)},\ S_{(i\times 8+6)}\right\rangle \\ M\left[(W,\ \mu_W),\ S_{(i\times 8+5)}\right] &= \left\langle \left(S_{(i\times 8+7)},\ \mu_W\right),\ \left(S_{(i\times 8+8)},\ \mu_W\right)\right\rangle \\ M\left[\$_{V_i},\ S_{(i\times 8+6)}\right] &= \left\langle S_{(i\times 8+9)},\ S_{(i\times 8+10)}\right\rangle \\ M\left[(R,\ \mu_R),\ S_{(i\times 8+9)}\right] &= \left\langle \left(S_{(i\times 8+11)},\ \mu_R\right),\ \left(S_{(i\times 8+12)},\ \mu_R\right)\right\rangle \end{aligned}\right\}\ i = 0,\ 1, .., 14$$

$$M[\$_T,\ S_{122}] = \langle S_{125},\ S_{126}\rangle$$

$$M[(W,\ \mu_W),\ S_{125}] = \langle (S_{127},\ \mu_W),\ (S_{128},\ \mu_W)\rangle$$

$$M[(R,\ \mu_R),\ S_{126}] = \langle (S_{129},\ \mu_R),\ (S_{130},\ \mu_R)\rangle$$

The set of final states is

$$F = \{S_3,\ S_4,\ S_7,\ S_8,\ S_{11},\ S_{12},\ \ldots,\ S_{122},\ S_{123},\ S_{127},\ S_{128},\ S_{129},\ S_{130}\}$$

The final state which we are interested in the most is off course the last one, S_{130} where all the vertebrae are recognized, so the corresponding membership μ_R is the membership for the "approximate spine".

CHAPTER 4:

Merging different models

In this section I want to explain some of the choices that were made when making the program that provides the functionality as schematically shown in Figure 11. This figure is an adapted version of the system proposal, that shows how much work has been done. Because a program must have a name that can be pronounced properly, has a nice ring to it, and also gives some information about what it actually does, I called the program MoBI, short for Model Builder/ Interpreter.

4.1 A partial implementation of the System Proposal; MoBI

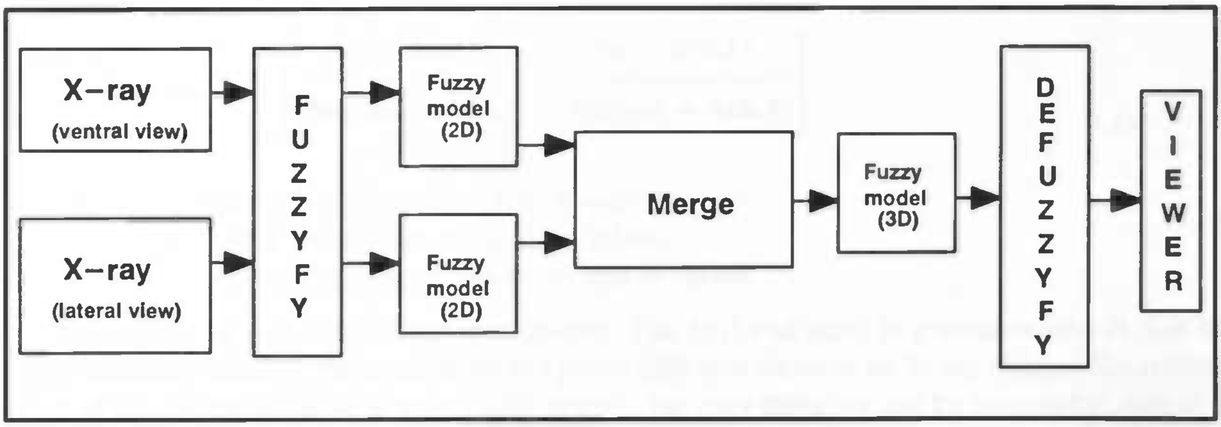


Figure 11: The program 'MoBI'

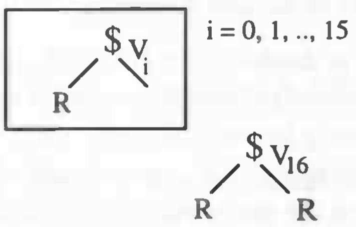
Though this system is quite complete when comparing it to the proposal of Figure 2, there unfortunately there is still a lot of work to be done. I will come to the further possibilities for the program that haven't yet been implemented later, but first I will describe the work that has been done by explaining the functionality of the program in a birds–eye view;

- The input for the system is data that is loaded from a file and internally stored in a matrix. This file contains the locations of the marked vertebral bodies in the X–rays of the spines, ordered along the length axis of the image (which is assumed to be the length axis of the spine).
- The internally stored data is used to build a binary input tree, the base for the fuzzy models of the vertebrae and the spine, for each view that is loaded from file.

- The vertebrae and the spine can be assigned with fuzzy membership values, stating their resemblance to 'approximate vertebrae' and 'approximate spines', respectively.
- Different two dimensional views (made from different angles) can be merged to a three dimensional view of the spine. For now this implementation is done quite straightforward, using crisp normalised coordinate values and foreknowledge on the reliability of the different views. This results in a new binary fuzzy tree model of the spine.
- The two dimensional 'approximate vertebrae' and 'approximate spine' notions have been extended the respective three dimensional models, corresponding to the new three dimensional model of the spine. Again the vertebrae and spine can be given fuzzy membership values stating their memberships to this new fuzzy features.
- The different models can be stored in a file, formatted in such a way that the viewer is able to image the approximate rectangles (semi three dimensional view of a two dimensional model) and approximate cubes (real three dimensional view) that model the approximate spines. Furthermore I have adapted the viewer slightly, so it is able of using colors, where the different colors represent different membership values.

4.1.1 Building the model

The input the program expects is the data that is collected from the X–ray images of the spine; seventeen groups of four two–tupels, describing the four edges of the vertebral body in a ventral or a lateral view. These datapoints are read from a file and internally stored into a datastructure. Using this datastructure the program builds a simplified version of the binary tree, which can be stored in a linked list (see formula (xvii)). At every leaf R the the points **UL**, **UR**, **LL** and **LR** are stored, labeled with their respective names. This results in the binary tree representation as shown in formula (xvii), which is indeed forms fuzzy binary root to frontier input tree.



(xvii)

With this binary input tree we can now reconsider the calculation of the grade of belonging of the spine that is modelled by the tree to the fuzzy feature of 'approximate spines'. This grade must be high (near one, implying little fuzzyness about the belonging of the spine to this class) when the spine has a normal shape, and low (near zero, again implying little fuzzyness, but this time about the spine *not* belonging to the class) when the shape differs a lot from a shape that is defined as a normal shape of the spine. The values in between represent the different fuzzy gradations of belonging to the class of 'approximate spines', i.e. when the shape looks *somewhat* like a spine, a membership of approximately 0.4 would be appropriate.

Because we modelled vertebrae like approximate rectangles, and the spine is a concatenation of these vertebrae we can say that a shape looks a lot like a spine when:

- The approximate vertebrae are stacked in the right way. This is guaranteed by the ordering of the datapoints along the length axis of the spine (the y–axis), so we need not worry about that.
- The vertebrae are not deformed, that is, using the current model of approximate vertebrae, they look a lot like rectangles.

Thus taking the the minimum over all membership values of the fuzzy approximate vertebrae constructing the fuzzy approximate spine model gives a good measure for the resemblance of the model to a normal spine.

$$\mu_{V_i} = \min\left(\mu_R, \mu_{V_{i+1}}\right), \; i = 0, 1, .. 15 \qquad \text{(xviii)}$$

$$\mu_{V_{16}} = \min\left(\mu_{R(1)}, \mu_{R(2)}\right) \qquad \text{(xix)}$$

This is exactly the definition of the grade of acceptance of the fuzzy binary input tree that represents the model of the spine as stated in Chapter 3. So the intuitive definition matches the mathematical one; therefor we may conclude that 'the human way' of modelling and interpreting X–ray images of the spine has been modelled correctly!

Normalisation of the data

Before the datapoints are stored in the list they are normalised using formula (xx). This forces all the datapoints to lie in the interval that is limited by the points (0,0) and (1,1).

$$(x, y)_{norm} = \left[\frac{(x - MinX)}{(Maxval - MinX)}, \frac{(y - MinY)}{(Maxval - MinY)}\right] \qquad \text{(xx)}$$

with: *MinX* the minimum of all x–values,
MinY the minimum of all y–values,
Maxval the maximum of x– and y–values.

The purpose of this normalisation is double. The first and most important reason is that the normalisation makes it much easier to merge the different views of an X–ray image. The orientation of the different views is not exactly known, the only thing we can be reasonable sure of is that the y–axis (the length axis of the spine) has approximate the same direction. Furthermore we know that the different views show the same spine, and thus the distance between the highest point and the lowest point must be approximately the same. This normalisation aligns two different views using this features. The second reason is simply that the viewer that I use assumes that its input is normalised in the same manner.

4.1.2 Constructing a 3D–view from two 2D views

Now that we have two 2D views of the spine, or when they are visualised using the viewer, two semi–3D views, the next step is the merging of these two views into a new, more accurate one, in this case a real 3D view. This would model the vertebrae as '*approximate cubic shapes*', the logical extrapolation of the notion of dimensional '*approximate rectangular*' vertebrae. When constructing a fuzzy definition of an 'approximate cube' I used the the fact that the 'approximate cube' is constructed from two 'approximate rectangles' namely the one constructed from the data from the ventral and the one constructed from the data emerging from the lateral view. The less these two look like a rectangles, the less the resulting 3D vertebra will look like a cube. Thus the definition of the fuzzy 'approximate cube' can very simply be stated as:

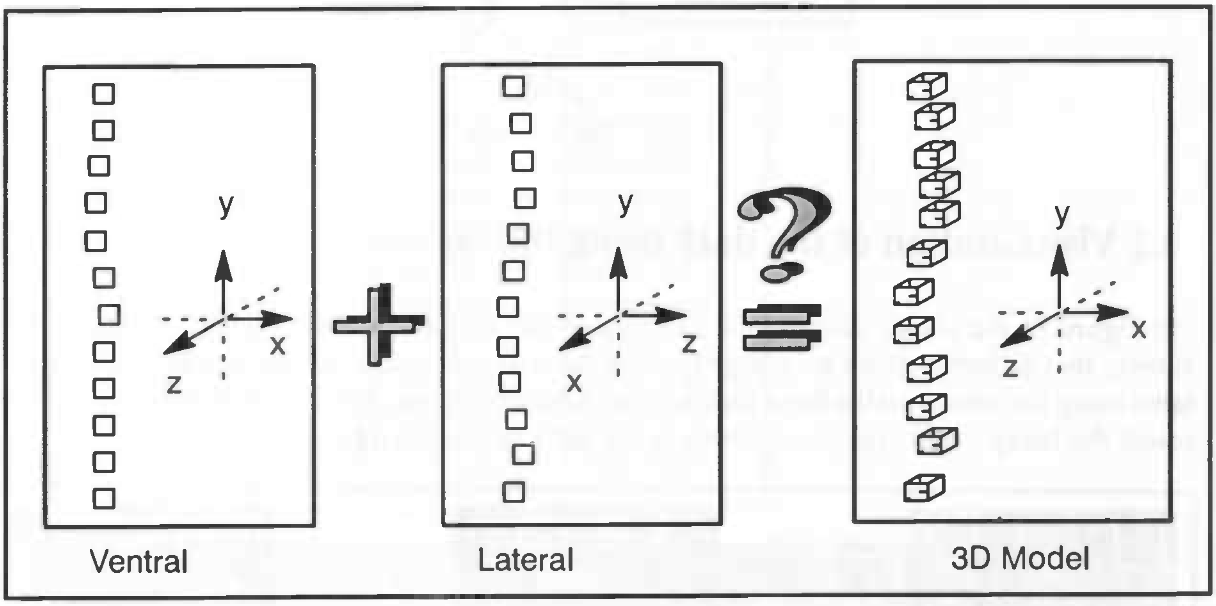


Figure 12: Merging two 2D views to a 3D model

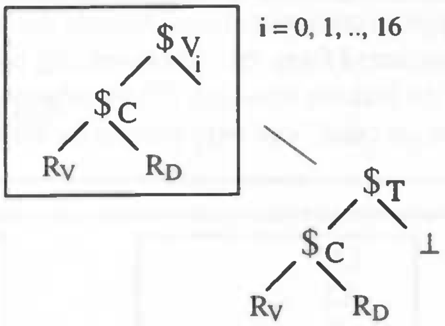
$$\mu_C = \min\left(\mu_R(ventral),\ \mu_R(lateral)\right) \qquad \text{(xxi)}$$

with: μ_R is the grade of similarity to a rectangle as stated in formula (xi).

Thus, when using the fuzzy root to frontier input tree as a model, merging two views into an 'approximate spine' model constructed from 'approximate cubes' can be reduced to simply calculating the union of the two view models, using the definition of chapter 3, that is, taking the minimum of each element, see formula (xxii).

$$\mu_{Merge}(t) = \min\left(\mu_{Ventral}(t),\ \mu_{Lateral}(t)\right),\ with\ t \in \left(T_{\Sigma(lateral)} \cap T_{\Sigma(lateral)}\right) \qquad \text{(xxii)}$$

The three dimensional model is stored as a dorsal and a ventral quadrangle, the ventral view is almost completely the same as the original ventral view, only the z–coordinate values are filled in, using the ones extracted from the lateral view. The ventral view is used as the reference model because here the vertebral bodies are easier to indicate than with the lateral view, where the longs and the organs obscure much of the vertebrae. The resulting tree structure is shown in formula (xxiii).



(xxiii)

4.2 Visualisation of the data using the viewer

In Figure 13 the visualisation of the merging of two two–dimensional models of the spine is shown; two different views are merged in to a three dimensional one, that has been slightly rotated using the viewer so the three dimensional structure shows. The colors of the vertebrae represent the fuzzy memberships as shown in the table below the figure.

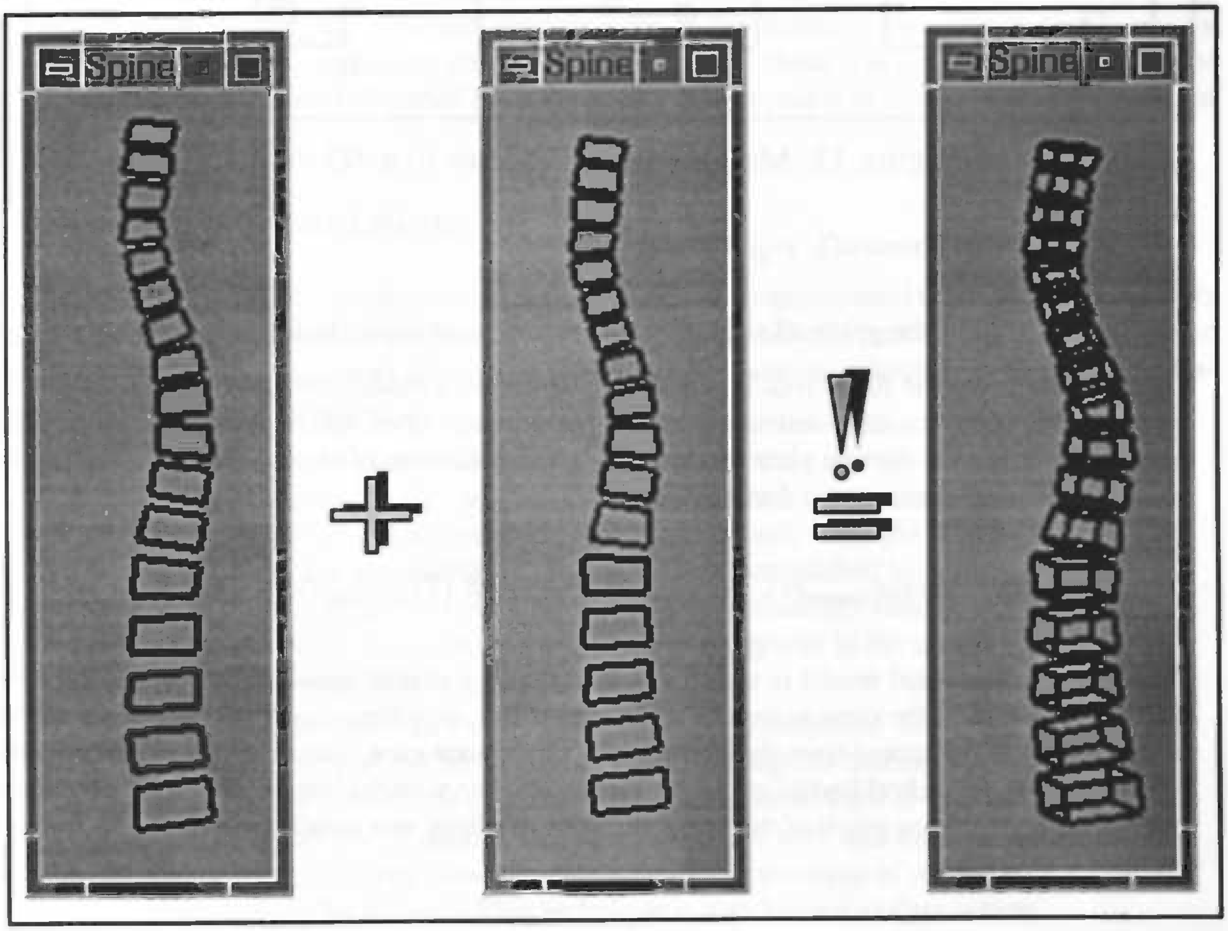


Figure 13: Visualisation of the merge

For the documentation on the viewer I want to refer to the thesis report of Hans Seinhorst and Erik Grave, who initially implemented this viewer for their own visualisation purposes and use an extended version as an integrated part of a system for fuzzy simulation and composition of human motion.

	Ventral	Lateral	Merge
T1	0.980682	0.931633	0.931633
T2	0.950433	0.970246	0.950433
T3	0.920403	0.933695	0.920403
T4	0.931408	0.951858	0.931408
T5	0.939093	0.940739	0.939093
T6	0.877827	0.905730	0.877827
T7	0.963002	0.959037	0.959037
T8	0.915242	0.931569	0.915242
T9	0.918118	0.931695	0.918118
T10	0.962446	0.938522	0.931695
T11	0.909753	0.942281	0.909753
T12	0.923817	0.916604	0.916604
L1	0.953840	0.913071	0.913071
L2	0.900113	0.865910	0.900113
L3	0.949937	0.891083	0.891083
L4	0.926446	0.945312	0.926446
L5	0.884294	0.808017	0.884294

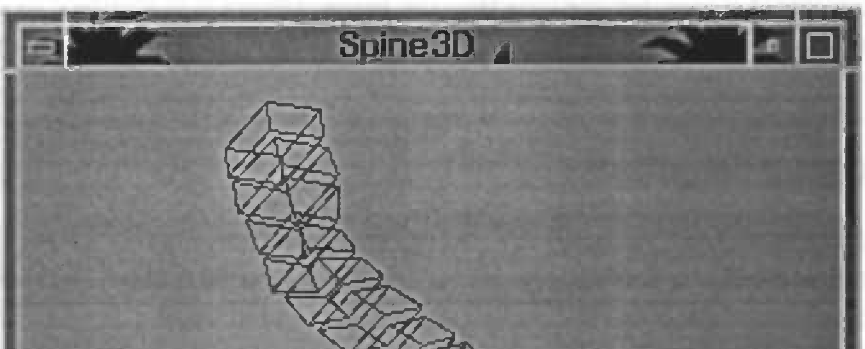


Figure 14: Three dimensional spine image

Conclusions

Amongst the imaging techniques that have been investigated, there is no technique that can meet all the demands stated for speed, harmlessness and accuracy. Because the main stumbling block is accuracy, it has been tried to find a solution in combining information on the position of the spine emerging from different imaging techniques in such a way, that the result is a more accurate one. Due to the absence of data from these different imaging techniques, there are not yet results to prove or disprove this possibility. However, experiments with combining data from different X–ray images provided no problems at all, suggesting that it ought to be possible.

This experiments included making a syntactic model for the spine. With this mode it is possible to represent the shape of the spine as it appears in the X–ray images of the spine that are currently used by the instrument maker for constructing the brace. I think it is justified to conclude that this model provides a well–founded basis for further development of the method as proposed in chapter 2.

Future recommendations

Because there was not enough time to implement all the ideas proposed in the previous chapters the implementation itself is unfortunately not yet complete, several parts of the system still have to be implemented, other parts can still be improved. Recommendations for extensions and improvements are the following:

- The input itself may be improved: instead of letting an expert pin–point landmarks in (X–ray) images (which is time consuming, therefore expensive and not guaranteed without errors), it should be possible to do this automatically. This may be accomplished by using (fuzzy) techniques for noise reduction, edge detection, clustering ([34], [35]), etc. More accurate data implies less fuzzyness in the model and thus a more accurate image.
- More data should become available for several purposes. The first of these purposes would be ‘training’. That is, using the training data for constructing a syntactic reference model that can be used as ‘default’ model in absence of data, or for comparing data with this model in order to assign a fuzzy measure representing the accuracy of the data (like ‘very’, ‘little’, etc). This technique is also known as data fusion. The second purpose would be the testing of the model: currently the test set is the same as the design set, therefore the performance of the model can not be measured objectively.
- Data emerging from new imaging techniques ought to be used in combination with X–ray data. This should provide us with enough in-

formation to enable defining new, even better models, merging operations for merging these new models with exsinting ones, and new fuzzy features that can express natrual language expressions used by experts (e.g. 'deformed', 'large curve', etc.).

- There does not yet exist a graphical user interface (GUI) combining the functionality of the model builder and the viewer. A good GUI would provide means for easy combination of data, default settings, viewer output adaptation, etcetera. The functionality provided by the software does not actually change, but the overall result would be a user friendly software package.

Literature

[1] K. Rohrschneider, R. O. W. Burk, H. E. Völcker, *Reproducibility of topometric data acquisition in normal and glaucomatous optic nerve heads with the laser tomographic scanner*, Greafe's Archive for Clinical and Experimental Opthalmology, vol. 231, pag. 457–464, februari 1993.

[2] K. Rohrschneider, R. O. W. Burk, H. E. Völcker,T. Takamoto, B. Schwartz, Laser scanning tomografy and stereophotogrammetry in three–dimensional optic disc analysis, Greafe's Archive for Clinical and Experimental Opthalmology, vol. 231, pag. 193–198, november 1992.

[3] R. J. Bartrum, H. C. Crow, *Transillumination lightscanning to diagnose breast cancer*, Am. J. Radiol., vol. 142, pp. 409–414, 1984.

[4] R. Lafreniere, F. S. Ashkar, A. S. Ketcham, *Infrared light scanning of the breast*, Am. Surg., vol. 52, pp. 123–128, 1986.

[5] G. Jarry S. Debray, J. Perez, J. P. Lebebvre, M. de Ficquelmont, A. Gaston, *In vivo transillumination of the hand using near infrared laser pulses and differential spectroscopy*, Ibid., vol. 11, pp. 293–299, 1989.

[6] G. Jarry, S. Ghesquiere, J.M. Maarek, F. Fraysse, S. Debray, Bui–Mong–Hund, D. Laurent, *Imaging mamalian tissues and organs using laser collimated transillumination*, J. Boimed. Eng., vol 6. pp. 70–73, 1984.

[7] B. Chance, J. S. Leigh, H. Miyake, D.S. Smith, S. Nioka, R. Greenfeld, R. Finander, K. Kaufmann, W. Levy, *Comparison of time–resolved and –unresolved measurements of deoxohemoglobin in the brain*, Proc. Nat. Acad. Sci. USA, vol. 85, pp. 4971–4975, 1988.

[8] H. Key, P. C. Jackson, P. N. T. Wells, N*ew approaches to transillumination imaging*, J. Biomed. Eng., vol. 10, pp. 113–118, 1988.

[9] M. aan de Brugh, *Gedachten op de foto*, Intermediar, nr. 36, pag. 27–28, 9 september 1994.

[10] D. S. Dilworth, E. N. Leith, J. L. Lopez, *Imaging absorbing structures within thick diffusing media}*, Appl. Optics, vol. 29, pp. 691–698, 1990.

[11] J. C. Hebden, R. A. Kruger, *Transillumination imaging performance: spatial resolution simulation studies*, Med. Phus., vol. 17, pp. 41–47, 1990.

[12] G. Jarry, L.P. Lefebvre, S. Debray, J. Perez, *Laser tomography of heterogeneous scattering media using spatial and temporal resolution*, Medical and Biological Engineering and Computing, vol. 31, pp. 157–164, march 1993.

[13] J. B. T. M. Roerdink, *Computerized Tomografy and its Applications: a Guided Tour*, Computer Science Notes CS–9206, Department of Computing Science, University of Groningen.

[14] G. de Mey, G. Merckaert, *Infraroodtermografie; Ruime mogelijkheden voor industrie en geneeskunde*, Het Ingenieursblad, nr. 11, 1990.

[15] S. Willner, *Moiré Topography for the Diagnoses and Documentation of Scoliosis*, Acta. orthop. scand., Vol. 50, pp. 295–302, 1979.

[16] R. Turner–Smith, J. D. Harris, G. R. Houghton, R. J. Jefferson, *A Method for Analysis of Back Shape in Scoliosis*, J. Biomechanics, Vol. 21, No. 6, pp. 497–509, 1988.

[17] R.L. van Renesse, J. W. Klumper, *Glas in voedsel – genezen is niet altijd te voorkomen*, VTM, nr. 24, pag. 31–34, 18 november 1993.

[18] D. Landeweer, *Computer vision in de strijd tegen gecontamineerde eindprodukten*, VTM, nr. 24 pag. 43–45, 18 november 1993.

[19] H. J. Stroot, A. L. Faber, *Design and realization of an automated vertebral Morpometry Algorithm*, Master's Thesis, Department of Electrical Engineering, University of Twente, 1994.

[20] L.O. Hall, A.M. Bensaid, L.P. Clarke, R.P. Veldthuizen, M.S. Silbiger and J.C. Bezdek. *A Comparison af Neural Network and Fuzzy Clustering Techniques in Segmenting Magnetic Resonance Images of the Brain*, IEEE Transactions on Neural Networks, vol. 3, no. 5, pp. 672–682, september 1992.

[21] Y. W. Lin, S. U. Lee, *On the Color Image Segmentation Algorithm Based on the Thresholding and Fuzzy c–means Techniques*, Pattern Recognition, vol. 23, no. 9, 1990.

[22] G. Coppini, R. Poli, M. Rucci, G. Valli, *A Neural Netwrok Architecture for Understanding Discrete Three–Dimensional Scenes in Medical Imaging*, Computers and Biomedical Research, vol. 25, pag 569–585, december 1992.

[23] S. K. Pal, R. A. King, *Image Enhancement using Smoothing with Fuzzy Sets*, IEEE Trans. Syst., Man., Cybern., vol. SMC–11, no. 7, pp. 494–501, juli 1981.

[24] S. K. Pal, R. A. King, *On Edge Detection of X–ray Images using Fuzzy Sets*, IEEE Trans. Pattern Anal. Machine Intell., vol PAMI–5, no. 1, pp. 69 –77, 1983.

[25] S. A. Kwabwe, S. K. Pal, R. A. King, *Recognition of Bones from X–ray of the Hand*, Int. J. Syst. Sci., vol. 16, no. 4, pp. 403–413, 1985.

[26] S. K. Pal, R. A. King, *Fuzzy Grammars in Syntactic Recognition of Skeletal Maturity from X–rays*, IEEE Trans. Syst., Man, Cybern., vol. SMC–16, no. 5, pp. 657–667, september/ october 1986.

[27] G. J. Docter, A. G. Veldhuizen, *Recognition of spinal deformities by means of a 3D back–shape analisys*, Proceedings of Symposium "*The human spine in research and practice*", pg. 62–65, 1985.

[28] A. G. Veldhuizen, *Ideopathic scoliosis. A biomechanical and functional anatomy study*, thesis, Groningen 1985.

[29] M. B. Coëlho, G. Kloosterhuis, *Zakwoordenboek der Geneeskunde*, geheel herziene druk, Elsevier/ Koninklijke PBNA, 1989.

[30] L. A. Zadeh, *Fuzzy Sets*, Information and Control, vol 8., pp. 338–353, 1965.

[31] J. C. Bezdek, S. K. Pal, *Fuzzy Models for Pattern Rectognition*, IEEE Press, The Institute of Electrical and Electronics Engineers, Inc., New York, 1982.

[32] E. T. Lee, *Fuzzy Tree Automata and Syntactic Pattern Recognition*, IEEE Trans. Pattern Anal. Machine Intell., vol. PAMI–4, no. 4, pp. 445–449, july 1982.

[33] R. Krishnapuram, J. M. Keller, Y. Ma, *Quantative Analysis of Properties and Spatial Relations of Fuzzy Image Regions*, IEEE Transactions on Fuzzy Systems, vol. 1, no. 3, pp. 222–233, august 1993.

[34] E. H. Ruspini, *A New Approach to Clustering*, Inform. Control, vol. 15, no. 1, pp. 22–32, July 1969.

[35] M. P. Windham, *Geometrical Fuzzy Clustering Algorithms*, Fuzzy Sets and Systems, vol. 10, pp. 271–279, 1983.

Appendices

A: MoBI source code

A.1 frtfta.h

```
#ifndef _FRTFTA_H
#define _FRTFTA_H

#define Equal(x,y)  (fabs ((x)-(y)) < eps)
#define max(x,y)    (((x) > (y)) ? (x) : (y))
#define min(x,y)    (((x) < (y)) ? (x) : (y))
#define mind(i,j,ilen) ((i)+(j)*(ilen))
#define even(x)     (1 - ((x) % 2))  /* x mod 2 = 0 --> x is even */

extern int          debuglevel;
extern double       eps;

#endif
```

A.2 frtfta.c

```
#include <stdio.h>
#include <stdarg.h>
#include <math.h>
#include <stdlib.h>

#include "frtfta.h"
#include "prcs_arg.h"
#include "version.h"
#include "vector.h"
#include "spine.h"
#include "misc.h"


/* Initalisation and default setting of option parameters          */

static int ShowHelp  = 0;               /* show the help */

static int v         = 17;              /* number of vertebrae */
static int vp        = 4;               /* number of datapoints  */

int debuglevel       = 0;               /* (test) fase  */
double eps           = 1e-3;            /* error tolerance */

static char * InFileName     = NULL;    /* input pattern-file  */
static char * InFileName_lat = NULL;    /* second input pattern-file  */
static char * OutFileName    = NULL;    /* output pattern-file */
static char * ExecFileName   = NULL;    /* name of the executable */
```

```
void intro()
{
                printf("Running %s version %s, compiled by %s, %s%s,\n",
                                ExecFileName, VERSION, USER, BUILD_DAY, BUILD_DATE);
                printf("on the %s, a %s platform.\n\n", MACHINE_NAME, OS_RELEASE);
}

static void Show_Parameters (void)
{
                printf("Vertebrae          == %d\n", v);
                printf("Vertebraepoints == %d\n", vp);
                printf("Debuglevel        == %d\n", debuglevel);
                printf("Epsilon              == %f\n", eps);
                printf("\n");
}

#define ARGTABLESIZE 8
struct argtable ArgTable [ARGTABLESIZE] = {
                { "-help"           ,   ARG_NONE, &ShowHelp },
                { "-vertebrae"      , ARG_INT , &v },
                { "-vertebraepoints" , ARG_INT , &vp },
                { "-debuglevel"     , ARG_INT , &debuglevel },
                { "-epsilon"        , ARG_DBL , &eps },
                { "-ventral"        , ARG_STR , &InFileName },
                { "-lateral"        , ARG_STR , &InFileName_lat },
                { "-outfile"        , ARG_STR , &OutFileName }
};

static void show_usage (void)
{
                int i;

                printf ("Program : %s\n", ExecFileName);
                intro();
                printf ("Options :\n");
                for (i = 0; i < ARGTABLESIZE; i++) {

                                printf ("    %-15s ", ArgTable [i].optstring);
                                switch (ArgTable [i].arg_fmt) {
                                                  case ARG_NONE :
                                                break;
                                                  case ARG_STR : printf ("string");
                                                break;
                                                  case ARG_INT : printf ("integer");
                                                break;
                                                  case ARG_DBL : printf ("double");
                                                break;
                                                  default : break;
                                } /* switch */

                printf ("\n");
                } /* for (i) */

                printf ("\n");
                exit (1);
}

/*
-- Function Name : Process_Parameters
-- Description     : Post-process the parameters given on the command line
*/

static void Process_Parameters (void)
{
                if (ShowHelp) show_usage ();

                if (v < 17 | v >= 19) failure ("Number of vertebrae must be 17 or 18.\n");

                if (vp < 4 | vp > 8) failure ("Number of vertebraepoints must be between 4 and 8.\n");

                if (InFileName_lat == NULL) {
                        fprintf (stdout,"No lateral input file specified, copying ventral name.\n");
                        InFileName_lat = InFileName;
            }

            if (OutFileName == NULL) {
                        fprintf (stdout,"No output file specified, stdout presumed.\n");
            }

            if (eps <=0 | eps >1) failure ("Epsilon should be between 0 and 1.\n");

}

/*
-- Function name : main
-- Description     : execute main program
*/

int main (int argc, char *argv [])
{
            Spine2D *S2D;
             Spine2D *S2D_lat;
            Spine3D *S3D;

            FILE *ipf     = NULL;
            FILE *ipf_lat = NULL;
            FILE *opf     = NULL;

            if (debuglevel > 5) fprintf (stderr, "Running.\n");

            ExecFileName = argv [0];
            argc--;
            argv++;

            intro();

            if (argc > 0) {
                        init_process_args (ARGTABLESIZE, ArgTable);
                        parse_args        (&argc, argv);
                        term_process_args ();
            }

            Process_Parameters ();
            Show_Parameters ();
            if (debuglevel > 2) fprintf (stderr, "Parameters processed.\n");

            if (!(ipf = fopen (InFileName, "r" )))
            failure ("Could not open input file %s\n", InFileName);

            if (!(ipf_lat = fopen (InFileName_lat, "r" )))
            failure ("Could not open input file %s\n", InFileName_lat);

            if ((opf  = fopen (OutFileName, "wb" )) == NULL)
            failure ("Could not open output file %s\n", OutFileName);

            S2D = LoadSpine2D (ipf, InFileName, v, vp);

            S2D_lat = LoadSpine2D (ipf_lat, InFileName_lat, v, vp);

/*
            Spine2DColorPrint (S2D, opf);
*/

            S3D = MergeSpine2Dto3D (S2D, S2D_lat);

            Spine3DColorPrint (S3D, opf);

            fclose (ipf);
            fclose (ipf_lat);
            fclose (opf);

            fprintf (stdout, "\nDone.\n");
```

```
                return (0);

}
```

A.3 spine.h

```
#ifndef _SPINE_H
#define _SPINE_H

typedef struct
{
                int  row, col;
                float *data;
} Matrix2D;

typedef struct
{
                float x, y;
} twotuple;

typedef struct
{
                Vector *BL, *BR, *OL, *OR;
} Vertebra2D;

typedef struct
{
                Vertebra2D *ventral;
                Vertebra2D *dorsal;
} Vertebra3D;

typedef struct Spine2D
{
                Vertebra2D *vertebra;
                struct Spine2D *next;
} Spine2D;

typedef struct Spine3D
{
                Vertebra3D *vertebra;
                struct Spine3D *next;
} Spine3D;


Vertebra2D *NewVertebra2D        (void);
Spine2D   *NewSpine2D           (void);

void    Vertebra2DPrint             (Vertebra2D *, FILE *);
void    Spine2DPrint                (Spine2D *, FILE *);
void    Spine3DPrint               (Spine3D *, FILE *);
void    Spine2DColorPrint          (Spine2D *, FILE *);
void    Spine3DColorPrint          (Spine3D *, FILE *);

Spine2D  *LoadSpine2D              (FILE *, char *, int, int);
Spine3D   *MergeSpine2Dto3D        (Spine2D *, Spine2D *);

#endif
```

A.4 spine.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>

#include "frtfta.h"
#include "misc.h"
```

```
#include "vector.h"
#include "spine.h"

/*
-- Function name : FuzzyRectangleMemb
-- Description   : Given four cosines, compute the membership of the
                   approximate rectangle they form.
*/
float FuzzyRectangleMemb (float A, float B, float C, float D)
{
                   return (1 - ((fabs(A)+fabs(B)+fabs(C)+fabs(D))/4));
}

/*
-- Function name : FuzzyVertMemb
-- Description   : Calculate the membership for the approximate vertebra.
*/
float FuzzyVertMemb (Vertebra2D *V)
{
                   float fvm;
/* The vectors point from A towards B, from B towards C etc, where A, B, C
   and D are the cornerpoints of the vertebral body rectangle: A----------B
                                                               |          |
                                                               |          |
                                                               C----------D

*/
                   Vector *AB = NewVector();
                   Vector *AC = NewVector();
                   Vector *BA = NewVector();
                   Vector *BD = NewVector();
                   Vector *DB = NewVector();
                   Vector *DC = NewVector();
                   Vector *CA = NewVector();
                   Vector *CD = NewVector();

                   VectorMin(AB, V->BR, V->BL);
                   VectorMin(AC, V->OL, V->BL);
                   VectorNegate(BA, AB);
                   VectorMin(BD, V->OR, V->BR);
                   VectorNegate(DB, BD);
                   VectorMin(DC, V->OL, V->OR);
                   VectorNegate(CA, AC);
                   VectorNegate(CD, DC);

                   fvm = FuzzyRectangleMemb (VectorCosAngle (AB, AC),
                                             VectorCosAngle (BA, BD),
                                             VectorCosAngle (DB, DC),
                                             VectorCosAngle (CA, C),
                                             );

                   if (debuglevel > 4)
                   fprintf (stdout, "Looks like rectangle membership: %f\n", fvm);

/* Return some space */

                   free(AB);
                   free(AC);
                   free(BA);
                   free(BD);
                   free(DB);
                   free(DC);
                   free(CA);
                   free(CD);

                   return fvm;
}

/*
-- Function name : FuzzySpineMemb
-- Description   : Calculate the membership for the approximate spine.
*/
float FuzzySpineMemb (Spine2D *S)
```

```
{
                float fvm;
                float fsm = 1.0;

                for (;S;S=S->next) {
                                fvm = FuzzyVertMemb (S->vertebra);
                                fsm = min(fsm, fvm);
                }

                return fsm;
}

/*
-- Function name : NewVertebra2D
-- Description   : Allocate room for a 2D vertebra.
*/
Vertebra2D *NewVertebra2D (void)
{
                return ((Vertebra2D *) alloc (sizeof (Vertebra2D)));
}

/*
-- Function name : NewVertebra3D
-- Description   : Allocate room for a new 3D vertebra.
*/
Vertebra3D *NewVertebra3D (void)
{
                return ((Vertebra3D *) alloc (sizeof (Vertebra3D)));
}

/*
-- Function name : NewSpine2D
-- Description   : Allocate room for a new 2D spine.
*/
Spine2D *NewSpine2D (void)
{
                return ((Spine2D *) alloc (sizeof (Spine2D)));
}

/*
-- Function name : NewSpine3D
-- Description   : Allocate room for a new 3D spine.
*/
Spine3D *NewSpine3D (void)
{
                return ((Spine3D *) alloc (sizeof (Spine3D)));
}

/*
-- Function name : NewMatrix2D
-- Description   : Allocate room for a matrix to scan the input.
*/
Matrix2D *NewMatrix2D (int v, int vp)
{
                Matrix2D *M = (Matrix2D *) alloc (sizeof (Matrix2D));

                M->row = v;
                M->col = vp*2;

                M->data = (float *) alloc ( v * vp * 2  * sizeof(float));

                return M;
}

/*
-- Function name : Matrix2DPrint
-- Description   : Print the contents of the matrix containing the input.
*/
void Matrix2DPrint (Matrix2D *M, FILE *f)
{
                int i, j;
                for (j=0; j < M->row; j++) {

                                for (i=0; i < (M->col); i++) {

                                 if (even(i))
                                 fprintf(f, "%f ", M->data [ mind( i, j, M->col)] );

                                 else
                                 fprintf(f, "%f \n ", M->data [ mind( i, j, M->col)] );

                 }

                                fprintf(f, "\n");

                }

                fprintf(f, "\n");
}

/*
-- Function name : VectorPrint2File
-- Description   : Print a vector to a file in a viewable shape.
*/
void VectorPrint2File (Vector *V, FILE *f)
{
/* Different from VectorPrint because of demands viewer */

                fprintf (f, "(%f %f %f) ", V->vec[0], V->vec[1], V->vec[2]);
}

/*
-- Function name : Vertebra2DPrint
-- Description   : Print a vertebra to a file, resulting in an approximate
          rectangular shape when imaged using the viewer.
*/
void Vertebra2DPrint (Vertebra2D *V, FILE *f)
{
                VectorPrint2File(V->BL, f);
                VectorPrint2File(V->BR, f);
                VectorPrint2File(V->OR, f);
                VectorPrint2File(V->OL, f);
                VectorPrint2File(V->BL, f);
}

/*
-- Function name : Vertebra3DPrint
-- Description   : Print a vertebra to a file, resulting in an approximat
                   cubic shape when imaged using the viewer.
*/
void Vertebra3DPrint (Vertebra3D *V, FILE *f)
{
                Vertebra2DPrint  (V->ventral, f);
                Vertebra2DPrint  (V->dorsal, f);
                VectorPrint2File (V->dorsal->OL, f);
                VectorPrint2File (V->ventral->OL, f);
                VectorPrint2File (V->ventral->OR, f);
                VectorPrint2File (V->dorsal->OR, f);
                VectorPrint2File (V->dorsal->BR, f);
                VectorPrint2File (V->ventral->BR, f);
}

/*
-- Function name : Spine2DPrint
-- Description   : Print a two dimensional spine to a file, resulting in
                   a 2D approximate spine when imaged using the viewer.
*/
void Spine2DPrint (Spine2D *S, FILE *f)
{
                for (;S;S=S->next) Vertebra2DPrint (S->vertebra, f);
                fprintf (f, "\n");
                if (debuglevel > 1)
                fprintf (stderr, "Spine dumped to file.\n");

}

/*
-- Function name  : Spine3DPrint
-- Description    : Print a three dimensional spine to a file, resulting in
            a 3D approximate spine when imaged using the viewer.
*/
void Spine3DPrint (Spine3D *S, FILE *f)
{
                for (;S;S=S->next) {
                                Vertebra3DPrint (S->vertebra, f);
                }

                fprintf (f, "\n");

                if (debuglevel > 1)
                fprintf (stderr, "Spine (3D) dumped to file.\n");
}

/*
-- Function name  : Spine2DColorPrint
-- Description    : The same as Spine2DPrint, but now membership values
                  are added for showing colors when imaged using the viewer.
*/
void Spine2DColorPrint (Spine2D *S, FILE *f)
{
                for (;S;S=S->next) {
                                Vertebra2DPrint (S->vertebra, f);
                                fprintf (f, "color: %f\n", (FuzzyVertMemb (S->vertebra)));
                }
                fprintf (f, "\n");

                if (debuglevel > 1)
                fprintf (stderr, "Spine dumped to file.\n");

}

/*
-- Function name  : Spine3DColorPrint
-- Description    : The same as Spine2DColorPrint but now for a three
                  dimensional spine.
*/
void Spine3DColorPrint (Spine3D *S, FILE *f)
{
                for (;S;S=S->next) {
                                Vertebra3DPrint (S->vertebra, f);
                                fprintf (f, "color: %f\n",
                                                FuzzyVertMemb (S->vertebra->ventral));
                }
                fprintf (f, "\n");

                if (debuglevel > 1)
                fprintf (stderr, "Spine (3D) dumped to file.\n");

}

/*
-- Function name  : Dump_Matrix2D
-- Description    : Dump a matrix containing the input data to a file
                  in a viewable form.
*/
void DumpMatrix2D (Matrix2D *M, FILE *f)
{
                int j;

                for (j=0; j < M->row; j++)
                fprintf (f, "(%f %f 0.0) (%f %f 0.0) (%f %f 0.0) (%f %f 0.0) (%f %f 0.0)\n",
                                M->data [ mind ( 0, j, M->col )],
                                M->data [ mind ( 1, j, M->col )],
                M->data [ mind ( 2, j, M->col )],
                M->data [ mind ( 3, j, M->col )],
                M->data [ mind ( 6, j, M->col )],
                M->data [ mind ( 7, j, M->col )],
                M->data [ mind ( 4, j, M->col )],
                M->data [ mind ( 5, j, M->col )],
                M->data [ mind ( 0, j, M->col )],
                M->data [ mind ( 1, j, M->col )]);

        fprintf(f, "\n");

} /* DumpMatrix2D */

/*
-- Function name : Normalize2D
-- Description   : Normalize the input on the interval (0,0)-(1,1) retaining
                   the same proportional distances.
*/
void Normalize2D (Matrix2D *M)
{
        int i, j;
        float Maxval, Minval, MinX, MinY;

        MinX = M->data [ mind ( 0, 0, M->row)];
        MinY = M->data [ mind ( 1, 0, M->row)];

        Maxval = max (MinX, MinY);

        for (j=0; j < M->row; j++)
        for (i=0; i < M->col; i++) {

                if (even(i))
                MinX = min (MinX, M->data [mind(i,j,M->col)]);

                else
                MinY = min (MinY, M->data [mind(i,j,M->col)]);

                Maxval = max (Maxval, M->data [mind(i,j,M->col)]);

        }

        Minval = min (MinX, MinY);

        if (debuglevel > 2) {
                fprintf ( stderr,
                "Normalisation, with x = x - %f / %f\n en y = y - %f / %f\n",
                        MinX, Maxval - Minval, MinY, Maxval - Minval);
                fprintf ( stderr, "Minval=%f, Maxval=%f, MinX=%f, MinY=%f\n",
                        Minval, Maxval, MinX, MinY);
        }

        for (j=0; j < M->row; j++) {

                for (i=0; i < (M->col); i++) {

                        if (even(i)) M->data [ mind ( i, j, M->col )] -= MinX;

                        else M->data [ mind ( i, j, M->col )] -= MinY;

                        M->data [ mind ( i, j, M->col )] /= Maxval - Minval;

                }
        }

} /* Normalize2D */

/*
-- Function name : compare
-- Description   : A comparison function for two-tuples.
*/
int compare (const void *A, const void *B)
{
```

```
                twotuple *a = (twotuple *) A;
                twotuple *b = (twotuple *) B;

                if (a->y == b->y) return (int)(b->x - a->x);
                else return (int)(b->y - a->y);

}

/*
-- Function name : Sort_matrix
-- Description   : Sort the matrix coordinates along the length axis.
*/
void Sort_matrix2D (Matrix2D *M)
{
                qsort (M->data, 0.5*M->row*M->col, 2*sizeof(float), &compare);
}

/*
-- Function name : Matrix2DInit
-- Description   : Initialise and fill the matrix using the inputfile.
*/
Matrix2D *Matrix2DInit(FILE *ipf, char *InFileName, int v, int vp)
{
                Matrix2D *M = NewMatrix2D(v, vp);

                int i, j;

                for (j=0; j<M->row; j++)
                for (i=0; i<M->col; i++) {

                                if (!feof(ipf))
                                fscanf(ipf, "%f", &(M)->data [mind(i, j, M->col)]);

                                else failure("Not enough datapoints in inputfile %s.\n",
                                                   InFileName);

                }

                Normalize2D(M);

                return(M);

} /* Matrix2DInit */

/*
-- Function name : ConvertMatrix2DToSpine2D
-- Description   : Convert the input stored in the matrix to a binary
                   input tree-like shape.
*/
Spine2D *ConvertMatrix2DToSpine2D(Matrix2D *M)
{
                Spine2D *Spine = NULL;
                Spine2D *S;

                int j;

                for (j=M->row-1;j>=0;j--) {

                                if (debuglevel > 5)
                                fprintf(stderr,"Converting vertebra %d.\n", j);

                                S = NewSpine2D();

                                S->vertebra = NewVertebra2D();
                                S->vertebra->BL = NewVector();
                                S->vertebra->BR = NewVector();
                                S->vertebra->OL = NewVector();
                                S->vertebra->OR = NewVector();

                                VectorAssign ( S->vertebra->BL,
                                                   M->data [mind (0, j, M->col)],
                                                   M->data [mind (1, j, M->col)],
                              0.0);

                VectorAssign (S->vertebra->BR,
                              M->data [mind (2, j, M->col)],
                              M->data [mind (3, j, M->col)],
                              0.0);

                VectorAssign (S->vertebra->OL,
                              M->data [mind (4, j, M->col)],
                              M->data [mind (5, j, M->col)],
                              0.0);

                VectorAssign (S->vertebra->OR,
                              M->data [mind (6, j, M->col)],
                              M->data [mind (7, j, M->col)],
                              0.0);

                S->next = Spine;
                Spine = S;

        }

        return(Spine);

} /* ConvertMatrix2DToSpine2D */

/*
-- Function name : LoadSpine2D
-- Description    : Initialise the binary input tree using the input matrix
                    as an intermediar.
*/
Spine2D *LoadSpine2D (FILE *f, char *fname, int v, int vp)
{
        if (debuglevel > 2)
        fprintf (stderr, "Loading spine from file %s.\n", fname);

        return ConvertMatrix2DToSpine2D(Matrix2DInit (f, fname, v, vp));
}

/*
-- Function name : MergeSpine2Dto3D
-- Description    : Combine a ventral and and a lateral form in order to
                    construct a new 3 dimensional binary input tree.
*/
Spine3D *MergeSpine2Dto3D (Spine2D *ventral, Spine2D *lateral)
{
/* For now it is assumed that the lateral view is taken from the RIGHT.
   Left views also merge without problems, but show a mirrored spine.
*/

        Spine3D *Spine = NULL;
        Spine3D *S;

        for (; ventral; ventral = ventral->next, lateral = lateral->next) {

                if (debuglevel > 5)
                fprintf(stderr,"Merging vertebrae.\n");

                S = NewSpine3D();

                S->vertebra = NewVertebra3D();

                S->vertebra->ventral = NewVertebra2D();
                S->vertebra->dorsal  = NewVertebra2D();

                S->vertebra->ventral->BL = NewVector();
                S->vertebra->ventral->BR = NewVector();
                S->vertebra->ventral->OL = NewVector();
                S->vertebra->ventral->OR = NewVector();

                S->vertebra->dorsal->BL = NewVector();
                S->vertebra->dorsal->BR = NewVector();
                S->vertebra->dorsal->OL = NewVector();
                S->vertebra->dorsal->OR = NewVector();

                /* computing values of the ventral part */

                VectorAssign (S->vertebra->ventral->BL,
                              ventral->vertebra->BL->vec[0],
                              ventral->vertebra->BL->vec[1],
                             -(lateral->vertebra->BL->vec[0]));

                VectorAssign (S->vertebra->ventral->BR,
                              ventral->vertebra->BR->vec[0],
                              ventral->vertebra->BR->vec[1],
                              -(lateral->vertebra->BL->vec[0]));

                VectorAssign (S->vertebra->ventral->OL,
                              ventral->vertebra->OL->vec[0],
                              ventral->vertebra->OL->vec[1],
                              -(lateral->vertebra->OL->vec[0]));

                VectorAssign (S->vertebra->ventral->OR,
                              ventral->vertebra->OR->vec[0],
                              ventral->vertebra->OR->vec[1],
                              -(lateral->vertebra->OL->vec[0]));

                /* and for the dorsal part */

                VectorAssign (S->vertebra->dorsal->BL,
                              ventral->vertebra->BL->vec[0],
                              ventral->vertebra->BL->vec[1],
                              -(lateral->vertebra->BR->vec[0]));

                VectorAssign (S->vertebra->dorsal->BR,
                              ventral->vertebra->BR->vec[0],
                              ventral->vertebra->BR->vec[1],
                              -(lateral->vertebra->BR->vec[0]));

                VectorAssign (S->vertebra->dorsal->OL,
                              ventral->vertebra->OL->vec[0],
                              ventral->vertebra->OL->vec[1],
                              -(lateral->vertebra->OR->vec[0]));

                VectorAssign (S->vertebra->dorsal->OR,
                              ventral->vertebra->OR->vec[0],
                              ventral->vertebra->OR->vec[1],
                              -(lateral->vertebra->OR->vec[0]));

                S->next = Spine;
                Spine = S;

        }

        return (Spine);
}
```

A.5 vector.h

```
#ifndef __VECTOR_H
#define __VECTOR_H

typedef struct
{
        float vec[3];
} Vector;


typedef struct
{
        float mat[4][4];
} Matrix;
```

```
Vector          *NewVector          (void);
void            VectorAssign        (Vector *, float, float, float);
void            VectorCopy          (Vector *, Vector *);
int             ZeroVector          (Vector *);
void            VectorNegate        (Vector *, Vector *);
void            VectorPlus          (Vector *, Vector *, Vector *);
void            VectorMin           (Vector *, Vector *, Vector *);
void            VectorScalar        (Vector *, Vector *, float);
float           VectorDotProduct    (Vector *, Vector *);
void            VectorCrossProduct (Vector *, Vector *, Vector *);
float           VectorLength        (Vector *);
int             EqualVectors        (Vector *, Vector *);
void            VectorNormalize     (Vector *, Vector *);
void            VectorTransform     (Vector *, Matrix *, Vector *);
void            VectorPrint         (Vector *);
float           VectorCosAngle      (Vector *, Vector *);
void            MatrixZero          (Matrix *);
Matrix          *NewMatrix          (void);
void            MatrixIdentity      (Matrix *);
void            MatrixMultiply      (Matrix *, Matrix *, Matrix *);
void            MatrixPrint         (Matrix *);
Matrix          *TranslationMatrix   (Vector *);
Matrix          *RotationMatrix     (Vector *, Vector *, Vector *);
void            TransformationMatrix (Matrix *, Vector *, Vector *, Vector *);

#endif
```

A.6 vector.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <math.h>
#include <string.h>

#include "misc.h"

#include "vector.h"

#define EPS          0.01
#define Equal(x,y)   (fabs ((x)-(y)) < EPS)

Vector *NewVector (void)
{
                return (Vector *) alloc (sizeof (Vector));
}

void VectorAssign (Vector *V, float x, float y, float z)
{
                V->vec[0] = x;
                V->vec[1] = y;
                V->vec[2] = z;
}

void VectorCopy (Vector *V, Vector *W)
{
                memcpy (V, W, sizeof (Vector));
}

int ZeroVector (Vector *V)
{
                return ((Equal (0.0, V->vec[0])) && (Equal (0.0, V->vec[1])) && (Equal (0.0, V->vec[2])));
}

void VectorNegate (Vector *V, Vector *W)
{
                VectorAssign (V, -W->vec[0], -W->vec[1], -W->vec[2]);
}

void VectorPlus (Vector *V, Vector *W, Vector *X)
{
                VectorAssign (V, W->vec[0] + X->vec[0],
                                 W->vec[1] + X->vec[1],
                                 W->vec[2] + X->vec[2]);
}


void VectorMin (Vector *V, Vector *W, Vector *X)
{
                VectorAssign (V, W->vec[0] - X->vec[0],
                                 W->vec[1] - X->vec[1],
                                 W->vec[2] - X->vec[2]);
}


void VectorScalar (Vector *V, Vector *W, float x)
{
                VectorAssign (V, W->vec[0] * x,
                                 W->vec[1] * x,
                                 W->vec[2] * x);
}

float VectorDotProduct (Vector *V, Vector *W)
{
                return (V->vec[0] * W->vec[0]  +  V->vec[1] * W->vec[1]  +  V->vec[2] * W->vec[2]);
}

void VectorCrossProduct (Vector *V, Vector *W, Vector *X)
{
                VectorAssign (V, W->vec[1] * X->vec[2] - W->vec[2] * X->vec[1],
                                 W->vec[2] * X->vec[0] - W->vec[0] * X->vec[2],
                                 W->vec[0] * X->vec[1] - W->vec[1] * X->vec[0]);
}

float VectorLength (Vector *V)
{
                return sqrt (VectorDotProduct (V, V));
}

int EqualVectors (Vector *V, Vector *W)
{
                return (Equal (V->vec[0], W->vec[0])  &&
                        Equal (V->vec[1], W->vec[1])  &&
                        Equal (V->vec[2], W->vec[2]));
}

void VectorNormalize (Vector *V, Vector *W)
{
                float l = VectorLength (W);

                if (Equal (l, 0.0)) {
                        VectorAssign (V, 0.0, 0.0, 0.0);
                        return;
                }

                VectorAssign (V, W->vec[0] / l, W->vec[1] / l, W->vec[2] / l);
}


void VectorTransform (Vector *V, Matrix *M, Vector *W)
{
                int i, j;

                for (j = 0; j < 3; j++) {
                        V->vec[j] = M->mat[j][3];

                        for (i = 0; i < 3; i++) V->vec[j] += M->mat[j][i] * W->vec[i];
                }
}
```

```
void VectorPrint (Vector *V)
{
        int i;

        for (i = 0; i < 3; i++) fprintf (stderr, "%15.10f  ", V->vec[i]);
        fprintf (stderr, "\n");
}

float VectorCosAngle (Vector *V, Vector *W)
{
        return (VectorDotProduct (V, W) /
                (VectorLength (V) * VectorLength (W)));
}

void MatrixZero (Matrix *M)
{
        memset (M->mat, 0, sizeof (M->mat));
}

Matrix *NewMatrix (void)
{
        Matrix *M = (Matrix *) alloc (sizeof (Matrix));

        MatrixZero (M);

        return M;
}

void MatrixIdentity (Matrix *M)
{
        int i;

        MatrixZero (M);

        for (i = 0; i < 4; i++)
                M->mat[i][i] = 1.0;
}

void MatrixMultiply (Matrix *O, Matrix *M, Matrix *N)
{
        int i, j, k;

        for (k = 0; k < 4; k++)
        for (j = 0; j < 4; j++)
        for (i = 0; i < 4; i++)
                O->mat[k][i] += M->mat[k][j] * N->mat[j][i];
}

void MatrixPrint (Matrix *M)
{
        int i, j;

        for (j = 0; j < 4; j++) {
                for (i = 0; i < 4; i++)
                        printf ("%f  ", M->mat[j][i]);
                printf ("\n");
        }
}


Matrix *TranslationMatrix (Vector *V)
{
        int i;

        Matrix *M = NewMatrix ();

        MatrixIdentity (M);

        for (i = 0; i < 3; i++) M->mat[i][3] = V->vec[i];
```

```
                return M;
}


Matrix *RotationMatrix (Vector *V, Vector *W, Vector *X)
{
                int i;

                Matrix *M = NewMatrix ();

                MatrixZero (M);
                M->mat[3][3] = 1.0;

                for (i = 0; i < 3; i++)
                {
                                M->mat[0][i] = V->vec[i];
                                M->mat[1][i] = W->vec[i];
                                M->mat[2][i] = X->vec[i];
                }

                return M;
}

void TransformationMatrix (Matrix *Trans, Vector *VRP, Vector *VPN, Vector *VUP)
{
                Matrix *T;
                Matrix *R;
/*      Matrix *Trans;*/
                Vector Rx;
                Vector Ry;
                Vector Rz;
                Vector negVRP;
                Vector Tmp;

                VectorNegate (&negVRP, VRP);

                T = TranslationMatrix (&negVRP);

                VectorNormalize (&Rz, VPN);
                VectorCrossProduct (&Tmp, VUP, &Rz);
                VectorNormalize (&Rx, &Tmp);
                VectorCrossProduct (&Ry, &Rz, &Rx);

                R = RotationMatrix (&Rx, &Ry, &Rz);

                MatrixMultiply (Trans, R, T);

                free (R);
                free (T);

/*              return Trans;*/
}
```

A.7 prcs_arg.h

```
/*
-- PRCS_ARG - Process arguments from the command line
--
-- [09-11-1994] : Hindrik Hettema
*/

#ifndef _PRCS_ARG_H_
#define _PRCS_ARG_H_

#define ARG_NONE  0
#define ARG_STR   1
#define ARG_INT   2
#define ARG_DBL   3

struct argtable {
```

```
                    char *optstring;
                    int   arg_fmt;
                    void *arg_value;
};

void init_process_args (int argts, struct argtable *argtp);
void parse_args        (int *argc, char *argv []);
void term_process_args (void);

#endif
```

A.8 prcs_arg.c

```
/*
-- PRCS_ARG - Process arguments from the command line
--
-- [09-11-1994] : Hindrik Hettema
*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "prcs_arg.h"

static int             arg_table_size = 0;
static struct argtable *arg_table_ptr  = NULL;

void init_process_args (int argts, struct argtable *argtp)
{
 assert (argts > 0 && argtp != NULL);

 arg_table_size = argts;
 arg_table_ptr  = argtp;
}

static void error_table_option (int i)
{
 assert (0 <= i && i < arg_table_size);

 fprintf (stderr, "Illegal use of option %s\n", arg_table_ptr [i].optstring);
}

void parse_args (int *argc, char *argv [])
{
 int tmp_argc = 0;
 int i;

 while (tmp_argc < *argc) {
   for (i = 0; i < arg_table_size; i++) {
     if (!strcmp (arg_table_ptr [i].optstring, argv [tmp_argc])) {
       if (arg_table_ptr [i].arg_fmt == ARG_NONE) {
         *((int *) arg_table_ptr [i].arg_value) = 1;
       }
       else {
         if (tmp_argc + 1 == *argc)
           error_table_option (i);
         else {
           tmp_argc = tmp_argc + 1;

           switch (arg_table_ptr [i].arg_fmt) {
           case ARG_STR:
             *((char **) arg_table_ptr[i].arg_value) = argv [tmp_argc];
             break;
           case ARG_INT:
             *((int *) arg_table_ptr[i].arg_value) = atoi (argv [tmp_argc]);
             break;
           case ARG_DBL:
             *((double *) arg_table_ptr[i].arg_value) = atof (argv [tmp_argc]);
             break;
```

```
      default:
       break;
      } /* end switch */
     } /* end if */
    } /* end if */
    break;
   } /* end if */
  } /* end for */

  if (i = = arg_table_size) {   /* No option found */
   if (*(argv [tmp_argc]) = = '–') {
    fprintf (stderr, "Illegal option %s\n", argv [tmp_argc]);
   }
   else break;
  }
  tmp_argc+ +;
 } /* while */

 for (i = 0; i <= *argc – tmp_argc; i+ +)
  argv [i] = argv [tmp_argc + i];

 *argc – = tmp_argc;
}

void term_process_args (void)
{
 arg_table_size = 0;
 arg_table_ptr  = NULL;
}
```

A.9 misc.h

```
#ifndef __MISC_H
#define __MISC_H


#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void failure (char *msg, ...);
void *alloc (size_t size);

#endif
```

A.10 misc.c

```
#include "misc.h"

/*
– –Function name : failure
– –Description     : Print an error message for a variable number of arguments.
*/
void failure (char *msg, ...)
{
    va_list arglist;

    va_start (arglist, msg);

                vfprintf (stderr, msg, arglist);

    va_end (arglist);

    exit (1);
}

/*
– –Function name : alloc
```

```
--Description     : Improved version of malloc (halts with message on failure).
*/
void *alloc (size_t size)
{
      void *q;

      if ((q = (void *) malloc (size)) == NULL)
            failure ("Out of memory\n");

      return q;
}
```

A.11 version.h

```
#define BUILD_DATE "17 July 1995"
#define BUILD_TIME "16:49"
#define BUILD_DAY "Monday"
#define MACHINE_NAME "rug100"
#define USER "csg331"
#define OS_RELEASE "HP-UX A.09.01"
#define VERSION "1.8.4.b"
#define BUILD "1"
```

A.12 Makefile

```
#
#
# Makefile for MoBI (Model Builder/ Interpreter)
#
#

VERSION          = 3.2(3D)
BUILD            = 1

EXECUTABLE       = MoBI
MAIN             = frtfta.c
INTERFACES       = prcs_arg.c vector.c misc.c spine.c
LANGUAGE         = C++

X_LIBS           = -lXt -lX11
X_LIBPATH        = -L/usr/lib/X11R5
MOTIF_LIBPATH    = -L/usr/lib/Motif1.2
X_CFLAGS         = -I/usr/include/X11R5
MOTIF_CFLAGS     = -I/usr/include/Motif1.2

CPLUS_CC         = gcc
CPLUS_CFLAGS     =
CFLAGS           = -DSYSV  -DXOPEN_CATALOG -D__DEMO__\
                                 $(X_CFLAGS) $(MOTIF_CFLAGS) -Wall
LIBPATH          = $(X_LIBPATH) $(MOTIF_LIBPATH)
LIBS             = $(X_LIBS) -lm

VERSION_H        = version.h
PRCS_ARG_H       = prcs_arg.h
MISC_H           = misc.h
VECTOR_H         = vector.h $(MISC_H) frtfta.h
SPINE_H          = spine.h $(VECTOR_H) $(MISC_H) frtfta.h
FRTFTA_H         = frtfta.h $(PRCS_ARG_H) $(VERSION_H) $(SPINE_H) $(VECTOR_H) $(MISC_H)

EXTRA_OBJS       =
APPL_OBJS        =

OBJS             = $(MAIN:.c=.o) $(INTERFACES:.c=.o)\
                                 $(APPL_OBJS) $(EXTRA_OBJS)

.SUFFIXES: .o .c
```

```
$(EXECUTABLE): intro version $(OBJS)
#$(EXECUTABLE): intro $(OBJS)
                        @echo
                        @echo Linking $(EXECUTABLE)
                        $(CC) $(OBJS) $(LIBPATH) $(LIBS) -o $(EXECUTABLE)
                        @echo
                        @echo Stripping $(EXECUTABLE)
                        @strip $(EXECUTABLE)
                        @echo "Done"
                        @echo

.c.o:
                        @echo Compiling $< [$(LANGUAGE)]
                        $(CC) -c $(CFLAGS) $< -o $@

intro:
                        @echo
                        @echo Building MoBI, version information: \
                        version $(VERSION) build $(BUILD)

version:
                        @echo Creating version.h
                        @echo '#define BUILD_DATE "'‘date +"%e %B %Y"‘"' > version.h
                        @echo '#define BUILD_TIME "'‘date +"%H:%M"‘"' >> version.h
                        @echo '#define BUILD_DAY "'‘date +"%A"‘"' >> version.h
                        @echo '#define MACHINE_NAME "'‘hostname‘"' >> version.h
                        @echo '#define USER "'‘whoami‘"' >> version.h
                        @echo '#define OS_RELEASE "'‘uname -sr‘"' >> version.h
                        @echo '#define VERSION "'$(VERSION)'"' >> version.h
                        @echo '#define BUILD "'$(BUILD)'"' >> version.h
                        @echo Done
                        @echo

frtfta.o:               frtfta.c                 $(FRTFTA_H)
prcs_arg.o:             prcs_arg.c              $(PRCS_ARG_H)
vector.o:               vector.c                $(VECTOR_H)
spine.o:                spine.c                 $(SPINE_H)
misc.o:                 misc.c                  $(MISC_H)

CC = \
@‘if [ "$(LANGUAGE)" = "C++" ]; then echo $(CPLUS_CC) $(CPLUS_CFLAGS);fi‘ \
‘if [ "$(LANGUAGE)" = "ANSI C" ]; then echo $(ANSI_CC) $(ANSI_CFLAGS); fi‘\
‘if [ "$(LANGUAGE)" = "KR-C" ]; then echo $(KR_CC) $(KR_CFLAGS); fi‘
```

B: Rotate source code

Because the viewer ('Rotate') and de model builder ('MoBI') use some mutual files these files are not shown below. This are the files: vector.c, misc.h and misc.c. The version file is has the same shape for both programs and is therefore also shown once.

B.1 medcom.c

```
#include "rotate.h"

void ProcessString (Widget widget, XtPointer clientData, XtPointer callData)
{
                        XmTextPosition pos;
                        Position x, y;
                        char *string;
                        char line[100];
                        char *p, *q;
```

```
                    Widget w = (Widget) clientData;

                    pos = XmTextGetInsertionPosition (w);

                    string = XmTextGetString (w);

                    q = p = string + pos;

                    while (*q != 0  &&  *q != '\n') q++;

                    if (p != string) p--;

                    while (p != string  &&  *p != '\n') p--;

                    if (*p == '\n') p++;

                    strncpy (line, p, q - p);
                    line [q - p] = 0;

                    fprintf (stderr, "%s\n", line);

                    XtFree (string);

/*                  cmd = parseInput (line);*/

                    fprintf (stderr, "cmd = %d\n", cmd);
}

void Process (Widget widget, XKeyEvent *event, String *args, unsigned *argc)
{
                    ProcessString (NULL, widget, NULL);
}

void ResetIt (Widget widget, XtPointer clientData, XtPointer callData)
{
                    Reset (NULL, NULL, NULL, NULL);
}

void CloseIt (Widget widget, XtPointer, XtPointer)
{
                    cmd = EXIT;
}

void ReloadIt (Widget widget, XtPointer, XtPointer)
{
                    Reload(widget, NULL, NULL, NULL);
}

Widget InitMEdCom (XtAppContext app, Widget shell, ModelView *modelView)
{
                    Widget shell2;
                    Widget form;
                    Widget buttonApply;
                    Widget buttonReset;
                    Widget buttonClose;
                    Widget buttonReload;
                    Widget rc;
                    Widget text_w;

                    Arg args[10];
                    XtActionsRec actions;

/* MoBI: Model Builder/ Interpreter */

                    shell2 =            XtCreatePopupShell ("MoBI, topLevelShellWidgetClass,
                                          shell, NULL, 0);

                    form =              XtVaCreateManagedWidget ("form",
                                        xmFormWidgetClass, shell2,
                                        XmNwidth, 700,
                                        XmNheight, 350,
                                        NULL);
```

```
        buttonApply =  X   tVaCreateManagedWidget ("  Apply  ",
                           xmPushButtonWidgetClass,
                           form,
                           XmNbottomAttachment, XmATTACH_FORM,
                           XmNleftAttachment, XmATTACH_FORM,
                           NULL);

        buttonReset =      XtVaCreateManagedWidget ("  Reset  ",
                           xmPushButtonWidgetClass,
                           form,
                           XmNbottomAttachment, XmATTACH_FORM,
                           XmNleftAttachment, XmATTACH_WIDGET,
                           XmNleftWidget,  buttonApply,
                           XmNtopAttachment, XmATTACH_OPPOSITE_WIDGET,
                           XmNtopWidget, buttonApply,
                           NULL);

        buttonReload =     XtVaCreateManagedWidget ("  Reload  ",
                           xmPushButtonWidgetClass,
                           form,
                           XmNbottomAttachment, XmATTACH_FORM,
                           XmNleftAttachment, XmATTACH_WIDGET,
                           XmNleftWidget, buttonReset,
                           NULL);

        buttonClose =      XtVaCreateManagedWidget ("  Close  ",
                           xmPushButtonWidgetClass,
                           form,
                           XmNbottomAttachment, XmATTACH_FORM,
                           XmNrightAttachment, XmATTACH_FORM,
                           NULL);

        rc =               XtVaCreateManagedWidget ("control area",
                           xmRowColumnWidgetClass, form,
                           NULL);

        XtSetArg (args[0], XmNrows, 20);
        XtSetArg (args[1], XmNcolumns, 70);
        XtSetArg (args[2], XmNtextString, "");
        XtSetArg (args[3], XmNeditMode, XmMULTI_LINE_EDIT);

        text_w = XmCreateScrolledText (rc, "action_area", args, 4);

        actions.string = "Process";
        actions.proc   = (XtActionProc) Process;
        XtAppAddActions (app, &actions, 1);

        XtOverrideTranslations (text_w, XtParseTranslationTable ("<Key>Return: Process() newline()"));

        XtManageChild (text_w);

        XtManageChild (rc);
        XtManageChild (buttonApply);
        XtManageChild (buttonReset);
        XtManageChild (buttonClose);
        XtManageChild (buttonReload);
        XtManageChild (form);

        XtAddCallback (buttonApply, XmNactivateCallback, ProcessString, text_w);

        XtAddCallback (buttonReset, XmNactivateCallback, ResetIt, &modelView);

        XtAddCallback (buttonClose, XmNactivateCallback, CloseIt, NULL);

        XtAddCallback (buttonReload, XmNactivateCallback, ReloadIt, NULL);

        return shell2;
}
```

B.2 rotate.h

```
#ifndef __ROTATE_H
#define __ROTATE_H

#include <stdio.h>
#include <string.h>
#include <math.h>

#include "vector.h"
#include "view.h"
#include "graph.h"

#include <Xm/Xm.h>
#include <Xm/DrawingAP.h>
#include <Xm/DialogS.h>
#include <Xm/PanedW.h>
#include <Xm/Form.h>
#include <Xm/RowColumn.h>
#include <Xm/LabelG.h>
#include <Xm/TextF.h>
#include <Xm/Text.h>
#include <Xm/PushB.h>
#include <Xm/TextP.h>
#include <Xm/MainW.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/IntrinsicP.h>

#define EXIT 1
#define NOP  0

extern int cmd;

void Reset          (Widget, XKeyEvent *, String *, unsigned *);
void Reload         (Widget, XKeyEvent *, String *, unsigned *);

Widget InitMEdCom (XtAppContext, Widget, ModelView *);

#endif
```

B.3 rotate.c

```
#include "rotate.h"

#define SQR(x)      ((x)*(x))

int cmd;

static ModelView modelView;

#define Equal(x,y)       (fabs ((x)-(y)) < 0.01)

void Rotate (Vector *Direction, View *view, float angle)
{
                    static Vector P, nVUP, nVPN, Cross;
                    static Vector Diff, Tmp1, Tmp2, Dir, Pr;
                    static Vector UnitY = {0.0, 1.0, 0.0};
                    float ang, length;

                    VectorCrossProduct (&Cross, &(view->VUP), &(view->VPN));

                    ang = acos (VectorCosAngle (&UnitY, Direction));

                    if (Direction->vec[0] < 0.0) ang = -ang;

                    VectorScalar (&Tmp1, &(view->VUP), cos (ang));
                    VectorScalar (&Tmp2, &Cross,      sin (ang));
```

```
        VectorPlus (&Dir, &Tmp1, &Tmp2);

        if (ang < -0.5*M_PI || ang > 0.5*M_PI) {
                VectorNegate (&Dir, &Dir);
                angle = -angle;
        }

        VectorScalar (&Tmp1, &(view->VPN), cos (angle));
        VectorScalar (&Tmp2, &Dir,         sin (angle));

        VectorPlus (&nVPN, &Tmp1, &Tmp2);

        length = fabs (cos (ang));

        if (! Equal (length, 0.0)) {
                VectorScalar (&Pr, &Dir, length);

                VectorScalar (&Tmp1, &Pr, cos (angle));
                VectorScalar (&Tmp2, &(view->VPN), -sin (angle) * length);

                VectorPlus (&P, &Tmp1, &Tmp2);

                VectorMin (&Diff, &(view->VUP), &Pr);
                VectorPlus (&nVUP, &P, &Diff);
        }

        else VectorCopy (&nVUP, &(view->VUP));

        VectorNormalize (&(view->VPN), &nVPN);
        VectorNormalize (&(view->VUP), &nVUP);
}

void Zoom (Widget widget, XButtonEvent *event, String *args, unsigned int *argc)
{
        static Position x, y;

        if (modelView.view->multiple)
        return;

        if (! strcmp (args[0], "down")) {
                x = event->x;
                y = event->y;
        }
        else {
                float length = event->y-y;

                modelView.view->zoom += (length / 5.0);

                XClearWindow (XtDisplay (widget), XtWindow (widget));
                ViewIt (&modelView);

                x = event->x;
                y = event->y;
        }
}

void Reset (Widget widget, XKeyEvent *event, String *args, unsigned int *argc)
{
        if (modelView.view->multiple)
        return;

        ResetView (modelView.view);

        ViewIt (&modelView);
}

void Reload (Widget widget, XKeyEvent *event, String *args, unsigned int *argc)
{
        FILE *f = fopen ("in.out", "rt");
        modelView.planes = Load (f, widget);
        fclose (f);
```

```
                Reset(widget, event, args, argc);
}

void MouseMotion (Widget widget, XButtonEvent *event, String *args, unsigned int *argc)
{
                static Position x, y;

                if (modelView.view->multiple)
                                return;

                if (! strcmp (args[0], "down")) {
                                x = event->x;
                                y = event->y;
                }
                else {
                                float length = sqrt (SQR (event->x-x) + SQR (event->y-y));

                                if (length == 0.0) return;

                                Vector Dir = {(event->x-x)/length, (event->y-y)/length, 0.0};

                                Rotate (&Dir, modelView.view, length/100.0);

                                XClearWindow (XtDisplay (widget), XtWindow (widget));
                                ViewIt (&modelView);

                                x = event->x;
                                y = event->y;
                }
}

void Translate (Vector *Direction, View *view, float length)
{
                Vector Cross, Tmp1, Tmp2, Dir;
                Vector UnitY = {0.0, 1.0, 0.0};

                VectorCrossProduct (&Cross, &view->VUP, &view->VPN);

                float ang = acos (VectorCosAngle (&UnitY, Direction));

                if (Direction->vec[0] < 0.0) ang = -ang;

                VectorScalar (&Tmp1, &view->VUP, cos (ang));
                VectorScalar (&Tmp2, &Cross,   sin (ang));

                VectorPlus (&Dir, &Tmp1, &Tmp2);
    VectorScalar (&Dir, &Dir, length);

                VectorPlus (&view->VRP, &view->VRP, &Dir);
}

void VRPMotion (Widget widget, XButtonEvent *event, String *args, unsigned *argc)
{
                static Position x,y;

                if (modelView.view->multiple)
                return;

                if (! strcmp (args[0], "down")) {
                                x = event->x;
                                y = event->y;
                }
                else {
                                float length = sqrt (SQR (event->x-x) + SQR (event->y-y));

                                if (length == 0.0) return;

                                XClearWindow (XtDisplay (widget), XtWindow (widget));

                                Vector Dir = {-(event->x-x)/length,
                                                    (event->y-y)/length, 0.0};
                                Translate (&Dir, modelView.view, length/500.0);

                                ViewIt (&modelView);

                                x = event->x;
                                y = event->y;
                }
}

void main (int argc, char **argv)
{
                Widget shell, drawingArea, shell2, pane, form, shell1, rc, text_w;
                Widget buttonApply, buttonReset, buttonClose;
                XtAppContext app;
                XGCValues gcv;
                GC gc;
                XtActionsRec actions;
                XtInputMask inputMask;
                XEvent event;
                String av;
                unsigned ac;
                Arg args[10];

                unsigned char *string;

                String translations = "<Btn1Motion>:   MouseMotion(motion)\n \
                                       <Btn1Down>:     MouseMotion(down)\n \
                                       <Btn1Up>:       MouseMotion(up)\n \
                                       <Btn2Motion>:   Zoom(motion)\n \
                                       <Btn2Down>:     Zoom(down)\n \
                                       <Btn3Motion>:   VRPMotion(motion)\n \
                                       <Btn3Down>:     VRPMotion(down)\n \
                                       <Btn3Up>:       VRPMotion(up)\n \
                                       <Key>Return:    Reset()";

                shell =         XtVaAppInitialize (&app, "Rotate", NULL, 0,
                                                &argc, argv, NULL,
                                                XmNmappedWhenManaged, FALSE,
                                                NULL);

                XtRealizeWidget (shell);

                shell1 =        XtCreatePopupShell ("Spine3D", topLevelShellWidgetClass,
                                                shell, NULL, 0);

                shell2 = InitMEdCom (app, shell, &modelView);

                actions.string = "MouseMotion";
                actions.proc = (XtActionProc) MouseMotion;

                XtAppAddActions (app, &actions, 1);

                actions.string = "Zoom";
                actions.proc = (XtActionProc) Zoom;

                XtAppAddActions (app, &actions, 1);

                actions.string = "VRPMotion";
                actions.proc = (XtActionProc) VRPMotion;

                XtAppAddActions (app, &actions, 1);

                actions.string = "Reset";
                actions.proc = (XtActionProc) Reset;

                XtAppAddActions (app, &actions, 1);

                drawingArea =   XtVaCreateManagedWidget ("", xmDrawingAreaWidgetClass,
                                                shell1,
                                                XmNtranslations, XtParseTranslationTable (translations),
                                                XmNwidth, (argc > 1 ? 1 : 0)*400+ 300,
                                                XmNheight, 400,
                                                NULL, 0);

            XtAddCallback (drawingArea, XmNresizeCallback, ResizeView, &modelView);

            FILE *f = fopen ("in.out", "rt");
            modelView.planes = Load (f, drawingArea);
            fclose (f);
            modelView.view = InitView (drawingArea, (argc > 1 ? 1 : 0));

            modelView.view->gc = AllocateGC (drawingArea);

            XtAddCallback (drawingArea, XmNexposeCallback, ExposeView, &modelView);

            XtCreateManagedWidget ("", xmMainWindowWidgetClass,
                                                    shell1, NULL, 0);

            XtCreateManagedWidget ("", xmMainWindowWidgetClass,
                                                    shell2, NULL, 0);

            XtPopup (shell2, XtGrabNone);
            XtPopup (shell1, XtGrabNone);

            cmd = NOP;

            do {
                        while (cmd != NOP  &&  cmd != EXIT) {
            /*                      cmd = action ();*/

                                    ViewIt (&modelView);

                                    if (XtAppPending (app)) {
                                                XtAppNextEvent (app, &event);
                                                XtDispatchEvent (&event);
                                    }
                        }

                        XtAppNextEvent (app, &event);
                        XtDispatchEvent (&event);

            } while (cmd != EXIT);
}
```

B.4 view.h

```
#ifndef __VIEW_H
#define __VIEW_H

#include "graph.h"
#include "misc.h"
#include "vector.h"

#include <X11/Intrinsic.h>
#include <X11/Xlib.h>

typedef struct
{
            Vector  VRP;
            Vector  VPN;
            Vector  VUP;

            float   zoom;

            Dimension    width;
            Dimension    height;

            int     multiple;

            Widget  drawingArea;

            GC      gc;
} View;

typedef struct Vectors
{
                Vector *V;
                struct Vectors *next;
} Vectors;

typedef struct Planes
{
                Vectors V;
                Pixel  color;
    struct Planes *next;
} Planes;

typedef struct
{
                Planes    *planes;
                View      *view;
} ModelView;

void    Play          (View *view);
void    ViewIt        (ModelView *);
void     ResetView    (View *);
Planes *Load          (FILE *, Widget);

void   ExposeView     (Widget, XtPointer, XtPointer);
View   *InitView      (Widget, int);
void   ResizeView     (Widget, XtPointer, XtPointer);

#endif
```

B.5 view.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include "vector.h"

#define delay(x)      napms(x)

#include "view.h"


#define FALSE       0
#define TRUE        1

#define OnXY        0
#define OnXZ        1
#define OnYZ        2

#define DEBUG   0

/*****************************************************************************\
*
*                                                    ^ y
*                                                    |
*                                                    |
*                                                    |
*                                                    +---------->
*                                                   /          x
*                                                  /
*                                                |/ z
*
\*****************************************************************************/

typedef struct
{
```

```
                float vec[2];
} Vector2D;

static Vector   origin;
static Vector2D origin2D;

void Project (Vector *V, Vector2D *W, int on)
{
                switch (on) {
                                case OnXY: W->vec[0] = V->vec[0];
                                                 W->vec[1] = V->vec[1];
                                                 break;
                                case OnXZ: W->vec[0] = V->vec[0];
                                                 W->vec[1] = V->vec[2];
                                                 break;
                                case OnYZ: W->vec[0] = V->vec[2];
                                                 W->vec[1] = V->vec[1];
                                                 break;
                }
}

void Vector2DCopy (Vector2D *destination, Vector2D *source)
{
                memcpy (destination, source, sizeof (Vector2D));
}

void MoveTo (Vector *V)
{
                VectorCopy (&origin, V);
                Project (V, &origin2D, OnXY);
}

void LineTo (View *view, Vector *coord, int diff)
{
                static Vector2D projection;

                Project (coord, &projection, OnXY);

                DrawLine ( view->drawingArea, view->gc,
                                 (int) (origin2D.vec[0]*view->zoom) + diff + view->width/2,
                                 (int) (-origin2D.vec[1]*view->zoom) + view->height/2,
                                 (int) (projection.vec[0]*view->zoom) + diff + view->width/2,
                                 (int) (-projection.vec[1]*view->zoom) + view->height/2);

                VectorCopy (coord, &origin);
                Vector2DCopy (&origin2D, &projection);
}

int Draw (Matrix *Trans, Planes *planes, View *view)
/* Every plane that is drawn has it's own color now.. */
{
                Planes *p;
                Vectors *v;
                Vector V;

                for (p = planes; p; p = p->next) {

                                XSetForeground (XtDisplay (view->drawingArea),
                                                view->gc,
                                                p->color
                                                );

                                VectorTransform (&V, Trans, p->V.V);

                                MoveTo (&V);

                                for (v = &p->V; v; v = v->next) {
                                                VectorTransform (&V, Trans, v->V);

                                                LineTo (view, &V, 0);
                                }
                }
```

```
}

Planes *Load (FILE *f, Widget widget)
{
          char vectors[1025];
          char *s, *c;
          Vector V;
          Vectors *W;
          int first           = 1;
          Planes *q;
          Planes *planes      = NULL;
          float *color        = (float *) alloc (sizeof (float));

          while (! feof (f)) {

                    fgets (vectors, 1024, f);

                    s = vectors;

                    if (sscanf (s, "(%f %f %f)",  &V.vec[0], &V.vec[1], &V.vec[2] ) != 3) return planes;

                    if (first)  {
                              planes = NULL;
                              first = 0;
                    }

                    q = (Planes *) alloc (sizeof (Planes));
                    q−>next = planes;
                    planes = q;

                    if ((c = strstr (s, "color:")) != NULL) {
                              sscanf (c, "color: %f", color);
                              planes−>color = GetSomePixel (widget, color);
                    }
                    else planes−>color = GetSomePixel (widget, NULL);

                    planes−>V.V = (Vector *) alloc (sizeof (Vector));
                    W = &planes−>V;
                    VectorCopy (W−>V, &V);
                    W−>next = NULL;

                    if ((s = strstr (s, ")")) != NULL) s += 2;

                    while (sscanf (s, "(%f %f %f)", &V.vec[0], &V.vec[1], &V.vec[2]) == 3) {
                              W−>next = (Vectors *) alloc (sizeof (Vectors));
                              W = W−>next;
                              W−>next = NULL;
                              W−>V = (Vector *) alloc (sizeof (Vector));
                              VectorCopy (W−>V, &V);

                              if ((s = strstr (s, ")")) != NULL) s += 2;
                    }
          }

          return planes;

}

void Play (Planes *planes, View *view)
{
          static Matrix Trans;
          static Vector VRP = {0, 0, 0};
          static Vector VPN;
          static Vector VUP;

          if (! view−>multiple) {
                    MatrixZero (&Trans);
                    TransformationMatrix (&Trans, &(view−>VRP), &(view−>VPN), &(view−>VUP));

                    ClearWindow (view−>drawingArea);

                    Draw (&Trans, planes, view);
```

```
                return;
        }
}

void ViewIt (ModelView *modelView)
{
        Play (modelView->planes, modelView->view);
}

void ResetView (View *view)
{
        VectorAssign (&(view->VRP), 0.0, 0.0, 0.0);
        VectorAssign (&(view->VUP), 0.0, 1.0, 0.0);
        VectorAssign (&(view->VPN), 0.0, 0.0, 1.0);

        view->zoom = 200.0;
}

View *InitView (Widget drawingArea, int multiple)
{
        View *view = (View *) alloc (sizeof (View));

        view->drawingArea = drawingArea;
        view->multiple = multiple;

        ResetView (view);

        XtVaGetValues (drawingArea,
                XmNwidth, &(view->width),
                XmNheight, &(view->height),
                NULL);

        return view;
}

void ExposeView (Widget widget, XtPointer clientData, XtPointer callData)
{
        ModelView *modelView = (ModelView *) clientData;

        Play (modelView->planes, modelView->view);
}

void ResizeView (Widget widget, XtPointer clientData, XtPointer callData)
{
        ModelView *modelView = (ModelView *) clientData;

        XtVaGetValues (widget,
            XmNwidth, &(modelView->view->width),
                  XmNheight, &(modelView->view->height),
                  NULL);

        Play (modelView->planes, modelView->view);
}
```

B.6 graph.h

```
#ifndef __GRAPH_H
#define __GRAPH_H

#include <stdlib.h>
#include <math.h>

#include <Xm/Xm.h>
#include <X11/Xlib.h>

Pixel GetPixel                  (Widget, short, short, short);
Pixel GetPixelByName            (Widget, char *);
Pixel GetSomePixel              (Widget, float *);
GC    AllocateGC                (Widget);
```

```
void  DrawLine                    (Widget, GC, int, int, int, int);
void  ClearWindow                 (Widget);

#endif
```

B.7 graph.c

```
#include "graph.h"

Pixel GetPixelByName ( Widget w, char *colorname )
{
                Display *dpy          = XtDisplay ( w );
                int scr               = DefaultScreen ( dpy );
                Colormap cmap         = DefaultColormap ( dpy, scr );
                XColor color, ignore;

                if ( XAllocNamedColor ( dpy, cmap, colorname, &color, &ignore ) )
                return ( color.pixel );
                else {
                                XtWarning ( "Couldn't allocate color" );

                                 return ( WhitePixel ( dpy, scr ) );
                }
}

Pixel GetPixel ( Widget w, short red, short green, short blue )
{
                Display *dpy  = XtDisplay ( w );
                int     scr  = DefaultScreen ( dpy );
                Colormap cmap = DefaultColormap ( dpy, scr );
                XColor   color;

                color.red   = red;
                color.green = green;
                color.blue  = blue;
                color.flags = DoRed | DoGreen | DoBlue;

                if ( XAllocColor ( dpy, cmap, &color ) )
                return ( color.pixel );
                else
                {
                                XtWarning ( "Couldn't allocate requested color" );
                                return ( WhitePixel ( dpy, scr ) );
                }
}

Pixel GetSomePixel (Widget widget, float * colorval)
{
                Display *dpy         = XtDisplay ( widget );
                int scr              = DefaultScreen ( dpy );
                short red            = 0;
                short green          = 0;
                short blue           = 0;

                if (colorval) {

                /* stretch the colors a LOT*/

                                *colorval = fabs (( *colorval - 0.80) * 5.0 );

                                red = (short) (65535 * *colorval);

                                GetPixel (widget , red, green, blue);
                }

                else {
                                XtWarning ("Color not found in datafile");
                                return ( WhitePixel (dpy, scr));

                }
}
```

```
GC AllocateGC (Widget widget)
{
                    XGCValues values;

                    values.foreground  = WhitePixel        (XtDisplay (widget),
                                                            DefaultScreen (XtDisplay (widget)));
                    values.background  = WhitePixel (XtDisplay (widget),
                    DefaultScreen (XtDisplay (widget)));

                    values.join_style  = JoinRound;

                    values.line_width  = 3;

                    retum XtAllocateGC (widget, 0,
                                           GCForeground | GCBackground |
                                           GCJoinStyle | GCLineWidth,
                                           &values, GCForeground | GCBackground, 0);
}

void DrawLine (Widget widget, GC gc, int x1, int y1, int x2, int y2)
{
     XDrawLine (XtDisplay (widget), XtWindow (widget),
                                          gc, x1, y1, x2, y2);
}

void Flush (Widget widget)
{
                    XFlush (XtDisplay (widget));
}

void ClearWindow (Widget widget)
{
                    XClearWindow (XtDisplay (widget), XtWindow (widget));
}
```

B.8 Makefile

```
#
#
# Makefile for the viewer 'Rotate'
#
#

EXECUTABLE      = rotate
MAIN            = rotate.c
INTERFACES      = vector.c misc.c graph.c view.c medcom.c
LANGUAGE        = C++

X_LIBS          = -lXm -lXt -lX11

X_LIBPATH       = -L/usr/lib/X11R5
MOTIF_LIBPATH   = -L/usr/lib/Motif1.2
X_CFLAGS        = -I/usr/include/X11R5
MOTIF_CFLAGS    = -I/usr/include/Motif1.2
CPLUS_CC        = g++
CPLUS_CFLAGS    =
CFLAGS          = -DSYSV  -DXOPEN_CATALOG \
                            $(X_CFLAGS) $(MOTIF_CFLAGS)
LIBPATH         = $(X_LIBPATH) $(MOTIF_LIBPATH)
LIBS            = $(X_LIBS) -lm

VERSION_H       = version.h
MISC_H          = misc.h
GRAPH_H         = graph.h
VECTOR_H        = vector.h
VIEW_H          = view.h $(VECTOR_H) $(MODEL_H)
ROTATE_H        = rotate.h $(VECTOR_H) $(VIEW_H) $(GRAPH_H) \
                            $(MODEL_H) $(USER_IO_H)
```

```
EXTRA_OBJS          =
APPL_OBJS           =

OBJS                = $(MAIN:.c=.o) $(INTERFACES:.c=.o) $(APPL_OBJS) $(EXTRA_OBJS)

.SUFFIXES: .o .c

$(EXECUTABLE): $(OBJS)
                    @echo Linking   $(EXECUTABLE)
                    $(CC) $(OBJS) $(LIBPATH) $(LIBS) -o $(EXECUTABLE)
                    @echo Stripping $(EXECUTABLE)
                    @strip $(EXECUTABLE)
                    @echo "Done"

.c.o:
                    @echo Compiling $< [$(LANGUAGE)]
                    $(CC) -c $(CFLAGS) $< -o $@

rotate.o:           rotate.c            $(ROTATE_H)
vector.o:           vector.c            $(VECTOR_H)
medcom.o:           medcom.c            $(ROTATE_H)
misc.o:             misc.c              $(MISC_H)
view.o:             view.c              $(VIEW_H)

CC = \
@‘if [ "$(LANGUAGE)" = "C++" ]; then echo $(CPLUS_CC) $(CPLUS_CFLAGS);fi’ \
‘if [ "$(LANGUAGE)"   = "ANSI C" ]; then echo $(ANSI_CC) $(ANSI_CFLAGS); fi‘\
‘if [ "$(LANGUAGE)"   = "KR-C" ]; then echo $(KR_CC) $(KR_CFLAGS); fi’
```