Lawrence Berkeley National Laboratory
University of Applied Science of Fribourg

Studies of Mechanical Recording Media with 3D Surface Profiling Methods: Data Collection and Analysis

Noé Lutz, Michel Yerly

Project type	Diploma project 2005 Computer Science Department EIA-FR University of Applied Science of Fribourg
Students	Noé Lutz, Michel Yerly
Supervisors	Carl Haber, Vitaly Fadeyev
Responsible professors	Frédéric Bapst, Ottar Johnsen and Béat Hirsbrunner
Work place	Lawrence Berkeley National Laboratory California USA
Start/end dates	15.08.2005/ 23.10.2005

Contents

Chapter 1

Introduction

Some time ago, audio recordings were stored on mechanical supports such as the very widespread phonographic record.

The playback from theses media is produced by the movement of a stylus running inside a groove. This movement is often changed into an electrical signal, which can then be amplified by an amplifier before being turned to sound by a speaker.

Different optical reading methods have been investigated in the past to reconstruct the sound of such mechanical supports [1], [2], [3].

1.1 An optical reading system

Some important recordings were archived and may nowadays not be read by a standard device because the simple contact of the stylus could irreversibly damage the sound data. Therefore, an alternative method was developed, that consists of reading the audio data by an optical process which avoids any physical contact with the support. Different optical methods were designed to retrieve the sound information of mechanical supports. This section tries to show the different approaches presented in [1], [2] and [3].

1.1.1 The 2D scanner

The two-dimensional scanning process consists of acquiring, a two dimensional high-resolution monochromatic picture of the support using specially designed hardware. Each pixel of the picture will provide information about the slope of the corresponding location on the medium: the steeper the point, the darker the pixel will be.

Visual Audio Project

The Visual Audio project, which is being developed at the University of Applied Science of Fribourg in Switzerland, aims to retrieve audio data from old phonographic records that are attacked by fungi or simply damaged. Because of the fast degradation of these records, people who are working on this project take high-resolution photographs of them, which is considered to be a good archiving method.

The photograph is then scanned with a specifically designed machine and some image processing is applied to retrieve the sound. The main goal of this project is to save these old records before they become unreadable. This means that the speed of sound extraction is more important than the audio quality of the result. Currently the sound extraction is close to real-time, which means that the processing duration of a whole record is almost equal to the length of the audio track stored on it.

1.1.2 The 3D scanner

The three-dimensional scanning method works almost the same as the two-dimensional one. It differs only in the way of measuring the different points on the surface of the medium. Instead of measuring the slope of each point, it accurately measures their depth. This results in a heights map representing the surface of the scanned medium. After the data is acquired, some mathematical methods are applied to simulate the stylus displacement.

This method was applied to reconstruct sound from an Edison cylinder at the Lawrence Berkeley National Laboratory ([2]). Edison cylinders have vertical groove modulation compared to phonographic records that have lateral groove modulation. In order to achieve a reasonable signal quality when reading such media, a 3D imaging method is required.

1.2 Goal

Dictation belts are one of the mechanical supports for audio recordings that were used mainly to record voice. They replaced the dictation cylinders, that were previously used during the 1940s due to their more convenient size. In fact, dictation belts can be folded and put in a folder for storage. A dictation belt consists of a rectangle of acetate-type soft-surface material that looks like a film and loops back on itself as shown in Figure 3.1. The sound was embossed onto the surface of the belt by a needle. Dictation belts were used by the government, in particular for recording conversations in police cars in the 70s.

1.3 Achieved work

This project gave rise to Groovster, a powerful software that can be used to analyze the dictation belts and extract the sound from them with a quality close to what is restituted by the speaker of the dictation belt player. The software is very easily extendable to analyze other kinds of media such as Edison's cylinders or grammophonic records. The whole study about the dictation belts and the grammophinc records was part of this project and is ready to be used for the next projects.

There are two reasons for which the work on this project has been slightly differently focused than the way it is described in the tasklist. Reorganizing the work was not a that hard because it is a research project and it will eventually be continuated later.

The first reason was that the dictation belt recorder machine did not behave the expected way. Instead of passing the waves to a simple transfer function, it

introduced harmonic distortion. That is why it was taken apart and analyzed in more details.

The second reason is that the fact that a dictation belt is translucent and the lamp used for the confocal probe was dying introduced a lot of measurement errors in the scanned data file. Instead of fixing dust or scratches, resources were put on correcting measurement errors.

1.4 Report organization

This report is split in 9 chapters including this one and 9 appendices. The first chapter is an introduction to this project whereas the second one shows the submitted task list. The third chapter explains all the data aquisition and the fourth the data analysis process. In the fifth chapter a dictation belt recorder machine is studied. The sixth one is all about the software "Groovster" that was created for this project. The tests and result obtained are presented in chapter seven whereas the conclusion is in the eighth chapter. Finally come the acknowledgements as chapter nine. In the appendices you can find the user guide and the class diagrams for Groovster, information about its 3D engine, a part of the study of the grammophone records, noise measurement, data acquisition file format, a listing of the provided files, the schematic of the dictation belt machine and some scanner hardware specifications.

Chapter 2

Task List

This chapter describes the objectives set for this project and what tasks need to be accomplished in order to achieve it. The main goal is to be able to reconstruct recorded sound from dictation belts and to show that a reasonable quality can be achieved using a three-dimensional optical surface metrology method. This metrology method consist of scanning the surface of the mechanical support using optical probes, similar to the ones described in [1] and [2]. This principal objective can be separated into several stages:

1. Getting started
2. Record sample sounds
3. Data acquisition
4. Data visualization
5. Data analysis

These tasks do not have to be executed in the order above; see section 2.6 for a description of the planned schedule. In the following sections, the above tasks are detailed.

2.1 Getting started

During the first days of work the goal is to get familiar with the hardware and the software used by the 3D-scanner. The 3D scanner has been used to scan Edison cylinders before. Understanding the measurement process and learn to code using Labview will be some good tasks to work on. LabView is the programming language chosen by Carl Haber to control the whole hardware, i.e. the motion controller and the probe, and to perform the data acquisition.

2.2 Record sample sounds

The main goal of this project is to extract the best possible sound quality from dictation belts. To achieve this goal, understanding the way the dictation belt recorder affects the sound signal in order to apply a proper equalization to the

signal has to be done. Some sound samples generated by a special sound analysis hardware will be recorded on an empty dictation belt. This belt can then be read with the 3D-scanner and the resulting sound wave can easily be compared with the original sound. These experiments will help to measure the amount of noise present on the medium as well as the signal-to-noise ratio.

2.3 Data acquisition

In order to acquire data from the dictation belts, the hardware first need to be adapted, hardware configuration and stage setup to them. Then a Labview program is needed to control the scanner's servomotors, to read the data that comes from the confocal probe, and eventually to write the acquired data to a file.

2.4 Data visualization

The data visualization is the main graphical user interface to analyze the data. It displays the acquired data and gives the user an idea of the groove shape. It is designed to help debugging and to tune some parameters used in the data analysis.

2.4.1 GUI and 3D environment

The data analysis program will have a graphical user interface where the user can tune various parameters and look at the groove shape. The groove will be displayed in both 2D and 3D. This task consists basically of creating the main window of the analysis program and setting up a 3D environment, which will allow to draw the shape of the groove in three dimensions.

2.4.2 2D & 3D display

The idea of the 3D display is to show a portion of medium surface in three dimensions and to allow the user to rotate the point of view around the surface in order to get a qualitative idea of the data. The 3D display has to be able to create a three dimensional surface given the points acquired during the data acquisition and to draw it. A two dimensional plot of the recorded points will give the user a more precise view. These two views will be very helpful to find errors in the data, such as dust or scratches, and to sunderstand how these errors affect the data analysis.

2.5 Data Analysis

After the data is read from the medium, it has to be analyzed in order to reconstruct the sound. The goal of this task is to identify the groove, measure its displacement in time, and to apply various corrections on the resulting signal. This task has been split into the following "sub-tasks".

2.5.1 Data quality check

The measuring process needs to be validated before analyzing the acquired data. In order to do this a program has to be written either in Labview or another programming language that takes as an input the file that contains acquired data, and tells the user some statistics about the quality of the acquired data. The program would have to compute, for example, the percentage of badly measured points or to show the groove geometry.

2.5.2 Algorithm for groove tracking

This algorithm takes as an input a "landscape" ($h(x, y)$) and produces as an output a waveform ($A(t)$). It has to find the groove of the scanned medium and determine the movement that a playback stylus would have made playing the medium.

2.5.3 Bad regions correction

If the medium is damaged or if there is dust on it, the analysis process might be able to detect this and possibly correct the defects. Measurements errors can also be corrected.

2.5.4 Geometric correction

Some geometric correction might be applied to the data to avoid signal distortion. Such distortions could arise if the dictation belt is warped or if, for example, the stage is not perfectly aligned during the measurement.

2.5.5 Transfer function and filters

Once the sound is extracted from the media, it has to be manipulated so that it resembles the original recorded sound as closely as possible. For this reason a study of the transfer function of a real dictabelt recorder using the sound samples previously recorded has to be performed. Once this function has been studied the inverse of this transfer function has to be applied to the extracted sound.

2.5.6 Resampling

Depending on the quality of scanned data, a lower or higher number of samples than what should be written in the target sound file will probably be obtained. Therefore the sound signal has to be resampled to a standard computer sound frequency.

2.6 Schedule

Figure 2.1 lists all the tasks with the estimated time investment for each task.

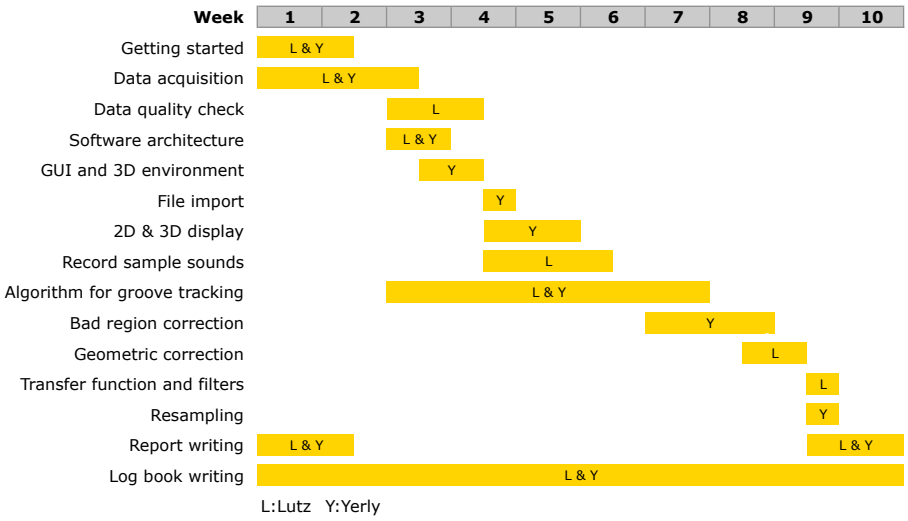


Figure 2.1: Schedule.

Chapter 3

Data Acquisition

3.1 Methodology

Sound extraction from mechanical sound carriers using 2D and 3D imaging methods has already been studied in the past [1], [3]. Particularly relevant to this project is the work that has been done previously in this lab to scan Edison cylinders using a 3D imaging technique [2].

In this work, 3D metrology is used to measure the surface structure of dictation belts. Dictaphone belts, also called dictation belts, are made of soft blueish plastic as shown in Figure 3.1. Sound is embossed in the material using a needle. The groove on these dictation belts is approximately $60\mu m$ wide but only $3\mu m$ deep, which makes it a very flat groove compared to gramophone records for example. The sound is encoded in the lateral movement of the groove. There are two different dictation belts: $15min$ and $30min$ dictation belts. The ones used in this project are $15min$ belts.

Figure 3.1: Mechanical recording medium of interest: dictation belt.

An example of the 3D structure measured using this technique is shown in Figure 3.2, and a cross-section of this data set is shown in Figure 3.3. Note that the two axes in Figure 3.3 are not on the same scale at all; the groove

is indeed flatter in reality. This data acquisition technique extracts basically a detailed map of the surface structure, making it possible to mathematically simulate the movement a stylus would have made if the belt had been played with a conventional dictation belt machine.

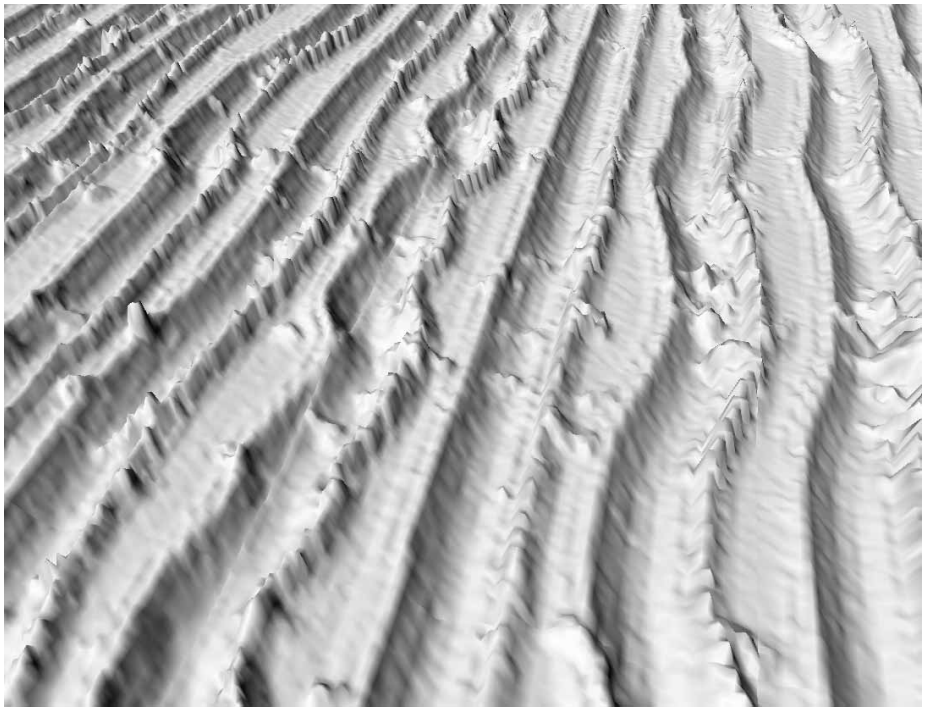


Figure 3.2: Measured surface structure of a dictation belt using 3D imaging techniques.

The 3D surface is obtained by measuring heights point-by-point using a color-coded confocal microscope. The confocal probe is moved in a controlled manner over the surface being measured, creating the height map shown in Figure 3.2.

In order for this technique to be viable, the resolution and accuracy of the measurement process have to be good enough to sense the smallest undulations on the medium surface caused by the sound. Several measurement parameters can also influence the quality of the sound extraction such as the spacing between the measurement points along the dictation belt axis as well as along the groove direction. The point spacing along the groove direction gives the sampling frequency of the optical playback. The point spacing along the dictation belt axis, on the other hand, contributes to the accuracy with which the groove profile geometry and position can be approximated at any given time. Furthermore, the noise added by the measurement process must be kept low enough to maintain a reasonable signal-to-noise ratio.

The following sections explain how the measurement hardware works and how it is set up.

3.1.1 Color-Coded Confocal Probe

The basic concept of color-coded confocal microscopy is show in Figure 3.4. A polychromatic pin-hole light source is used. The tiny light spot goes though a lens with exaggerated chromatic aberration before it is directed to the surface to measure. Because of the lens, each wavelength, i.e. color, comes into focus at a different depth. The light reflected by the surface is measured and analyzed by a spectrometer to find the wavelength that is in focus. This device works basically as peak detector, since the intensity of the wavelength that comes into focus on the surface will be the highest in the reflected signal.

The bright xenon light source inputs its signal into the high resolution optical sensor (CHR), that sends it to the confocal probe. When the light is reflected back from the surface to measure into the high resolution optical sensor, the difference between the two spectra is computed, i.e. the input and output spectrum, and the wavelength that has the highest intensity is measured. From the calculated wavelength, the distance between the probe and the surface can be computed. The accuracy of the measurement is typically around 100 nanometers in a range of approximately $330\mu m$.

In its the default configuration, the CHR is configured to measure the distance between the probe and the surface of the medium. This configuration does not work for dictation belts because they are transparent. The CHR gets confused when both surfaces, the top and the bottom of the belt, are within the $330\mu m$ range of the probe. There is a second measuring mode that measures the thickness of transparent materials. The thickness of the dictation belt is not directly interesting, but in this mode the distance of both surfaces is measured and only the distance to the first (closest) surface is used.

The confocal probe rate is the number of points the probe measures per second. This value can be set between $33Hz$ and $4kHz$. The exposure time for a measurement is given by the inverse of the frequency. The time between two measurements is considered to be negligible.

The CHR is connected to a computer via two serial links. One is used to send commands to the CHR and the other port is used to read out the measured data. Every distance measurement is encoded over 5 ASCII characters and send to the data channel. The five characters correspond to a numerical value in hundreds of microns. For example, the value "23159" means $231.59\mu m$. At the end of each measurement a carriage return and a line feed character are sent (CRLF). The command channel is used to configure the CHR parameters such as the number of measurements per second.

The CHR can be put in a standby mode in which it waits for an external trigger signal. While in standby mode, also called "waiting for trigger" mode, the CHR does not sending any data. As soon as the CHR receives a TTL trigger signal, it resumes the transmission of measurements with a response time of about 10 microseconds. This feature is used to synchronize the measurements with the moving linear stage, which is explained later.

Table 3.1 lists the characteristics of the confocal probe used.

3.1.2 Motion stages

When scanned, the dictation belt is put around a metallic cylinder that has a slightly smaller radius than that of the dictation belt. The metallic cylinder is

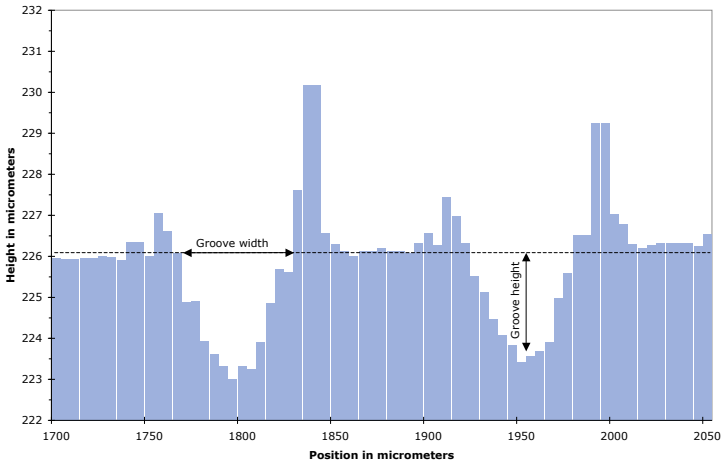


Figure 3.3: Cross-section of measured surface showing two grooves.

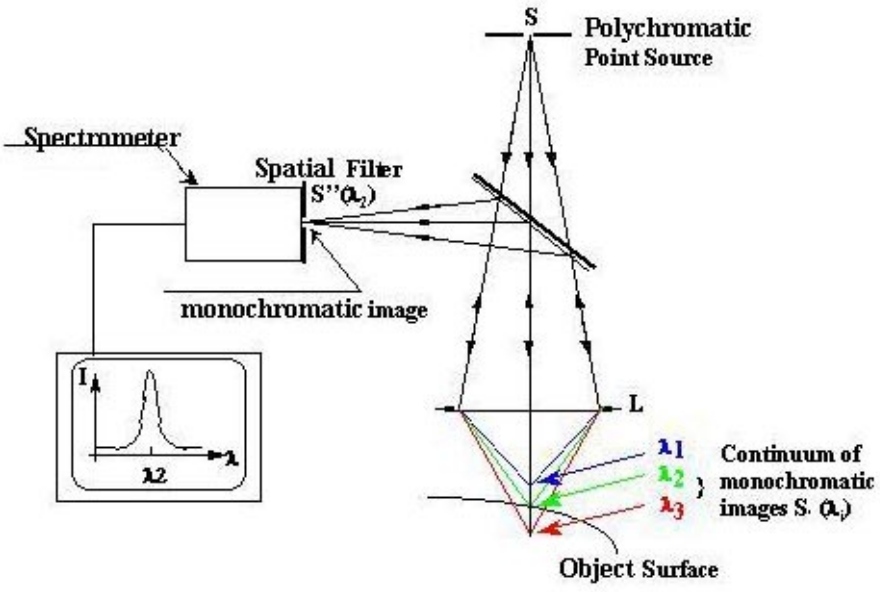


Figure 3.4: Basic concept of color-coded confocal microscopy. *The lens at position L has a large chromatic aberration causing the three wavelength shown to focus at different depths. Figure is courtesy of STIL SA, used by permission.*

Table 3.1: Color-coded confocal probe characteristics.

Name	Value
Manufacturer	STIL SA (France)
Probe Model	STIL CHR 150
Measurement Range	$350\mu m$
Sampling Frequency	$33Hz - 4000Hz$
Vertical Resolution	10 nanometers
Vertical Accuracy	100 nanometers
Trigger Response Time	10 microseconds

mounted onto a rotational motion stage. The confocal probe, is mounted onto a linear motion stage. Figure 3.5 shows the overall setup of the measurement hardware without the motion controller, which is in the computer.

Both the rotational stage as well as the linear stage are controlled by DC servo motors and read out by linear encoders. These two servo motors are controlled by a motion controller which is in the PC. Both stages have a control loop similar to a PID controller in order to achieve higher accuracy. The resolution of the linear stage is 10 nanometers and its accuracy is around $0.4\mu m$.

3.1.3 Data Collection

The data measured from the confocal probe is sent over a serial link to a PC where the data acquisition software stores the data in a binary file.

3.1.4 Complete Picture

Figure 3.6 shows the whole hardware setup needed for the measurements. On the left, the motion controller is responsible to move the confocal probe, shown in the picture on the right, in a controlled way over the whole surface of the dictation belt in order to obtain the expected height map. The PC also commands the CHR by sending it a trigger signal whenever data needs to be measured. Measured data is sent from the CHR to the DAQ software which runs on the same PC as the motion controller over a serial link. As descibed in Subsection 3.1.1 the light spot generated by the bright light source goes first into the CHR and then to the confocal probe.

3.2 Measurement Processes

3.2.1 Introduction

The previous sections describe how the confocal probe is mounted onto a linear motion stage and how the dictation belt is fixed on a metallic cylinder on the rotational stage. As a matter of convention, the linear stage motion direction parallel to the dictation belt axis is referred to as the *lateral* direction. The motion direction of the rotational stage along the grooves is referred to as *temporal* or *azimuthal* direction.

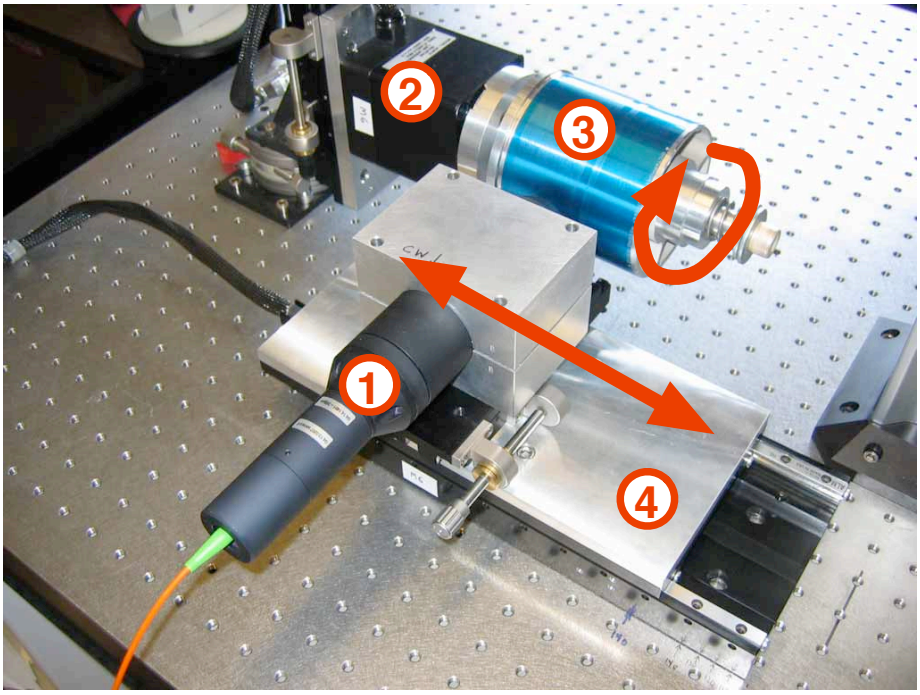


Figure 3.5: Overall setup of the measurement hardware. *1. is the confocal probe, 2. is the rotational stage and encoder, 3. is the dictation belt mounted onto a aluminum cylinder and 4. is the linear stage and encoder. The red arrows show the motion of the rotational and linear stage.*

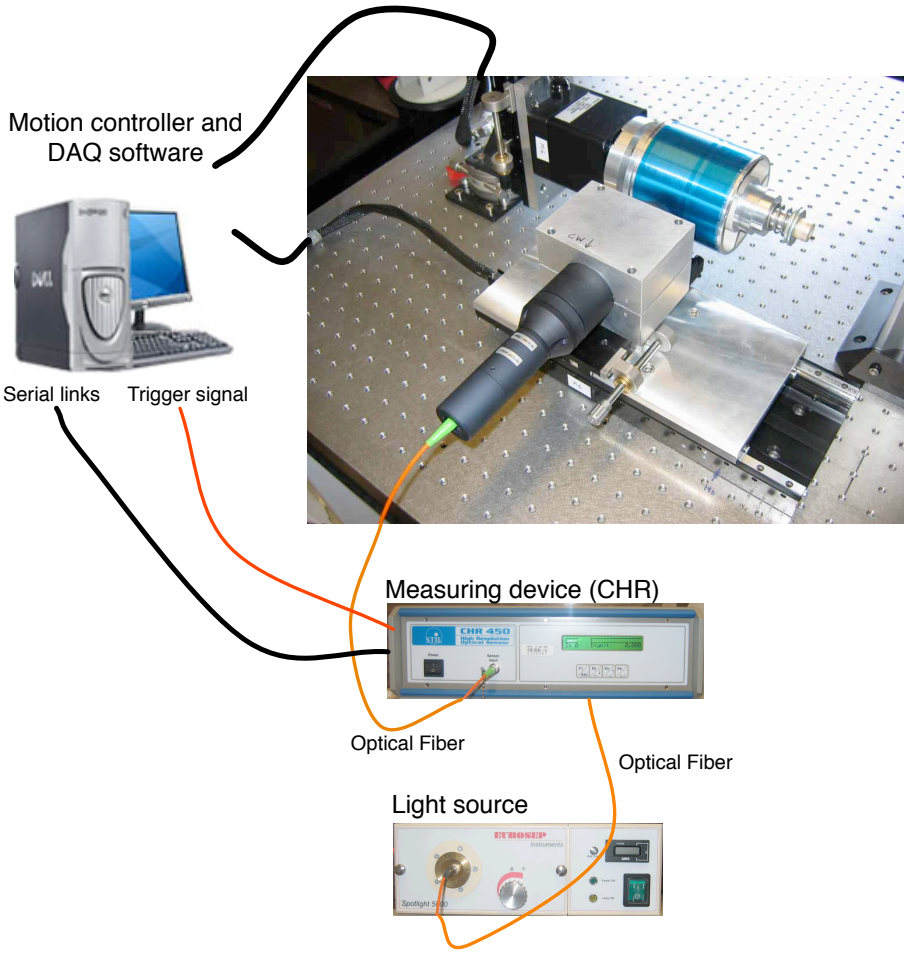


Figure 3.6: Complete picture of the hardware setup.

As a result of a dictation belt measurement, a hight map that contains heights for different lateral and temporal positions is generated. The general scanning strategy is to measure surface heights while the confocal probe is moving laterally along the dictation belt axis. After every lateral scan, the rotational stage position is incremented in azimuthal direction by a certain angle. The points acquired during a scan in the lateral direction will be called a *slice*. A partial slice was shown in Figure 3.3. The sampling interval along the azimuthal direction depends upon the needed audio sampling frequency. In the lateral direction, the sampling interval needs to be chosen such that the number of points suffices to determine the groove shape and more importantly its position.

Since the confocal probe is moving while measuring data, the sensor will average the measurement over an elongated region in the lateral direction. The size of that region over which the measurement is averaged is dependent on the spot size of the confocal probe, the velocity of the linear stage, and the sampling frequency of the probe.

For the measurement of dictation belts, a spacing of $5\mu m$ between points in the lateral direction is considered to be enough. Since the groove width is approximately $60\mu m$, thirteen measurement points can be used to approximate the groove shape. In the temporal direction, the audio sampling frequency is given by

$$f_s = \frac{num\ angular\ increments \cdot r.p.m.}{60}. \tag{3.1}$$

Dictation belts rotate at 42 rotations per minute, which is relatively slow. Because of this slow rotation speed, the number of angular steps for a whole revolution has to be accordingly high to achieve a reasonable audio sampling frequency. Different sampling frequencies are considered: $12.6kHz$, $25.2kHz$ and $50.4kHz$. The choice is dependent on the desired quality and the time available for the entire scan.

For dictation belts, the sampling rate of the confocal probe was either $1kHz$ or $2kHz$. At higher frequencies, the number of errors increases because the intensity of the reflected light is too low (see Chapter E). At lower frequencies, it would take too much time to scan a reasonable portion of the dictation belt. It takes about 50 hours to measure $5mm$ in the lateral direction (about 40 grooves) with a sampling frequency of $50.4kHz$, a lateral point spacing of $5\mu m$, and a confocal probe rate of $2kHz$. This scan corresponds to about one minute of sound. Since this scanning method is really time consuming, there is an apparent tradeoff between the sampling rate, sampling frequency, and the quality of the measurement.

3.2.2 Acquisition Software

The whole data acquisition software was implemented using LabView from National Instruments. LabView is a graphical development environment particularly designed to run measurements and to create control applications. The motion controller hardware that controls both stages comes with LabView libraries to control it. Additionally, this laboratory has a lot of experience with this tool which is why LabView is used for data acquisition in this project.

Before a measurement can be run, the user has to set a certain number of parameters:

Name	Unit	Description
ϕ_{start}	$^\circ$	Initial position of the rotational stage.
$\Delta\phi$	$^\circ$	Angular increment of the rotational stage between two slices.
$num_{\phi steps}$	1	Number of angular steps.
r_{init}	mm	Initial position for linear stage.
r_{start}	mm	Starting measurement position of linear stage.
$r_{distance}$	mm	Distance to measure along the axis.
Δr	μm	Spacing between two measurement points.
probe rate	Hz	Confocal probe rate.

These parameters are described below as they are used in the measurement process. The measurement process implemented in LabView to acquire data from dictation belts looks like this:

```
1  Read user defined parameters.
2  Write header into the output file.
3  Initialize motion control groups.
4  Configure confocal probe.
5  Move rotational axis to starting point ($\phi_{start}$).
6  Set the velocity of the linear stage ($v_{r}$).
7  Setup the output trigger on the linear stage used to
8      start measurements.
9  foreach $\phi$
10 {
11     Move linear stage to initial position($r_{init}$).
12     Put the confocal probe into
13         trigger mode (waiting for trigger).
14     Start the movement of the linear stage.
15     Read the data from the probe.
16     Write data to the file.
17     Move rotational axis by $\Delta\phi$.
18 }
19 Close output file.
```

At the beginning of the measurement process, all user-defined parameters are read. A header with the scanning parameters is then written to the output data file. See Appendix F for more details on the data acquisition file format. The motion controller then initializes the two motion axes, i.e. the linear and the rotational stage. The sampling frequency of the confocal probe is set at the value chosen by the user. Both $1000Hz$ and $2000Hz$ have been used as sampling frequencies. Once this is done, the rotational axis moves to the ϕ_{start} position. Usually ϕ_{start} is chosen to be zero. The software then sets the velocity as well as the acceleration for the linear stage. The velocity of the linear stage is given by the distance between two measurements and the confocal probe rate. It can be computed as follows:

$$v_r = probe\ rate \cdot distance\ between\ two\ measurement. \tag{3.2}$$

For a probe rate of 2000 measurements a second and a distance of $5\mu m$ between two measurements, the velocity of the linear stage would be set to

$v_r = 2 \cdot 10^3 \cdot 5 \cdot 10^{-6} = 10 \cdot 10^{-3} = 10mm/s$. After the speed has been set, the software configures the trigger that is sent to the confocal probe and tells it when to take data. The trigger is configured to fire whenever the linear stage is moving within two boundaries that are given by the starting position r_{start} and the scanning distance $r_{distance}$ along the dictation belt axis. Since the distance between two measurement points is considered as constant, the velocity of the linear stage has to be constant while the confocal probe is taking data. The linear stage therefore starts its movement at r_{init} and the measurement at r_{start}. The distance between r_{init} and r_{start} is typically chosen to be $10mm$ which is a safe distance for the linear stage to reach constant velocity. Once the trigger has been set, the main acquisition loop is entered. The main loop is repeated for the number of steps indicated by the user. At each step, the linear stage is moved to r_{init}. The confocal probe is then set into the trigger mode where it waits for an incoming signal to start measuring. When the linear stage crosses r_{start}, the trigger signal is sent to the confocal probe. As soon as the trigger signal is received, the confocal probe starts to acquire measurement points. Data is acquired from the serial port of the PC connected directly to the confocal probe and written in the SGL format to the output file. At the end of the loop, the rotational axis is moved by $\Delta\phi$ for the next slice. Once all slices have been measured, the data file is closed and the acquisition software terminates.

3.3 Discussion

The measurement range of the confocal probe is supposed to be around $330\mu m$. Unfortunately, if transparent surfaces are measured, such as the surface of dictation belts, certain wavelength are refracted. In the case of the dictation belt, wavelengths in the blue region get refracted. If the dictation belt surface is at a certain distance from the confocal probe where blue is the wavelength that is supposed to be in focus on the surface, practically no light is reflected, which leads to measurement errors. This phenomenon reduces the measurement range of the confocal probe. A small range of about $80\mu m$ has shown good measurement results. The problem is that the metallic cylinder that holds the dictation belt is not a perfect cylinder. It is difficult to keep the surface in range for the whole measurement. One solution could be to install a system similar to an auto-focus that would detect when a slice is outside the range and correct the position of the probe. An additional axis would be needed for this purpose. Another, easier way would be to scan the belt in three or four runs, stop the measurement, realign the probe when needed, and continue the measurement. In the present case, this was not too much of an issue since the belt had a relatively even surface. It could become an issue if older belts were scanned that would not be as even, for example because they were folded for storage.

Chapter 4

Data Analysis Process

The goal of the data analysis process is to extract the groove displacement from the acquired data and to eventually create its sound equivalent. As mentioned in Section 3.2, the data are acquired in slices along the dictation belt axis. The data are basically processed in the same way as the belt is scanned, i.e. slice per slice, although more than one slice might be considered simultaneously depending on the task.

Before the actual data analysis process is described, the first section of this chapter covers the quality of the acquired data. The whole data analysis process has been divided into four different steps. The first step, called *data preprocessing*, detects and corrects measurement errors that occur during data acquisition. It also corrects some of the geometrical distortion the dictation belt might have. The second step of the data analysis process detects the groove position and extracts its movement in time. Different groove tracking approaches are considered and compared. The extracted groove positions have to be linked together in order to form a signal sequence in time, which is also part of the *groove detection* step. In the third step, called *data post-processing*, error detection and correction is applied to the signal to remove clicks and lower noise. In the last step of the data analysis process the signal is filtered, resampled, scaled and written into a *wav* file. The complete data analysis process is illustrated in Figure 4.1.

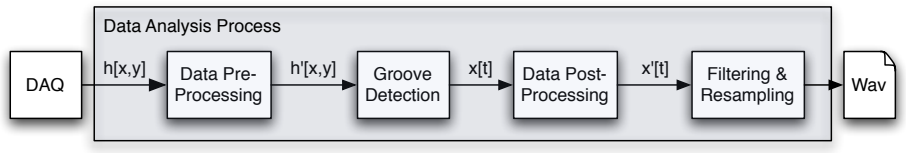


Figure 4.1: Complete data analysis process.

4.1 Data Quality Check

A first data quality check was done by implementing a preliminary version of the data analysis process in LabView. This tool validated the measurement process

and showed that the quality of the data was at least reasonable since a male voice present on the output sound file could easily be understood. This result suggested that there were no major quality issues either in the measurement process or in the data analysis process.

In order to monitor and measure the performance of the data analysis process, it is crucial to determine the various sources of error and noise in the measurement process. Four sources of errors are considered: The confocal probe, the linear motion stage, the rotational stage, and the synchronization trigger.

4.1.1 Confocal probe

Repeated measurements of the same location on a dictation belt were conducted at different probe rates. For these measurements the linear stage as well as the rotational stage were not in motion. In this case, the only noise source is the confocal probe. Different measurements show that the noise level increases with the sampling rate of the confocal probe and also that it is lower on flat surfaces compared to surfaces with a certain slope. See Section E.1 for details on these measurements. The noise level is typically around $0.13\mu m$ for measurements using a probe rate of 2000 samples per second. Compared to the groove depth, which is around $3\mu m$, the noise represents about 4.33%. This is quite a lot. Some data analysis approaches described later in this chapter fit a groove shape to the data in order to lower the effect of this noise.

Measurement errors are usually isolated and can be easily detected and corrected since they do not follow the expected shape of the groove. They usually are off by a lot, which makes the detection even easier. Section 4.2 discusses in detail how such errors are detected and corrected.

4.1.2 Linear motion stage

The confocal probe takes data at a constant rate while the linear motion stage moves with a constant velocity over the dictation belt. Measurement points are therefore assumed to be equally distant from each other. If the velocity of the linear motor is not perfectly constant, some errors would appear in the data. The linear motion stage noise has not been measured separately. However a measurement was conducted that shows the total noise introduced by the confocal probe, the trigger and the linear motion stage combined. At a probe rate of 2000 samples per second, the standard deviation of the measurement points is around $0.3\mu m$, which is more than double the noise measured without movement. Compared to the groove depth, which is around $3\mu m$, the noise represents about 10%! For more information on this measurement, see Section E.2. Again by fitting the data using least squares fit, the effect of noise on the resulting groove position is reduced.

4.1.3 Trigger

The so-called trigger is the signal that is generated by the motion controller to tell the confocal probe when to measure data. This signal is sent whenever the linear motion stage crosses a certain position. If this signal were not very precise in time, the confocal probe would not start the measurement at the same position for each slice. This would introduce a lateral offset between slices.

4.2 Data Preprocessing

The data aquired by the 3D scanner needs to be corrected before a groove extraction algorithm can be applied to it because it contains some errors that could cause the algorithm to work improperly. Some errors are due to scanner malfunctions (measurement errors) whereas others come from the shape of the belt itself and support cylinder misalignment (warpage). This section explains some techniques used to fix these problems before an analysis algorithm is invoked.

4.2.1 Measurement errors

On the whole scanned surface it may happen, that some points were badly acquired because the light returned to the confocal probe was either too weak or too bright, depending on the actual material on the point (dust) and the point's normal orientation. This situation often results in a point whose height is completely off. In most cases the height of these error points is either arround 0 or arround $330\mu m$, which is around the maximal range of the confocal probe. Some of them are less off but still remain obvious.

Detecting Measurement Errors

These badly measured points are easy to detect because they are usually isolated and have heights that are very different from the correct points. The interpolation provides a good error correction as long as only good points are taken as the base of the interpolation. There are several ways to decide if a point is good or not. The following algorithm explains one of them:

```
1  Take a slice.
2  Consider all points of this slice as bad.
3  x = a given threshold.
4  foreach point in this slice
5  {
6      if the height differences between all the
7      contiguous points in a given window arround
8      the current point are all smaller than a given
9      threshold
10     {
11         Mark the current point as a good point.
12     }
13 }
14 foreach point in this slice from left to right
15 {
16     if the current point is good
17     and the height difference between itself and
18     the next point is smaller than a given threshold
19     {
20         Mark the next point as good.
21     }
22 }
23 foreach point in this slice from right to left
24 {
25     if the current point is good
26     and the height difference between itselft and
27     the previous point is smaller than a given threshold
28     {
29         Mark the previous point as good.
30     }
```

```
31  }
```

One of the main advantages of using a window is that a group of contiguous badly measured points that are almost at the same height will not be considered as good points. Another technique is to fit a line across the slice and reject all the point that are too far from the line, but this method has some disadvantages. First of all, the line fit will be erroneous because of the actual bad points, and second of all it will not fix, in opposition to the former method, the bad points inside the grooves because they will be considered as close enough to the line.

Fixing Measurement Errors

As mentionned in the previous section, the interpolation is a good way to correct measurement errors. To determine points used to construct the interpolation, the patterns shown in Figure 4.2 are used. The center square represents the point to interpolate, the grey ones the points to use for the interpolation, and the white ones are not used. In other words, to use a pattern all the grey squares must be good points. The preferred pattern is the number 1 and the least adequate one is the number 14. This means that the first pattern will be used whenever possible, and if it is not, the second will be tried and so on. The interpolation in the x direction is preferred for the dictation belts because the distance between points in x is smaller than the distance between points in the other direction, unless more than 60000 line scans around the belt are taken. The first four patterns use 4 aligned points. A 3rd order polynomial that crosses exactly these points is computed and thus the missing point can be easily determined. The four next patterns use 2 aligned points and the missing point is computed by a linear interpolation. The last four patterns also use a linear interpolation to find the missing point. This method could be improved using a third point.

Once a point is recovered it stays marked as a bad point to avoid its use as the basis for another interpolation, which could potentially introduce additional errors. In some cases, it is impossible to fix a measurement error without basing the interpolation on previously interpolated points. In this case the correction algorithm might be applied several times.

4.2.2 Warpage

Warpage is the fact that the scanned surface of the medium is not perfectly flat. It can either be caused by the fact that the scanner is not perfectly calibrated or because of the actual warpage of the scanned medium.

In the case of the dictation belts with the 3D scanner, both of these warpage phenomena appear. The confocal probe moves in the x direction to acquire a line of points. If the dictation belt does not have exactly the same angle as the confocal probe's linear axis then the average distance that the confocal probe will measure will be either shorter and shorter or longer and longer along the the x axis. This will result in the addition of a line to the data as shown in the following equation.

$$f'(x, y) \quad = \quad f(x, y) + ax + b \tag{4.1}$$

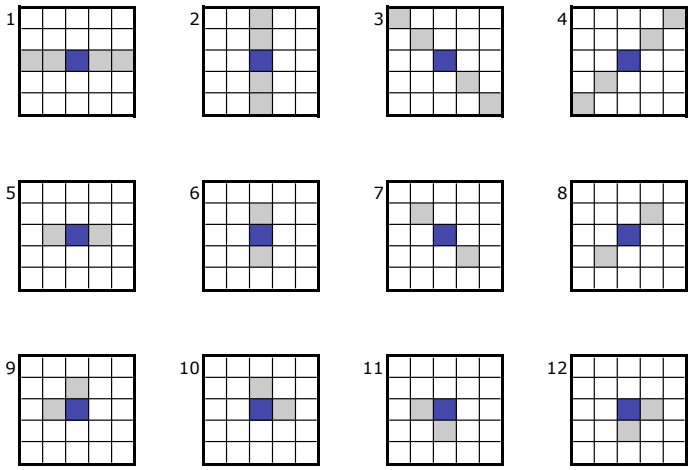


Figure 4.2: Interpolation patterns

The dictation belts have also their own physical warpage, especially when they were folded and stored in a folder. This warpage also has an influence on the distance between the belt and the confocal probe. The forces applied on the belt to put it on the scanning cylinder as well as the belt curvatures may result in lateral warpage, which is actually the only warpage that will be manifested into the sound. This warpage will cause the addition of low frequencies, which can then be easily filtered (see Section 4.4). Because the groove is moving laterally, the sound will not suffer from the vertical low frequency modulations.

Actually correcting the vertical warpage will lead to more robust and easily implementable algorithms. For this reason, the surface should be "normalized". This means that it should be flat at a large scale. To do this, for each slice in the x direction a line is fitted using the least squares method and this line is subtracted from the slice. The line works well for short scanning distances but should be replaced by some higher order polynomials for longer scanning distances.

4.3 Groove Detection

The groove detection algorithm iterates through the acquired data slices. Each slice contains a number of grooves, depending on the measured width of the dictation belt. For each slice, the algorithm detects the position of each groove it contains and links the groove positions of the current slice with the ones of the previous slice in order to create a single groove. Different methods were used to identify the groove position, but all use the groove bottom, i.e. the lowest point in the groove, to represent its position. Note that this point is not necessarily part of the data set but rather calculated by fitting a groove shape to the measurement points.

Since the groove shape on the dictation belts is not very distinct (see Figure 3.3), a robust method had to be chosen to detect interesting regions that contain a groove. Once these regions have been defined, a first approximation of the groove bottom position within each region is computed. The groove position is then calculated as accurately as possible given the first approximation.

The algorithm used to extract the groove movement can be described as follows:

```
1  foreach Slice
2  {
3    Locate interesting regions that contain a groove.
4    foreach Region
5    {
6      Approximate the groove bottom location.
7      Compute groove bottom position.
8    }
9    Link groove bottoms of current slice with previous slice.
10 }
```

Different approaches were investigated to compute the best possible groove bottom position. All of them fit a certain function, which is a model of the stylus geometry, to the neighbor data points of the first approximation. The next subsections describe these different techniques.

4.3.1 Fit Parabola

The first idea is to fit a second degree order polynomial to the neighbor points of the first approximation. In order to do that the least squares fit algorithm is used. Although the groove shape is closer to a circle than a parabola, the fitting operation is much simpler to compute for a parabola than it is for a circle.

If the function is given by $y(x) = ax^2 + bx + c$ the x position of the minima is located at $\frac{-b}{2a}$.

More information on least squares fit can be found in [6].

Constant Opening Factor

Another approach is to fit a parabola with a constant factor for the x^2 term. If the parabola equation is given by $y(x) = ax^2 + bx + c$, a is chosen to be constant. Since the shape of the recording needle is probably not changing over time, this might be a promising way to model the shape of the groove. The constant value for a was averaged over the whole data set by fitting a regular parabola in the first place. If the a factor is constant the equation becomes a first order polynomial, which is given by $y'(x) = bx + c$, where $y'(x)$ is obtained by subtracting ax^2 to each value of $y(x)$.

Minimize Distance in X

The general least square fit computes analytically the minimum of the function given by:

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - f(x_i)}{\sigma_i} \right]^2. \tag{4.2}$$

Since the measurement errors (standard deviation) σ_i are not known, they are set to be a constant value: $\sigma_i = 1$. N is considered to be the number of measurement points to fit. y_i is the ith measurement point and $f(x_i)$ is the function value for a certain value of x_i. $f(x_i)$ is given by $ax^2 + bx + c$ where a, b and c are the parameters to optimize. The least square method minimizes the sum of the squares of the distance in y between each measurement point and the parabola.

Remember that the groove has a lateral modulation. The goal is to find the best possible approximation in x to represent the groove position. The least square fit method finds the best parabola by minimizing distances in y. By setting the coefficient of x^2 to be a constant the axes of the linear function can be inverted ($y \Leftrightarrow x$). By doing this the least square fit minimizes the distance in x between the measurement points and the function.

4.3.2 Fit Circle Shape

Pictures of the needle show that its shape is like a circle. This approach considers the use of nonlinear fitting methods to fit the best circle with constant radius to the measurement points. The parameters to optimize are the coordinates of the center of the circle. Each y value is then computed like that:

$$y = p_y - \sqrt{r^2 - (p_x - x)^2}. \tag{4.3}$$

Where (p_x, p_y) is the center of the groove and r is the radius of the circle.

4.3.3 Repeated Fit

To compute the best approximation of the groove bottom position only one fit might not be enough. Here different ideas are considered to compute several fits and choose the best one.

Variable Window Width

Previously the number of points to consider in the fit was constant. If the window width was 11 for example, five points on each side of the first approximation together with the first approximation were considered to be part of the fit. Here several window sizes are considered. If the size of the window is even, two possibilities exist to place the window around a center point. Both were computed. The question arises how to compare those fits. The mean squared error can not be used in this case since the number of points considered in the fit is variable. A fit with less measurement points is very likely to have a lower mean squared error than a fit with more points. The χ^2 distribution could be used to compare two fits.

Variable Window Position

Instead of changing the window size as described earlier, the window position of the measurement points could be moved to different position in order to find the best fit. Typically the first approximation is considered to be the center of the measurement points. Here the window was moved from -3 points to $+3$ points. For example if the first approximation is 125 and the window size is 11, the considered point intervals would be: $[117127], [118128], ..., [120130], ..., [124133]$. The best fit was considered to be the fit with the smallest mean squared error. The mean squared error of a fit is given by:

$$mse = \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i))^2, \tag{4.4}$$

which is essentially the sum of squares of the distance between the measurement points and the fitted function.

Remove Bad Points

Another idea is to remove the bad points from the considered points and recompute the fit. A point is considered to be bad if its distance to the fitted curve is too large. Points that do not follow the usual groove shape, i.e. measurement points of dust, would be removed. In order to decide if a point is too far away from the fitted curve the standard deviation of the distance point to curve was measured for 10000 fits and a window size of 11. The standard deviation is $\sigma = 0.2\mu m$. If the distances from the points to the curve are considered to be normally distributed, then 68% of the points will be within one standard deviations away from the curve, 95% of the points will be two standard deviations away and 99.7% will be within 3 standard deviations. The factor used as a

threshold needs to be tuned to exclude the bad points and keep the good ones. A minimum number of points was set in order to avoid fits with too few points.

Note that the standard deviation appears to be the same for all the 11 points, which suggests that the chosen window size is certainly not too large.

4.4 Data Post-Processing

The result of the groove detection step is essentially a signal that contains the groove position for each time step with the same sampling frequency that was used during the measurement process. It basically corresponds to the amplitude over time. Before the data is filtered and resampled some errors are removed. In the data preprocessing step measurement errors have been removed and corrected. The goal of the data post-processing is to correct errors in the signal that occur due to dust or scratches. Two kinds of error correction have been considered here.

4.4.1 Silence Generation

When the signal is very weak it is most likely noise. The silence generation replaces this noise by real silence. Actually dictation belts were not used to store music but speech so the risk of losing some interesting information is quite low, altough it still exists. To do that silence generation, the analyzer splits the wave in chunks of $20ms$, which can still contain the lower expected frequency on the dictation belt ($50Hz$). Then the signal's mean sum of absolute deviation from the signal mean value is computed for each of these blocks. If more than a given number of consecutive blocks have an energy lower than a given threshold, they are replaced by silence. This technique helps in most of the cases to get a waveform closer than the actual recorded one but the result does not sound really good for the human ear because of the noise is no more continuous.

4.4.2 Z Shift Detection

As the groove on the dictabelt is modulated in the width and not in the depth, the height (Z) of the extracted groove should stay constant or at least vary slowly. The Z shift detection algorithm finds peeks in the Z component of the groove and consider the corresponding points of the groove as bad. To find the bad point, the same algorithm as discussed in Section 4.2.1 is applied. A linear interpolation on their X component is then applied to recover them. This algorithm actually removes some clicks due to dusts on the dictation belt.

4.5 Filtering and Resampling

Signal processing operations are applied to the reconstructed audio signal before it is outputted to a Wav file.

4.5.1 Resampling

In digital audio several sampling rates are commonly used, among them: $11.025kHz$, $22.050kHz$, $44.1kHz$. The bandwidth of the dictation belt recorder is between

$3kHz$ and $4kHz$ starting at $100Hz$ to $3.5kHz$, the resulting Wav file is stored using a sampling frequency of $11.025kHz$. According to the Nyquist sampling theorem, the sampling frequency of the resulting Wav file must be greater than twice the bandwidth of the reconstructed signal, in order to avoid aliasing. Therefore a sampling frequency of $11.025kHz$ is enough.

Resampling in cases where the ratio between the two sampling frequency is given by a rational number $\frac{L}{M}$ is typically done by up-sampling the signal by a factor L and then down-sampling it by a factor M, where L and M are positive integer values [5], [4]. The up-sampling process is called interpolation and the down-sampling process is called decimation. The interpolation operation inserts $L-1$ equidistant zero-valued samples between each consecutive samples of an input sequence $x[n]$:

$$y[n] = \begin{cases} x[n/L], & n = 0, \pm L, \pm 2L, ... \\ 0, & otherwise. \end{cases}$$

The output $y[n]$ is then passed though a low-pass filter that cuts off the spectral components higher than $\frac{1}{2}L$. The decimation operation simply keeps every Mth sample of the input sequence $x[n]$:

$$y[n] = x[nM].$$

In order to avoid aliasing the signal $x[n]$ has to first go though a low-pass filter that cuts all frequency components higher than $\frac{1}{2}M$.

Data was acquired at different sampling frequencies: $12.6kHz$, $25.2kHz$ and $50.4kHz$ (see Chapter 3 for more details). Considering for example a sampling frequency of $50.4kHz$ for the input signal and $11.025kHz$ for the resulting waveform, then the ratio between the two sampling frequencies would be given by $\frac{11025}{50400} = \frac{7}{32}$. The input signal would be up-sampled by a factor of 7 and down-sampled by a factor of 32 to reach the desired sampling frequency of $11.025kHz$.

For details on the implementation of the resampling operation, please refer to Section 6.3.

Chapter 5

Study of a Dictation Belt Recorder

5.1 Sound Samples Recording

Several dictation belts with tests patterns were recorded on blank belts with the dictation belt recorder. The external microphone input is used to feed the signal generated by the computer into the dictation belt machine. Sound patterns were generated with LabView and converted to analog signals with the high-accurarcy data acquisition card (NI PXI 4461). The typical sound pattern is a series of eight sine waves at different frequencies going from $50Hz$ to $6.4kHz$. The duration of each frequency is typically two seconds. At each frequency change silence is generated for half a second. The frequency is doubled at each frequency change ($50Hz$, $100Hz$, $200Hz$, ..., $6.4kHz$).

Having these known sound samples it is now possible to compare the extracted sound wave with the generated wave and see how the dictation belt machine affects the sound.

5.2 Apparatus

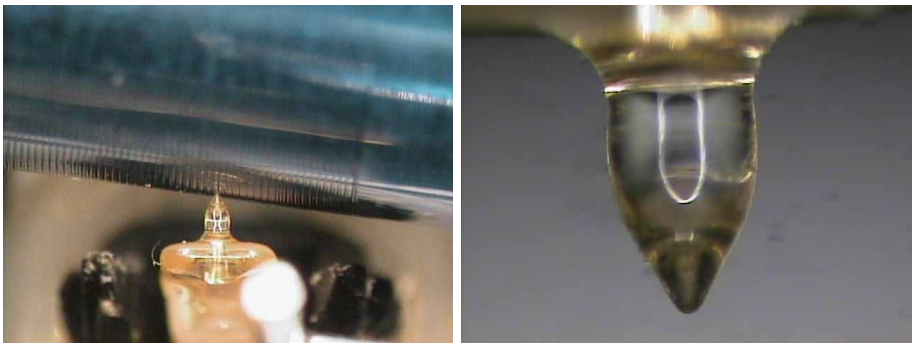
The dictation belt machine used here is probably the last generation of dictation belt machines produced. It already uses transistors as electronic components as opposed to older versions that used tubes. Dictation belts can be played back as well as recorded. The machine has two different pickups to do that: a crystal pickup for playback and a sapphire magnetic pickup for recording.

5.2.1 Playback Pickup

The playback pickup is a crystal pickup that outputs a voltage that is proportional to the needle displacement, i.e. the groove movement. Figure 5.1 shows two pictures of the playback needle. On the left, the needle is shown in a groove. In is interesting to notice that only a tiny part of the needle is actually in the groove. The second picture on the right shows a close-up of the playback needle.

The dictation belt machine was modified in order to read the signal across the pickup. Figure 5.2 shows where the signal was measured. The whole schematics

can be seen in Chapter H. Note that the rest of the electronics was de-connected while measuring the signal across the needle to avoid any perturbation.



(a) (b)

Figure 5.1: Dictation belt machine playback needle *(a) Shows the needle in a groove. (b) Is a close-up on the playback needle.*

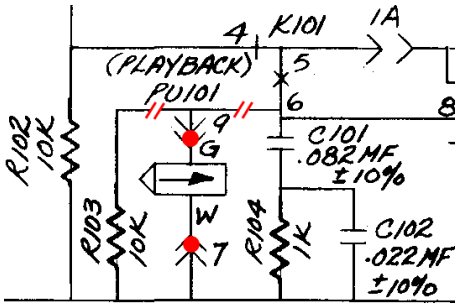


Figure 5.2: Schematic of playback needle connection.

5.2.2 Recording Pickup

The recording pickup is magnetic. It has a sapphire needle, which embosses the sound signal into the soft material. The magnetic pickup shown in Figure 5.3 can be modeled as a damped harmonic oscillator. At very low frequency, the position of the needle is proportional to its driving force, which is the input voltage and at higher frequency, the position of the needle is given by the integral of the input voltage. The implications of this is discussed later.

The dictation belt recorder was modified to also read the signal of the recording needle. Figure 5.4 shows where the signal was measured. The whole schematics can be seen in Chapter H.

A picture of the recording needle is shown in Figure 5.5. On the left you can see the recording needle and partially the metal piece that stays in between the magnets and in the coil. On the right its a close-up on the recording needle. Its

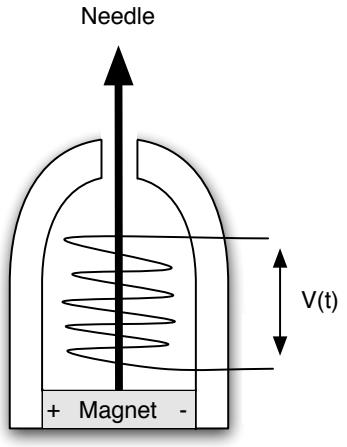


Figure 5.3: Schema of the recording needle.

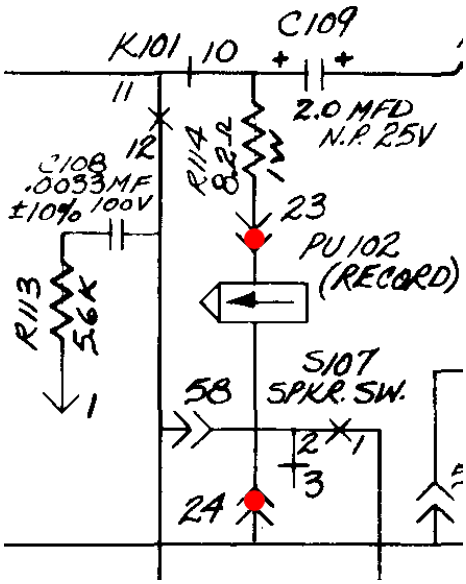
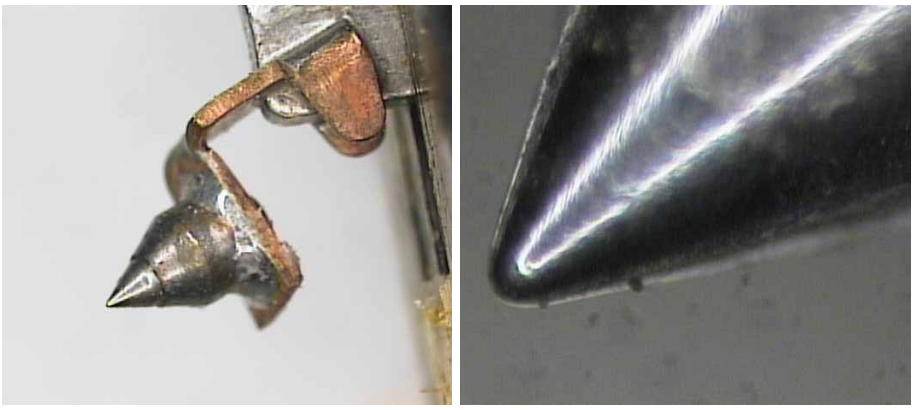


Figure 5.4: Schematic of the recording needle connection.

not possible to picture the needle in a groove since the needle is berried under the dictation belt player.



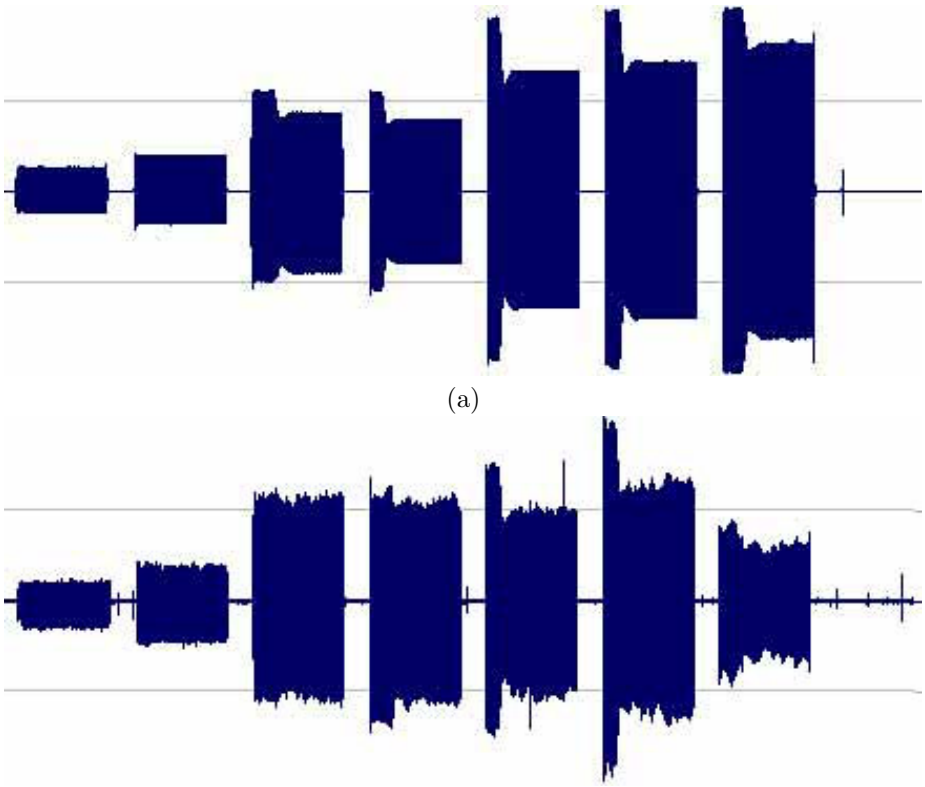
(a) (b)

Figure 5.5: Dictation belt machine recording needle *(a) Recording needle. (b) Is a close-up on the recording needle.*

As mentioned earlier, the movement of the recording needle is given by the integral of the input voltage. If the input is given by $sin(\omega t)$, the signal recorded would be $-\frac{1}{\omega} \cdot cos(\omega t)$. Hence higher frequencies will be more attenuated than lower frequencies. This is indeed what is measured. Figure 5.6 shows on the top a series of seven sine waves that have higher and higher frequencies. This is the typical test sound pattern described earlier. This signal was measured across the recording needle. The electronics of the dictation belt machine apparently boosts high frequencies before the signal is recorded, since the input signal (not shown here) had the same amplitude for all the frequencies. The image at the bottom shows the exact same signal measured across the playback needle. One can see that the higher frequencies where attenuated more than the lower ones.

5.2.3 Speaker Output

A connector to the speaker output was also added in order to measure the signal after the electronic circuit. To have the same impedance as the speaker a small resistor of 8Ω was connected at the output.



(a)

(b)

Figure 5.6: Comparison of input and output signal of the dictation belt. *(a) Sine waves at different frequencies measured across the recording needle. (b) The same sine waves measured at the playback needle.*

Chapter 6

Software

6.1 Analysis

6.1.1 Needs

In the mind of going ahead with the sound extraction of the different media, LabView cannot be used anymore. Its biggest advantages are the "easy get in" and the huge library it provides. Unfortunately since it is a graphical programming language, a lot of time is spent to arrange blocks and wire them together. The nested structures really cause a big problem when the programmer needs to add some blocks into a frame that is already full. In order to resize the frame he has to first resize the frame that contains this frame and so on, and then rearrange all the wires. Actually the waste of time for small modifications is completely disproportionate.

That is why there is a need to create a "regular" program that will be able to extract the sound from the scanned surface of a dictation belt. The sofware should be as easy as possible extendable to extract sound from other types of mechanical media. It has to provide a graphical user interface that shows the surface in a 3D view as well as in a 2D view. The user must be able to see what the analyzer is doing in order to debug it. He can start it, pause it, resume it and stop it. The software must be able to be command line runnable in order to use it from a batch file.

6.1.2 Programming language

For the realization of the software five programming languages where taken in consideration: C, C++, Microsoft.NET languages, Java and Microsoft Visual Basic 6, which are currently the mostly used programming languages.

For the code to be reusable and well-structured, the language should be object oriented. C and Microsoft Visual Basic 6 are not object oriented and can therefore not be used.

The next step is to decide to either use a managed or an unmanaged programming language. C++ is unmanaged, Java as well as Microsoft.NET languages are. The biggest advantage of C++ in comparison to managed programming languages is its speed of execution because it has not to be interpreted by a virtual machine at runtime. On the other hand the programmer must manage

itself memory allocations. Furthermore, C++ is an old language and is therefore unpleasing; for example the coder has to write function prototypes to make the compiler aware that some functions exists some place further in the code. As no special requirements are needed concerning the execution speed of the program, a managed programming language will be used.

At this point either Java or a Microsoft.NET language has to be chosen. The biggest Java advantage is its portability. The Java Virtual Machine has been released for all the most common operating systems, which is not the case for the Microsoft.NET Framework that exists currently only for Microsoft Windows. But Microsoft.NET languages have the advantage to be more advanced than Java. Furthermore, Microsoft Visual Studio provides a great integrated development environment which helps saving a lot of time when creating user interfaces. As long as all the computer used for this project are running Microsoft Windows XP and that there was no particular requirement concerning the portability of the code, and that a user interface has to be designed, a Microsoft.NET language will be used.

The Microsoft.NET framework supports a lot of programming languages (over 25). Each part of the same project can be written in a different language. For this project, C# will be preferred because of its syntax which is a really good mix between C++ and Java.

6.1.3 Graphic API

The program should be able to display the surface of as smoothly as possible, which involves the best possible frame rate, when the user is moving the camera. Because of the resolution of the surface, which is about 152 millions of triangles per square inch, an acceptable frame rate can only be reached by using a graphic application programming interface that provides a direct access to the graphic card.

Today there are two widespread graphic libraries that allow direct access to the hardware. Those are Microsoft DirectX and OpenGL. The former is property of Microsoft Corporation and runs only under Microsoft Windows while the latter is an open source library that runs on most of the operating systems. Both are free. As long as the program will run under Microsoft Windows the restricted compatibility of Microsoft DirectX is not a problem. Since Microsoft Visual Studio is used, using Microsoft DirectX will lead in a better IDE integration, especially for debugging. That is why this one will be used.

6.1.4 Math and DSP library

As LabView from National Instrument was previously used, the use of Measurement Studio from the same company is very appropriated. Measurement Studio is almost the same library as LabView uses, written for Visual Studio.NET.

6.1.5 Input

The software has to be able to read the file created by the acquisition program. The input file format is described in the section F.

6.1.6 Output

The program may produce different kinds of output depending on the used analyzer. At least one of the analyzer has to be able to produce a wave file (.wav). Please refer to the different analyzers descriptions in Appendix 4.2.1 to learn more about the file they create.

6.2 Design

6.2.1 Architecture

There are three big parts in this software: the one that provide access to the measured data, the user interface, and the analyzer. As each of these parts is subject to future evolution, they are each implementing a dedicated interface. This allows Groovster to use as measurement accessor, analyzer, resp. graphical user interface any class that implements one of those interfaces. The Figure 6.1 shows a simplified class diagram that should give a good idea of the software architecture.

To make Groovster as flexible as possible, the code classes that implements these interfaces is compiled to some shared libraries, which allows dynamic class loading. That means that there is no need to modify the existing code to add new features. The class diagrams for the existing Groovster's analyzers, measurement accessors and graphical user interfaces can be found in Appendix B.

At the program startup, a window shows all the implementations found for each part and allows to choose which will be used to run the program, as shown in Figure 6.2.

6.2.2 Concurrent Programming in C#

As most of the programming languages, C# offers threading and synchronisation techniques. They are several ways for synchronising resources as explained in [10] but this program only uses a simple one called "monitors". The C#'s threading library owns the class Monitor, which exposes some static methods for monitored synchronization. The software uses actually only the following ones, which are explained in the following sections.

- Monitor.Enter(object)
- Monitor.Exit(object)
- Monitor.Wait(object)
- Monitor.Pulse(object)

Monitor.Enter

The static method Monitor.Enter(object) locks the object passed as parameter. If the object is already locked by another object, the thread that called the method is blocked until it can get the lock.

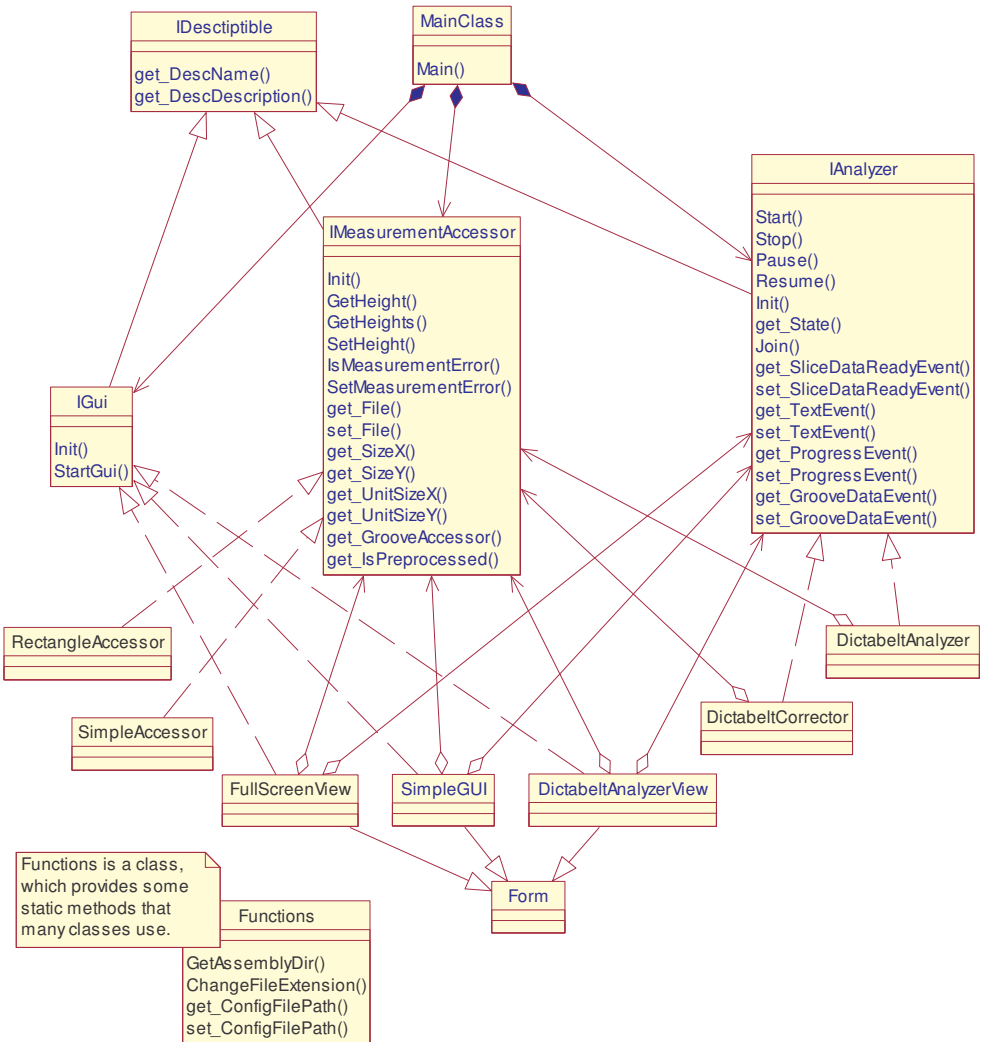


Figure 6.1: Architecture overview

Monitor.Exit

The static method Monitor.Exit(object) unlocks the object passed as parameter. If some other threads were waiting for the the lock, one of them can then grab it.

Monitor.Wait

The static method Monitor.Wait(object) makes the calling thread release the lock on the given object and wait until it is explicitly pulsed by another thread. Then it tries to reacquire the lock on that object. If it cannot it will wait to be pulsed again. See the Monitor.Pulse method for more information. The Wait method can also be used with a condition as parameter. If the waiting thread is woken up and the condition is not satisfied, it is going to wait again.

Monitor.Pulse

The static method Monitor.Pulse(object) makes the first thread in the waiting queue to be woken up and get the lock on that object. There is also a method PulseAll that wakes up all the waiting thread, but only one is going to be able to take the lock. The other one are going to wait again. This is useful when the method Wait is used with a condition. See the Monitor.Wait method for more information.

The "lock" statement

The C# programming language offers a syntactic sugar which is the "lock" statement used to lock an object.

```
\begin{verbatim}
lock(mySharedObject)
{
    // Synchronized code
}
\end{verbatim}
```

The lock statement tries to acquire the lock on the object given as parameter. If the object is already locked by another thread, the calling thread is going to wait for the lock on that object to be released. Once it got the lock it enters the synchronized region and releases the lock once it finished it. If an uncatched exception occurs in the synchronized region the lock will be automatically released. The lock statement is equivalent to the following code:

```
\begin{verbatim}
Monitor.Enter(mySharedObject);
try
{
    // Synchronized code
}
finally
{
    Monitor.Exit(mySharedObject);
}
\end{verbatim}
```

6.2.3 3D view of the Surface

DirectX

Microsoft DirectX is a application programming interface designed especially for game programming and multimedia applications. It is made of different modules which handle graphic output, sound recording and outputting, networking and input devices such as keyboards, mice, joysticks. Most of them provide direct access to the hardware, which leads to a huge performances increase.

In this project only Direct3D a part of DirectGraphics, which is the graphics output module, is used. Graphic objects in Direct3D are conceptually a couple of triangles put together. The triangle is the only 3D surface that Direct3D is able to draw. Objects are composed of a vertex buffer and optionally an index buffer. Vertices are 3D points with some other properties such as color, coordinates in texture or normal vector. The index buffer represents how the triangles have to be spanned over the vertices. Once an object has to be drawn, its vertices and indices are sent to the 3D pipeline, which will process them and render the 3D object to a 2D surface, which is the screen. See Section C.1 for more information about the Direct3D 3D pipeline.

Medium's Sufrace 3D Model

The easiest way to create the 3D model of the medium's surface from its height map is to span two triangles for each four adjacent heights as shown in the Figure 6.3.

Most of nowadays graphic cards are not able to handle objects that contains more than about 20000 triangles. That is why media surface, which may contain more than $1 billion$ of triangles, is split in many rectangles of less than 20000 triangles. This also allows to load only visible rectangles from the input file. The detection of visible and not visible objects are discussed in Section 6.2.3.

The models have to be lighted in order to make the user easier to identify the peeks and the valleys. The Figure 6.4 shows that it is very difficult to see the relief if the surface is not lighted. A directional light is used because it is best, which produce shading effects, in terms of computation time. Direct3D applies the lighting on vertices, thus any triangle has the lighting effects of its three vertices.

The lighting force of each vertex is calculated by computing the dot product of the vertex' normal and the light source's inverted direction vector. Positive values and 0 are considered as unlighted while 1 corresponds the the maximum lighting. The lighting of each vertices is calculated at each rendering because both light or vertices are susceptible to move from one frame to the next one.

Culling

Each 3D triangle that has to be rendered consume time for its computation. The culling operations consist in reducing the number of triangle to be drawn. In the section some techniques used in this project that allow to reduce the number of triangles to render are explained.

One of the easier culling technique to is the face culling because it is supported by DirectX. It consists in making one face of each triangle invisible. For example if you want to draw a cube, you will set all the faces that are inside the

cube to be invisible, assuming that the cube is never going to be observed from inside of it. This technique basically divides the number of triangles to draw by two.

Altough the face culling decreases considerably the number of triangle to be rendered, it does unfortunately not avoid those triangles to cross a part of the 3D pipeline because the positions of their vertices on the 2D screen are used to determine which face of the triangle is being seen.

A very good improvement that would avoid a lot of triangle to go to the 3D pipeline is called the object culling. This technique consists in testing if an object is visible or not from the viewing point before sending it to the 3D pipeline.

It is useful to define geometrically the portion of the scene that the user is seeing in order to avoid sending outer objects that will finally not be displayed to the 3D pipeline. This visible volume is called the viewing frustum, which is delimited by six planes: the left plane, the right plane, the top plane, the bottom plane, the near plane and the far plane (also known as far and near clipping plane). The left, right, top and bottom planes include respectively the left, right, top and bottom borders of the screen. These four points contains all a point called the eye. The left and the right planes intersect at the eye with an angle of 45. The near and the far planes are parallel to the screen. Their goal is to avoid objects that are to far or to close to be drawn (computation time saving).

To check an object is outside the viewing frustum or not, instead of testing the visibility of each of its composing triangles, the objects are placed in a so called axis aligned bounding box, which simplify considerably calculations (computation time saving). The axis aligned bounding box is the smallest parallelepiped that can fully contain the object and that is aligned with the 3 unit vectors (x, y and z). Please see Appendix C.2 for more details about the detection of objects in frustum.

Scene Organization

The code that is responsible for determining whether an object is visible or not has to check the visibility of all the objects on by one (O(n))and this is done for each step of the camera movement. This sadly decreases the frame rate of the 3D visualization.

A good techniques to decrease this selection time is to organize the objects of the scene in an octree. The octree is to a 3D scene what the quadtree is to a 2D scene. Due to the two dimensional nature of the paper on which this report is written, an explanation of the quadtree will be much more clearer.

Using a quadtree in a 2D scene consists in considering the scene as one big cell which contains 4 subcells. Each of these subcells contain 4 subcells and so on until a given nesting level is reached (see Figure 6.5). Each object of the scene is placed in the smallest cell that can fully contain it (see Figure 6.6). Once the program has to decide which objects are visible, it start from the biggest cell and check its visibility. There can be three cases:

- Case 1: The cell is fully visible. All the objects contained in it, all subcells included, will be visible.

- Case 2: The cell is not visible. All the objects contained in it, all subcells included, will not be visible.
- Case 3: The cell is partially visible. Its object and its 4 cells have to be tested for visibility. Its cells will also lead to one of these three cases, so the operation will be done on each of the subcells as if they were the biggest cell taken as example.

The quadtree offers to determine objects visibility in an O(ln n) expected complexity. They provide also non negligible advantages in object collision detection that will not be discussed in this report. The octree works exactly the same way as the quadtree except that a cell is a volume that is split 8 smaller cells.

Vertices optimization

As discussed in sectio 6.2.3 the model has to be lighted. In order to make this possible, it has to contain a normal vector on each of its vertices. The easiest way to set up the normal vectors is to store the vertex buffer of the model in the DirectX's class "Mesh" and call the method "ComputeNormals" on it, which will cause all the normal vectors to be automatically computed considering all the faces each vertex is contributing to.

When a vertex buffer is created it basically opens a data stream to the graphic card that allow to write the vertices directly into its memory as if the programmer were writing into a file. The problem is that transfering data to the graphic board takes time and the internal operation of "ComputeNormals" is to load the vertex buffer and the index buffer of the model from the graphic card into the system memory, computes the normals and send the vertex buffer back to the graphic card.

The time needed to load a rectangle of 20000 triangles was about $400ms$ of which $320ms$ (80%) were spent on loading the model to the graphic card and computing the normals. This delay is too long especially when the program has to load hundreds of such rectangles, that is why some optimizations have to be done. Let n be the number of triangles, then the size of one vertex, the size of one index, the size of the vertex buffer, the size of the index buffer, and the total size are given by the following equations.

$$\begin{aligned}
VertexSize &= sizeof(float) * (NbCoordinatesPos + \\
&\quad NbCoordinatesNormal) + sizeof(Color) \\
&= 4bytes * (3+3) + 4bytes \\
&= 28bytes && (6.1) \\
IndexSize &= sizeof(short) = 2bytes && (6.2) \\
VertexBufferSize &= n * VerticesPerTriangle/6 * VertexSize \\
&= n * 3/6 * 28bytes = n * 14bytes && (6.3) \\
IndexBufferSize &= n * VerticesPerTriangle * IndexSize \\
&= n * 3 * 2bytes \\
&= n * 6bytes. && (6.4) \\
Total &= VertexBufferSize + IndexBufferSize \\
&= n * 20bytes && (6.5)
\end{aligned}$$

Note : division by 6 because each vertex is used six times (for 6 different triangles).

As long as there is no need to move neither the light source nor the medium surface model, all its vertices will allways have the same lighting values. This property allow to perform a prelighting on these vertices, which consist of altering their color to create the lighting effect. This technique offers two advantages: first the vertices do not need to own a normal vector anymore and second the vertex buffer will therefore be smaller thus faster transfered to the graphic card, and the graphic card will not need to compute real-time lighting anymore on the surface model, which will lead in a better frame rate. The new vertex size, the new vertex buffer size and the new total size are given by the following equations.

$$\begin{aligned} VertexSize &= sizeof(float) * NbCoordinatesPos + sizeof(Color) \\ &= 4bytes * 3 + 4bytes = 16bytes \qquad (6.6) \\ VertexBufferSize &= n * VerticesPerTriangle/6 * VertexSize \\ &= n * 3/6 * 16bytes = n * 8bytes \qquad (6.7) \\ Total &= VertexBufferSize + IndexBufferSize \\ &= n * 14bytes \qquad (6.8) \end{aligned}$$

The new data set represents 70% of the preceeding one. The rectangles will not only load faster ($90ms$ instead of $400ms$) but also the graphic card will be able to handle 1.4 times the number of rectangles it could have loaded before. The lighting values are computed the same way as explained in section 6.2.3 and mixed with the actual color of the vertex.

Surface Loader

As each rectangular portion of the medium surface still takes a long time to load ($80ms$), and the program has to load new surfaces even when the user is moving the camera, this operation is performed by a separate thread called the Surface Loader. This way the user can move the camera smoothly even when new rectangles are being loaded.

The Surface Loader thread accesses a shared object that contains a list of the rectangles that have to be in the scene and, because the analyzer may change some measured values, a list of measurements to reload. The GUI thread, which is basically the main window of the program, build these two lists according the camera position and what the analyzer did. The Surface Loader thread can then read those lists and load the needed rectangles and unload the ones that are not needed anymore. Because the lists may change while the Surface Loader is processing them, it works on a copy of them.

The GUI thread makes use of three flags encapsulated in a shared object to control the Surface Loader thread. The first flag indicates whether the list of rectangles that have currently to be in the scene has changed or not. The second one is set when the program wants to force the Surface Loader to consider the new list of orders it just created instead of finishing processing the old one, and the third flag allows to terminate the Surface Loader thread. Its workflow is explained underneath:

```
1  while true
2  {
3      Lock the three flags.
4      while none of the three flag is set
5      {
6          Release lock on the three
7          flags and wait to get lock again.
8      }
9      Clear flag "force consider new list".
10     if flag "terminated" is set
11     {
12         Release lock on the three flags.
13         Exit While.
14     }
15     Lock the orders list, copy it, unlock it.
16     Clear flag "orders list changed".
17     Release lock on the three flags.
18     foreach rectangles to (un)load
19     {
20         Lock the three flags.
21         if flag "force consider new list" is set
22         {
23             Unlock the three flags.
24             Exit While.
25         }
26         Unlock the three flags.
27         (Un)load the rectangle
28     }
29     Correct all the measurements to correct.
30 }
31 Unload all squares.
```

The Surface Loader gets a list of rectangle to be in the scene. It could unload all the rectangles and load the one that are in the list each time it gets a new list but because of the required loading time it has handle its rectangle in a little smarter way. It keeps a list of all the rectangle that are currently in the scene up to date. Each time it gets the new list, it determines which rectangles have to be loaded and which one have to be unloaded. The following algorithm performs that in a complexity of O(n) but only works if the list of rectangles is sorted, that is why rectangles must have an ordering relationship between them made in a way that none of them is equal to another one.

```
1  i = 0, j = 0, newRectInSceneList = empty list
2  while i < nbRectInScene or j < nbRectToBeInScene
3  {
4      if i < nbRectInScene
5          rectInScene = rectInSceneList[i]
6      else
7          rectInScene = null
8      if i < nbRectToBeInScene
9          rectToBeInScene = rectToBeInSceneList[j]
10     else
11         rectToBeInScene = null
12     if rectInScene = null or
13     (rectToBeInScene <> null
14     and rectInScene > rectToBeInScene)
15     {
16         Load the rectangle "rectToBeInScene".
17         Add it to newRectInSceneList.
18         j = j + 1
19     }
```

```
20      else if rectToBeInScene = null or
21      (rectInScene <> null
22      and rectInScene < rectToBeInScene)
23      {
24          Unload rectangle "rectInScene".
25          i = i + 1
26      }
27      else
28      {
29          Add rectInScene to newRectInSceneList.
30          i = i + 1
31          j = j + 1
32      }
33  }
```

6.2.4 Information exchange

The analyzer and the graphical user interface need to communicate. The graphical user interface may ask the analyzer to pause, resume, start or stop whereas the analyzer may ask the graphical user interface to display some data. The following sections explains the communication process between these two parts.

GUI talks to Analyzer

The GUI talks to the analyzer by invoking some thread commands. Actually these thread commands are mapped by analyzer specific methods that are declared in the analyzer interface. These method allow the graphical user interface to start, stop, pause and resume the analyzer thread. It is important to keep in mind that these methods do not guarantee that the thread will be immediately aware of the call.

Analyzer talks to GUI

The analyzer thread might want the graphical user interface to be notified of its progress. For this reason events have been chosen to make this communication. The graphical user interface has to subscribe to the event it is interested in then the analyzer will fire them when needed. There are four kinds of event that the analyzer may raise:

- Data ready event
- Groove data event
- Text event
- Progress event

The data ready event is sent when the analyzer thinks that some data he processed are interesting and should be displayed. This event can hold curves to draw in the 2D view and points to reload in the 3D view.

The groove data event is raised when the analyzer extracted some groove data. Typically the GUI1 subscribes to this event to update the sound panel when it has to.

The text event is raised when the analyzer wants to send some text to the graphical user interface. In most of the cases these texts are printed into a console.

The progress event is fired when the analyzer wants to notify the graphical user interface of its progression during a long task.

6.3 Implementation

6.3.1 Class Diagram

The classes diagrams can be found in the Appendix B. There are 7 diagrams, which each explain a part of the existing libraries in Groovster exept the first one, which is a global view:

- The global architecture diagram
- The Rectangle Accessor diagram
- The Simple Accessor diagram
- The SimpleUI diagram
- The Full 3D diagram
- The Dictabelt Analyzer diagram
- The Dictabelt Corrector diagram

6.3.2 Interfaces Description

IAnalyzer

The interface IAnalyzer describes the methods and the properties to implement to create a compatible analyzer. Some of them are used to control the analyzer thread, some are for the events subscribing. The method *Init* should be used as a delayed constructor, because the object could be instanciated but never used.

IBoundBoxed

IBounding box forces a class that implements it to own an axis aligned bounding box. Objects that implement this interface can be added to an octree. (octrees and AABBs are explained in Section 6.2.3).

IDescriptible

This interface is designed to make the classes that implements it to be able to describe themselves. It has two properties: *DescName*, which should return the name of the class and *DescDescription*, which should return the description of the class.

IGui

The graphical user interfaces of Groovster have to implement this interface to be able to communicate both with the analyzer and the measurement accessor. The method *Init* should be used as a delayed constructor, because the object could be instanciated but never used.

IMeasurementAccessor

This interface describes the methods and the properties to implement to create a compatible measurement reader. The main methods are *GetHeight* and *SetHeight*, which provide an access to the measured surface heights. They hide the way the data is stored in the file. If the file format changes, the programmer has only to write a new class that implements this interface.

IPointSet

IPointSet describes a generic point set structure that is "published" by the analyzer.

6.3.3 Classes Description

AnalyzerDataReadyEventArgs

Event argument sent from the anayzer every time some data has been processed and is ready to be consumed.

AnalyzerGrooveDataEventArgs

Event argument sent from the anayzer when groove data is ready.

AnalyzerProgressEventArgs

Event argument sent from the anayzer to tell its progress.

AnalyzerTextEventArgs

Event argument sent from the anayzer when it wants to send text information. This object is typically used to send debug information from the anayzer to the user interface.

ApplyBadGroovesFilter

Remove bad regions at the beginning and the end of the groove file. Indicates to the user bad regions that are in the middle of the groove file. If several consecutive groove positions could not be found by the analyzer, the region is considered bad.

BitMapWriter

This class is used to create a bitmap given its width and height and draw in it using the *SetPixel* method. It can then save the image as a PNG file.

BoundingBox

BoundingBox represents an axis aligned bounding box (AABB are explained in Section 6.2.3).

CircleFit

This class provides some static methods used to fit a circle as well as possible in a group of points.

CircleOp

Finding the center of a circle given two points on it and its radius can be achieved by a static method of this class. Another one computes the distance from a point to a circle.

Controller3D

This is the leader for the 3D view in $GUI1$ and $Full3D$. It decides which portions of the surface have to be loaded and which have to be unloaded using an octree and a frustum (see Section 6.2.3). Then a thread that it created and controls take care of loading and unloading these portions.

DictabeltAnalyzer

This class is the analyzer for the dictation belts. It implements IAnalyzer. From the scanned surface of a dictation belt it extracts as well as possible the sound that was recorded on it, using different algorithms. At this point there is no a way to select which algorithm will be used but modifying a method call in the code.

DictabeltAnalyzerView

DictabeltAnalyzer view is the main window of $GUI1$. It implements IAnalyzer. It has three panels: the surface panel, the sound panel and the console panel. The surface panel displays a 3D view of the medium scanned surface and a 2D view of a slice of it. The sound panel shows the actual extracted groove and the console panel displays text messages that were sent by the analyzer.

DictabeltCorrector

Correcting measurement errors on the scanned surface is the job of this class, which implements IAnalyzer. From the scanned surface of a dictation belt it creates another surface that is supposed not to contain all these errors using a sequence of algorithm, which can be described by a setting (see Section A.2.4 for the settings desciption).

DllSelection

This class displays a window that allow the user to choose which libraries he wants to use in Groovster. It searches the folder where *Groovster.exe* is located for compatible libraries. There are three kind of libraries to select: the analyzer,

the graphical user interface and the measurement accessor. Once the user clicks *Ok* the properties *ma*, *gui* and *analyzer* contains each an object representing respectively the measurement accessor, the graphical user interface and the analyzer. The method *Init* has to be called on each of them before using them. If the user clicked *Exit* or closed the window the *Exit* property is set to true.

FullScreenView

FullScreenView is a class that implements IGui and shows a full screen 3D view of the scanned medium surface. It does not use the analyzer.

Frustum

This class represents a viewing frustum. Each time the view matrix or the projection matrix changes it has to be updated using the *Update* method to consider the new viewing frustum, which causes it to recompute its 6 planes. The method *ContainsBoundingBox* allow to test if a given axis aligned bounding box (AABB are explained in Section 6.2.3) is inside or outside the viewing frustum or intersects with it.

Functions

All the global method should be in this class, which provides only static methods. Any other class should have access to it.

Globals

Globals provide some global static method for the Yerlutz3D project.

GrooveAccessor

Access data from a groove file. A groove file contains a signal, which is basically a list of groove points. The groove file also contains the sampling rate of the signal.

GroovePoint

Class that contains a comparer to compare groove points in respect to their value in X. This class can be extended to have one comparer for each dimension (x,y,z).

HoldableButton

The holdable button looks like a standard Windows button exept that it fires events at regular intervals as long as the button is pressed. It behaves like the keyboard, that means that when it is pressed it fires one event, then it waits during a given delay and then it starts repeating events.

IniReader

The IniReader class provides an easy way to access configuration settings. The operator [] as been defined so that the programmer can use the syntax $myIniReader[key]$ to access a setting given by its key.

InputBox

This class allows to display a message box in which the user has to edit a text and can then click *Ok* or *Cancel*. This class was taken from [11].

MainClass

This is the starting class of Groovster. It contains the $Main$ method, which is the entry point of the program.

O3D_Grooves

This is an extension to Object3D. It is used to draw the extracted grooves in the 3D view.

O3D_Meshed

This is an extension to Object3D. It contains an additional mesh to store the vertices and indices.

O3D_Plane

This is an extension to Object3D. It is used to draw the red rectangle in the 3D view that represents the slice currently shown in the 2D view.

O3D_Surface

This is an extension to O3D_Meshed. It is designed to hold a portion of the scanned surface as a 3D object that can be inserted in the scene. It creates a prelighting of the vertices to reduce the size of the data transmitted to the graphic card, so the scene lights should not be moved after the creation of instances of that class.

Object3D

This is the base class for any 3D object. It has a position and basic methods like *Draw*, which draws the object to the backbuffer or *Restore*, which should be called when the device has been lost.

Octree

This class represents an octree (octrees are explained in Section 6.2.3). The *AddObject* allows to add objects which implement the IBoundBoxed interface to the octree. $GetVisibleObjects$ gets all the visible objects from a given frustum.

OutputErrorImageFile

Converts the data that is contained in the groove file to an image that represents the distribution of the errors that occured during the analysis process. For each groove point in the groove file a pixel in the image represents its error level. The higher the error is the darker is the pixel. The $Perform$ method must be called to do this operation.

OutputTxtFile

Converts the groove file to a simple text file that contains one groove bottom position per line. The $Perform$ method must be called to do this operation.

OutputWavFile

Creates a wave file (.wav) from a groove file. The $Perform$ method must be called to do this operation.

OutputXPosFile

Converts the groove file to an SGL xpos file that can be used later by LabView. The $Perform$ method must be called to do this operation.

ParamConfigForm

This class shows a form that allows the user to edit the Groovster settings.

Progress

Progress is a form that contains either a progress bar or a message. It is used to show the progress of a long task. The $Progress$ property sets the fill percentage of the progress bar whereas the $Message$ property sets the text to display.

RectangleAccessor

Reads measurements from a file data. The measurement file contains a height map in SGL format. The data file contains a header followed by the height points one by one in SGL format. This class loads only partially the data file. It separates the whole measurement surface in many rectangles. If a measurement that is not in memory is accessed, the class loads the rectangle that contains the point in memory and unloads the least recently used rectangle.

Resampling

This class implements the resampling function used in LabView. It uses finite response filters to do the resampling operation.

Scene3D

This class is used to represent a three-dimensional scene in which 3D objects can be added and removed. $AddViewPort$ and $RemoveViewPort$ allow to add or remove a view port from the scene's view ports list. The method $RenderAll$ renders the scene to all the subscribed view ports.

SGLConverter

This class provides static methods to convert Labview's SGL floating point value to C#'s float and vice versa.

SimpleAccessor

Reads measurements from a file data. The measurement file contains a height map in SGL format. The data file contains a header followed by the height points one by one in SGL format. This class seeks in the file every time a point is accessed.

SimpleGUI

This class is used to run the analyzer in the background without any graphical display to speed things up. It only shows the console.

ViewPort3D

This class represents a view port that is like a camera in a 3d scene. The camera is moving on a sphere and looking at its center. The sphere can be moved and the radius can be varied. It is a control and can thus be placed on a form using the Visual Studio designer.

Waveform

This class represents a wave form, which is basically a serie of samples with a given frequency (sample rate).

WavFileAccessor

The WavFileAccessor class allows to create a wave file (.wav) from a given wave form. The sampling frequency can be chosen between $11025Hz$, $22050Hz$ and $44100Hz$. The number of bits per sample is fixed to 16. The wave file contains only one channel.

ZedGraphControl

ZedGraph is an open source library coded in C#. It provides a control called ZedGraphControl that can be placed on a form. It is a really easy way for plotting some curves. This library can be downloaded from [11].

6.4 Issues and Known Bugs

There are still some bugs in Groovster. This section shows a list of known bugs and issues that are present in Groovster 1.0.

6.4.1 Device Lost Exception

Affects $Gui1$ and $Full3D$. This error makes Groovster crash down and occurs in the 3D view when the application lost the graphic card because another application took control over it. This can typically happen when the screen saver starts, when hanging a second monitor or when the Windows login screen is shown. To solve this problem, the class $Scene3D$ should be modified.

6.4.2 Out of Memory Exception

Affects $Gui1$ and $Full3D$. This error occurs when the system is running out of memory. Because the 3D view might consume a lot of memory when a large portion of the surface is shown and that the program does not perform any check upon available memory Groovster may crash down if there is not enough RAM. To solve this problem, the class $Controller3D$ should be modified.

Another cause of this problem is that Groovster consumes a lot of RAM for many operations it performs. To try to avoid this trouble you should reduce the size of allowed memory for the Rectangle Reader (which would by the way make Groovster run slower), and do not create any PNG file nor any XPOS file. If the surface you analyse contains more than about 2 billions of measurement points and you ask to produce some png files, Groovster will inevitably crash down because the size of the internal bitmap will be over about $2GB$, which is actually the larger data set that the Bitmap class can handle.

6.4.3 Incomplete Surface

Affects $Gui1$ and $Full3D$. This issue causes the scanned surface to be truncated at the right and the far borders (X+, time+). This is due to the fact that only complete squares of 64x64 measurements are loaded into the 3D view, so if the number of measurement points in the width of the surface is not a divisible by 64, the remaining points will not be loaded. The same happens for the time direction. To fix that, the class $Controller3D$ should be modified.

6.5 Future

As previously mentionned the software is designed to be easily adapted to be able to extract sound from other mechanical media. This section gives guidelines to extend Groovster for some other media. Please refer to Appendix A.3 to get more information about extending Groovster.

6.5.1 Gramophonic Records

Scanned with the 3D scanner this media produces a measured surface very similar to the dictation belts. There is thus very few adaptations to perform.

- Create a new analyzer by copying the existing DictabeltAnalyzer.
- Adapt the analyzer to match the groove shape of a gramophonic record.
- Change the RPM setting to the actual rotation speed of the gramophonic record.

- Change the *Length* setting to circumference at the average radius of the record.

6.5.2 Phonographic Cylinders

- Create a new analyzer by copying the existing DictabeltAnalyzer.
- Adapt the analyzer to match the groove shape of a phonographic cylinder.
- Adapt the analyzer to take the height displacement in consideration and not the width displacement anymore.
- Change the RPM setting to the actual rotation speed of the phonographic cylinder.
- Change the *Length* setting to circumference of the cylinder.

6.5.3 Phonographic Records

The phonographic records in term of software extension is inbetween phonographic cylinders (Section 6.5.2) and gramophonic records (Section 6.5.1). It should be easy to merge these two analyzer to get a new one capable of analyzing phonographic records.

6.5.4 Stereophonic Records

The existing dictabelt analyzer puts in its groove file the position of the groove in width, height and time. By some shape adaptations, it should be easy to extract the two channels. Of course the wave writer has to be modified to produce a wave file that contains two channels.

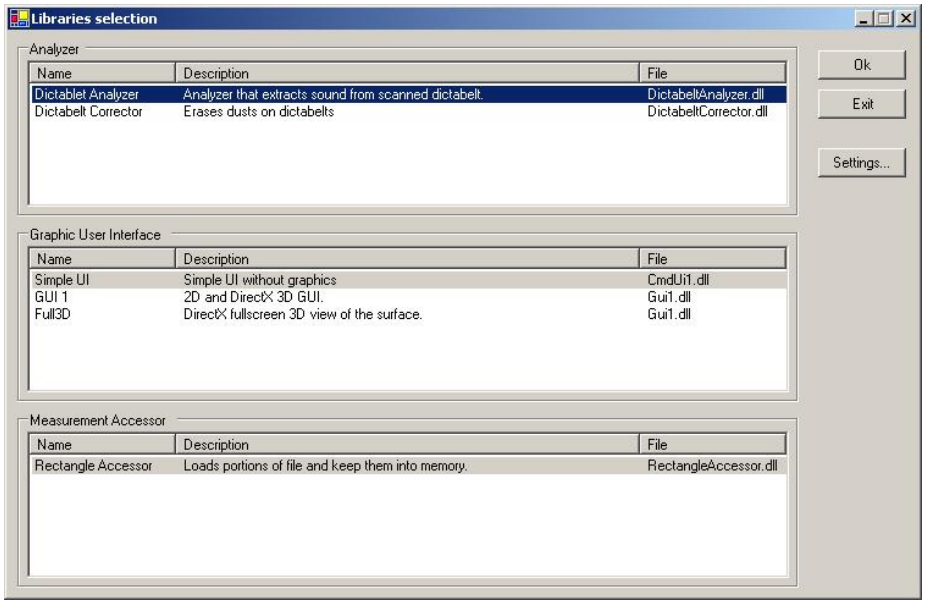


Figure 6.2: Libraries selection window

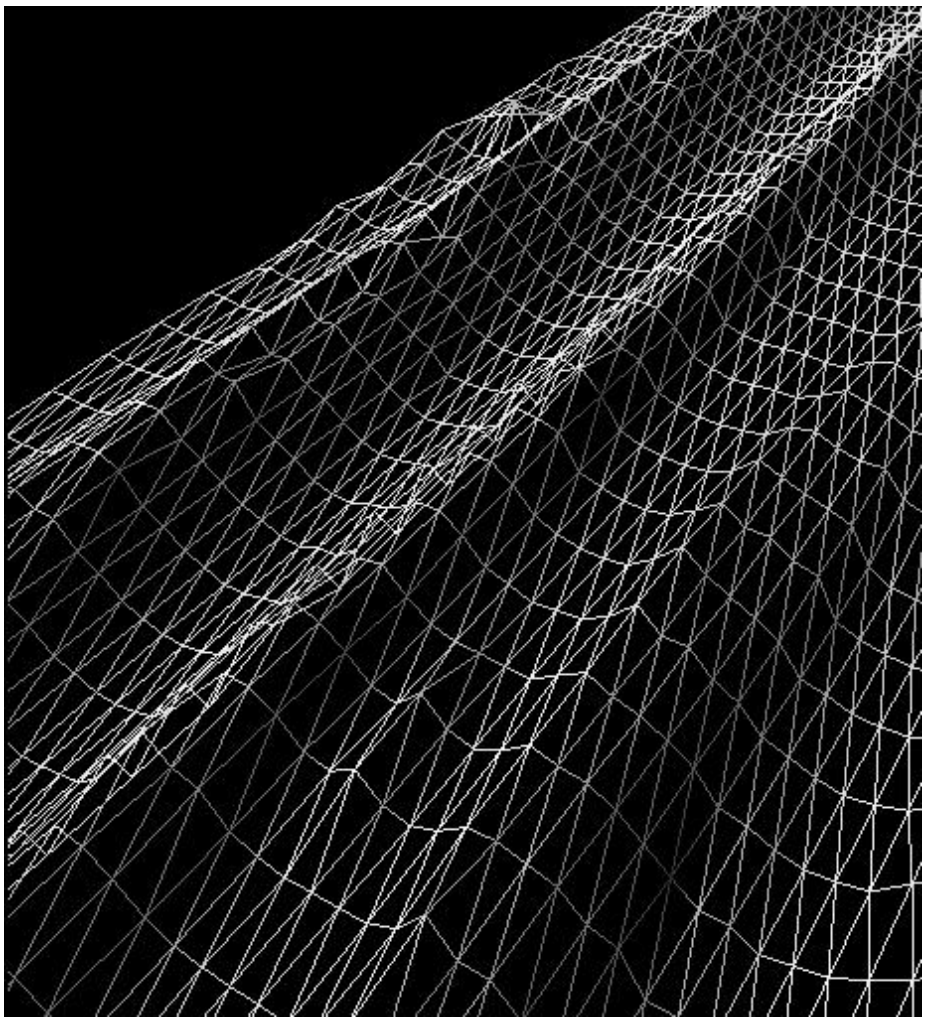


Figure 6.3: Surface triangulation

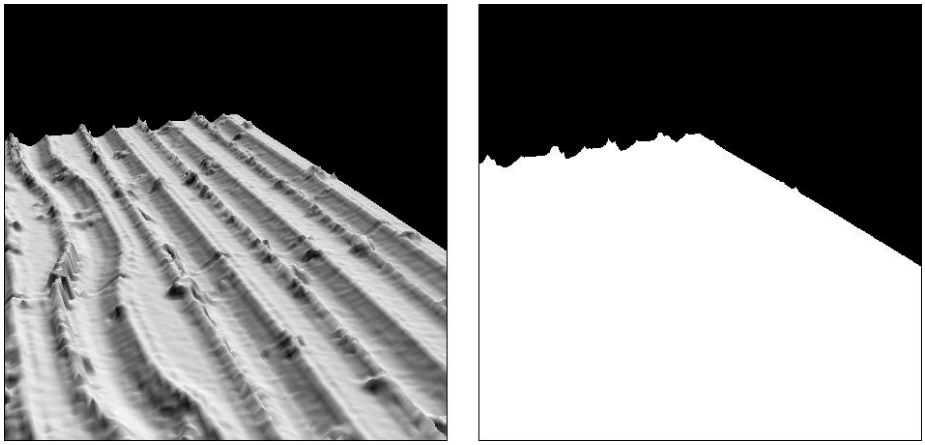


Figure 6.4: Lightings *Both picture show the same part of the model. The first one is not lighted and the second one is.*

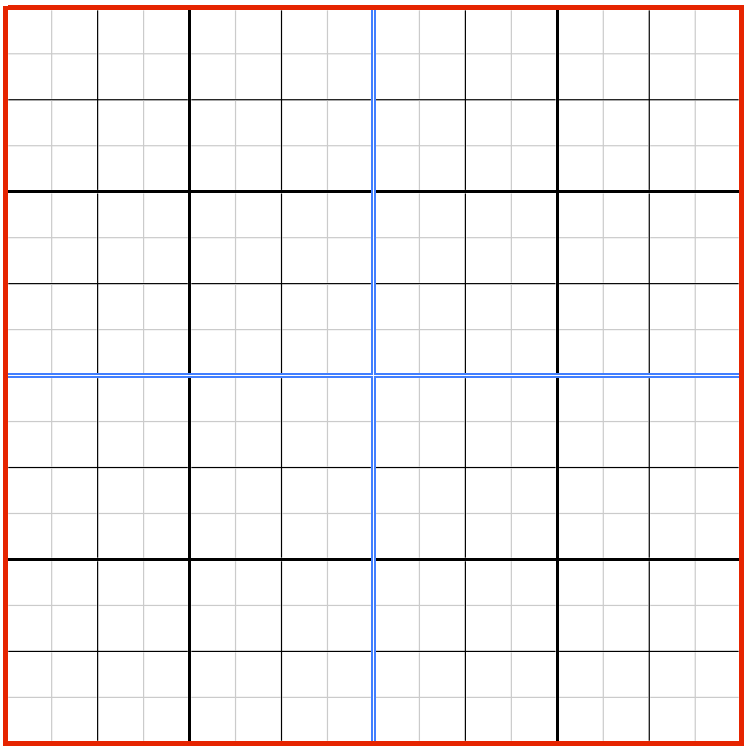


Figure 6.5: Quadtree example *Each cell of the quadtree is split in 4 smaller cells until a given nesting level.*

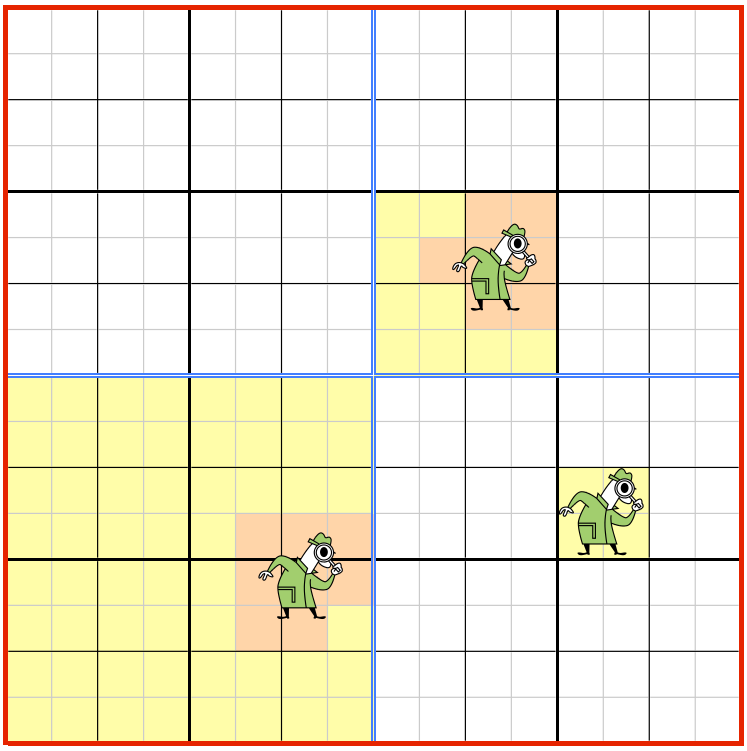


Figure 6.6: Quadtree containng objects *The red cells shows the cell that the object crosses and the yellow one are the smaller cells that fully contain the objects.*

Chapter 7

Tests and results

7.1 Signals Comparision

Figure 7.1 shows an important comparison of signals at different frequencies measured at different places in the measurement process and after data analysis process.

Recorder Input is the sound generated with the sound card that is sent into the dictation belt machine. The signal measured across the recording needle is called *Recording Needle*. *Optical Extraction* is obtained using the 3D measuring technique and the Groovster data analysis software. The measured at the playback needle is called *Playback Needle*. The *Speaker Output* corresponds to the signal measured at the dictation belt player speaker output. Note that the amplitudes were scaled and therefore they can not be compared.

A first observation is that the signals read out are closer to the input signal at higher frequencies than at lower frequencies. The system might be designed to have the best frequency response, with the lowest distortion, at frequencies higher than $1kHz$. This would be very surprising since the human voice has a lot, if not most, frequency components below $1kHz$.

In addition the signal at the recording needle at $100Hz$ and $400Hz$ is clearly the derivative of the input signal. This becomes less visible as the frequency goes higher. This means that the electronics between the recorder input and the recording needle boosts high frequencies by taking the derivative.

Since the recording needle is a damped harmonic oscillator one would expect the signal on the belt to be the integral of the signal across the recording needle. At $400Hz$ and above this appears to be the case. At $100Hz$ the playback needle signal is very similar than the recorded signal. This suggests that the recording needle movement is driven by the input voltage as opposed to the integral of the input voltage signal. $200Hz$ might be the limit between the two.

The input signal is definitely deformed by the electronic components before it is applied to the recording needle. But it seems that the signal applied on the recording needle is symmetric, at least above $200Hz$. This symmetry implies an X,Y correction curve could be used in the time domain to get back to the original signal. If it is possible to get form the optical extraction tho the signal applied at the recording needle it might be possible to use a correction curve to get back to the original signal.

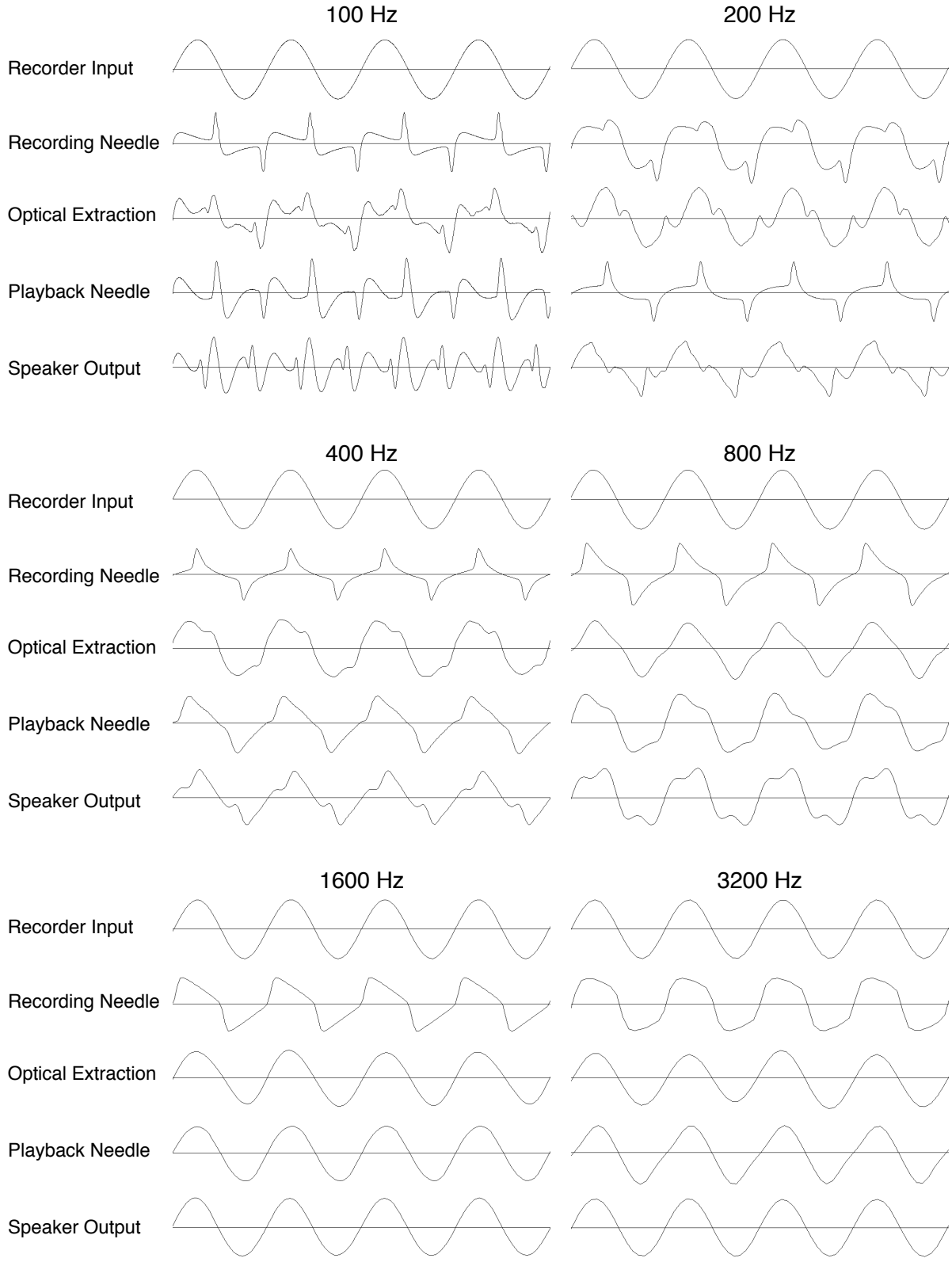


Figure 7.1: Signal amplitude comparison.

From this comparison it is hard to tell with certitude what is really on present on the dictation belt. The frequency response of the recording pickup is not known and since different results are experienced when comparing the optical extraction and the signal measured at the playback needle it is even hard to tell which is the "true" representation of the groove on the belt. The optical readout is probably a more accurate representation of what is on the belt. Although some high frequency noise is experienced in the optical measured data, the optical method has a much wider frequency response than the playback needle, which means that it does not tend to average the groove movement like the playback needle does.

7.2 Measurement Quality

The output wave form of the data analysis process shows a lot of high frequency noise. Some extend of it is removed by the low-pass filter, but it can still be heard in the audio file. This noise is probably introduced either by the data acquisition process or the data analysis process. The noise of the measurement points is relatively high compared to the groove depth; it is around 10%. Since the noise level is lower when the stage is not moving it is either introduced by the linear stage or the trigger. The CHR handbook states that the response time of the trigger is around $10\mu s$ independently of the sampling frequency. This means that the probe rate needs to be synchronized with the incoming trigger. Previous measurements with a different motion controller and trigger signal generator have shown that the synchronization is not done properly by the CHR. The manufacturer appears to know that problem but was not able to give a solution to fix that.

7.3 Sound Quality

The sound quality is comparable to the speaker output. No noise reduction filters were used on the extracted sound files. Since the noise is probably introduced by the measurement process or the data acquisition process, there is no point in removing the noise at this point. The source of the noise has to be identified first. Once it has been shown that the noise level was reduced to a maximum, noise reduction filters could be applied to filter the residual noise.

7.4 Processing Time

The data acquisition process takes about 50 hours to measure one minute of sound at a sampling frequency of $50.4kHz$. The time needed to analyze such a data file, which is about $200MB$, is several orders of magnitude lower. On a nowadays workstation it takes less than 20 minutes. It can go down to 5 minutes depending on the data analysis algorithm that is used.

In this project no particular emphasis was placed in the speed of execution of the data analysis process. The architecture of the software was designed to be modular in order to make it very easy to extend the software. This adds some computational overhead. Since the data acquisition takes so much time there no point in optimizing the execution time of the analysis process.

Chapter 8

Conclusion

The results obtained show that the 3D metrology technique to measure dictation belt surfaces and extract the groove displacement works. The sound can be extracted without any contact with the surface and with a sufficient quality to hear the speech of someone taking. The quality of the data acquisition process and methodology needs to be improved and refined in order to get a better quality of the measurements, which has obviously a big impact on the result.

The data analysis might also be improved by considereing different groove detection techniques and improving the existing ones. Another algorithm would be to fit a sphere in the groove over several groove slices to get the position of the groove. The sphere is probably a better approximation of the needle's shape than a simple 2D circle or parabola, but it would lead to a much more complex algorithm. It is not expected to reduce the noise, but some distortion due to the fact that the implemented algorithms determine the groove position not exactly where it is when the grooves move in the higher frequencies range.

The software created is a powerful tool that can be easily modified and extended to extract the sound of other kinds of mechanical media. Its different views have and will really help for the development of new features. The estimated time to add capabilities for scanning another type of media is between one and five days. The user can then really focus on what would make the sound better.

Chapter 9

Acknowledgements

The authors would like to thank in particular Carl Haber for his advices, help, support, patience and especially for giving them the opportunity to do this work at the Lawrence Berkeley Lab. They also thank Prof. Ottar Johnsen for having taken the time to visit them and discuss several signal processing issues. They thank Prof. Frédéric "Slangster" Bapst for his comments and advices and Vitaliy Fadeyev for his introduction to the project. Last but not least the University of Applied Sciences of Fribourg, which has partially funded the stay of the autors.

Bibliography

[1] Fadeyev, V., Haber, C., "Reconstruction of Mechanically Recorded Sound by Image Processing", LBNL Report-51983.

[2] Fadeyev, V., Haber, C., "Reconstruction of Recorded Sound from an Edison Cylinder using Three-Dimensional Non-Contact Optical Surface Metrology".

[3] Cavaglieri, S., Johnsen, O., and Bapst, F., "Proc of AES 20^{th} International Conference", Budapest, Hungary 2001, Oct 5-7.

[4] Johnsen, O., "Le changement de fréquence d'échantillonnage et le suréchantillonnage", Ecole d'Ingénieurs de Fribourg, March 2004.

[5] Mitra, S., K., "Digital Signal Processing, A Computer-Based Approach, Second Edition", McGraw-Hill International Edition, 2001.

[6] Press, W., H., Teukolsky, S., A., Vetterling, W., T., Flannery, B., P., "Numerical Recipes in C, The Art of Scientific Computing, Second Edition", Cambridge University Press, 1988-1992

[7] STIL S.A., "CHR 450, Operation and maintenance manual".

[8] National Instruments, Test and measurement, http://www.ni.com.

[9] Wikipedia, The Free Encyclopedia, http://en.wikipedia.org.

[10] Multi-threading in .NET, Introduction and Suggestions, http://www.yoda.arachsys.com/csharp/threads/.

[11] Source Forge, http://www.sourceforge.net.

Appendix A

Groovster User guide

A.1 Installation

A.1.1 Requirements

In order to run Groovster you must have the following software :

- Microsoft .NET Framework 1.1
- Microsoft DirectX 9

Both are free for use and can be downloaded from Microsoft's website (www.microsoft.com). If one of these components is not currently installed on your computer, please refer to the following sections.

Groovster was tested on a PC running Windows XP with a 1.8GHz 32-bit processor, 1024GB RAM, stripping serial ATA and a ATI with 128MB memory as graphic card.

A.1.2 Microsoft .NET Framework

The .NET Framework is required to run any program written in a .NET language. It is basically a virtual machine that transcripts the CIL (common intermediate language) into binary code that can be fed to the processor. Groovster uses the version 1.1 of this framework, which is free and can be downloaded from Microsoft's website (www.microsoft.com). Please follow their installation instructions.

A.1.3 Microsoft DirectX 9

DirectX 9 is an API (application programming interface) used to provide hardware direct access and emulate features that are not supported by it. Groovster uses Direct3D, a part of DirectX, to access the graphic card to render its 3D scenes with an acceptable frame rate. DirectX is free and can be downloaded from Microsoft's website (www.microsoft.com). Please follow their installation instructions. If you experience troubles running Groovster, you might need to install the DirectX 9 SDK april 2005.

A.1.4 National Instrument Measurement Studio

Measurement Studio is an library dedicated to signal processing, automation and much more. You have to install it before running Groovster. You can download an evaluation version on www.ni.com. Please follow their installation instruction.

A.1.5 Groovster

Once you have installed the .NET Framework and DirectX 9, you should be able to run Groovster without any special handling. You may want to copy the whole Groovster's directory to your hard disk drive.

A.2 Using Groovster

When you start Groovster you get first a window asking you to select the libraries you want to use. The analyzer library is actually the algorithm that will be applied on your data to extract the sound or fix some scratches. The GUI is the graphical user interface you want to use and the measurement accessor defines the way your data file are going to be read. The Groovster's existing libraries are described in sections A.2.1, A.2.2 and A.2.3.

On this starting window there is a button labelled "Settings". When clicking it the configuration window opens and allow to set the values for any setting. To change one of them double-click on it. The meaning of each setting is explained in section A.2.4. Once you close this window it will ask you if you want to save the changes or not. If you click "Yes" the newly defined settings will be used for all future operations. Settings are saved in "config.xml". You can backup this file if you want to restore old settings.

If you start Groovster passing some command line arguments it will assume that you want to run it without graphical user interface. This allow to create batch file to automate a serie of analyses. The syntax used to call Groovster in background mode is the following:

groovster.exe analyzer accessor file settingchange*

- analyzer : path to the library (dll) that contains the analyzer you want to use.
- accessor : path to the library (dll) that contains the measurement accessor you want to use.
- file : path to the measurement file (dll) you want to analyze.
- settingchange : new setting definition. The syntax is section:key:value. Example: "DictabeltAnalyzer:OutputXPosFile:off". There can be zero, one or many setting changes separated by spaces.

A.2.1 Analyzers

Dictabelt Analyzer

The dictabelt analyzer is designed to extract the sound from a dictation belt. It analyzes a measurement file where the time is spanned over the Y axis and

the groove displacement over the X axis. It produces a groove file that contains information about the groove position and from this file it generates the sound.

Dictabelt Corrector

The dictabelt corrector is an analyzer that takes as input a file that contains measured data from a dictation belt where the time is spanned over the Y axis and the groove displacement over the X axis. It produces another measurement file that is supposed to contain the same data without any measurement errors and no warpage.

A.2.2 GUIs

SimpleUI

This GUI simply display a console that will show debug information. Select "Open" in the "File" menu, pick your measurement file and the analyze algorithm should start. You can just close the window when it is over. At any time you can save the content of the console by clicking "File" and "Save".

GUI1

GUI1 is the most sophisticated GUI provided with groovster. There are three tabs displaying either the surface, the sound or the console explained in the following paragraphs. The analyzer execution can be controlled with the buttons "play", "pause" and "stop". The option "Autmatic pausing" in the "Options" menu allows the algorithm to pause itself when it thinks it should (decided by the analyzer's programmer).

The surface panel is splitted in two parts. The left part shows a 3D view of your measurements file and the right part shows a slice of it. A red, translucent rectangle shows in the 3D view what is shown in the slice view. You can move and resize this rectangle using the three control circles underneath the slice view. The leftmost circle is used to move the rectangle left, right, up and down. The center button of this circle allows to scale vertically the rectangle according the slice's profile. The middle circle is used to move the rectangle forward and backward. You can enter a slice number to jump over the surface. With the rightmost circle you can scale horizontally or vertically the rectangle. You can also zoom in the slice view by spanning a rectangle with the mouse on the area you want to zoom into. Right click to unzoom. In the "Options" menu you may want to enable the slice tracking, which will cause the camera to move together with the red rectangle.

In the 3D view you can move the camera using the mouse. There are two moving : translation represented by a wind rose and rotation represented by two arrows. The following table shows the associated movement for each mouse axis in both modes.

	Translation mode	Rotation mode
X axis	Move the surface left or right	Rotate horizontally
Y axis	Move the surface forwards or backwards	Rotate vertically
Wheel	Move the surface up or down	Zoom in or out
Right button	Switch to Rotation mode	Switch to Translation mode

The sound view shows the waveform of the extracted groove. You can zoom it by spanning a rectangle with the mouse on the area you want to zoom into. You can also pan it by holding the shift key and draging the mouse over. At a given magnification it will start to display error bars for each extracted point. These error bars are given by the analyzer algorithm.

The console panel is the text output of the analyzer algorithm, the same as SimpleGUI.

Full3D

The analyzer you choose has no effect with this GUI because it is just not used at all. Is is only a full screen 3D view of the surface. Please refer to section A.2.2 for the camera movement.

A.2.3 Measurement Accessors

SimpleAccessor

This was the first designed accessor. It seeks the file for each measurement required by both the analyzer and the graphical user interface. You should prefer using the RectangleAccessor.

RecangleAccessor

When the either the analyzer or the graphical user interface requires a measurement point the RectangleAccessor will check if this point is currently loaded into the system memory. If it is not it will load a whole rectangle of the surface into the system memory so that subsequent accesses in the same region will be much faster.

When the maximum number of loaded rectangle is reached, the RectangleAccessor will unload the less recently used to make space for the next one to be loaded. The size of the rectangles and the maximal memory usage are given by some settings described in section A.2.4.

A.2.4 Settings

These section enumerates Groovster's settings and provides a short explanation for each of them.

Global

Setting	Description
RPM	Rotations per minute of the scanned medium
MediumCircumference	Circumference of the scanned media in micrometers
TemporaryFileExtension	File extension used for the temporary files
PreprocessedFileExtension	File extension used for the preprocessed files
DataFileExtension	File extension used for the measurement file
GrooveFileExtension	File extension used for the groove file

RectangleAccessor

Setting	Description
RectWidth	Number of measurement points in width of a measurement rectangle
RectHeight	Number of measurement points in height of a measurement rectangle
MaxMemoryUsage	Default maximum memory usage allowed in bytes, determines the maximum number of rectangles in memory
AccessMode	("R" or "RW") Default access mode. RW creates a temporary file.

DictabeltAnalyzer

Setting	Description
Max_DZ	in micrometers
Valley_Threshold	in micrometers
Minimum_Groove_Height	in micrometers
Valley_Width	in micrometers
Poly_Width	in micrometers
Poly_Degree	Order of the first polynom to fit
Min_Valley_Height	micrometers
ApplyLowPassFilter	("0" or "1")
LowPassCutoffFreq	Hz
ApplyHighPassFilter	("0" or "1")
HighPassCutoffFreq	Hz
OutputSamplingFrequency	
MaximumBadRegionTimeAllowed	Maximum time in seconds the groove can be unfound without warning or correction by the BadGrooveFilter
MaximumRemovedSound	Maximum sound portion (in seconds) that will be removed at the beginning or at the end of the detected groove of it is detected as bad
GrooveMappingWindowSize	
OutputWavFile	("on" or "off") Tells if the analyzer should produce a wave file
OutputXPosFile	("on" or "off") Tells if the analyzer should produce a xpos file
OutputGroovePngFile	("on" or "off") Tells if the analyzer should produce a groove png file
OutputErrorPngFile	("on" or "off") Test if the analyzer should produce an error png file
GrooveRadius	microns
CP.PointMaxDistance	Circle probe's point maximal acceptable distance in micrometers
CP.MinNumberOfPoints	Circle probe's minimal acceptable number of point on the circle

DictabeltCorrector

Setting	Description
Sequence	("n,n,n,...") Algorithms sequence. n = 1 to 5 for the different algorithm, n = 0 to normalize the surface.

FullScreenView

Setting	Description
AutomaticCamera	("on" or "off") Indicates if the camera is travelling itself accross the surface

Controller3D

Setting	Description
GrooveYOffset	Height offset in micrometers of the groove in the 3D display
GrooveDisplay	("on" or "off") Indicates if the groove is displayed in the 3D view

Yerlutz3D

Setting	Description
FillMode	("solid", "wireframe" or "points") 3D triangles fill mode
ZBufferDepth	("16", "24" or "32") Depth in bit of the Z buffer
LightVector	("x,y,z") Direction of the light

A.3 Adding New Features

Groovster is designed to be extended without having to recompile its source code. This section explains how to create new features. The basic concept is to create a new class library, which contains at least one class that implements some specific interface. Compiling such a project will result in a dynamically linked library (dll). By putting this dll in the same folder as the Groovster executable file it will automatically be taken in consideration at the next run.

A.3.1 Adding an Analyzer

To create an new analyzer you have to create a class library, which contains a class that implements the interface "IAnalyzer" defined in "Globals.dll". An analyzer should create its own thread to run its algorithm because it needs to be able to be started, stopped, paused and resumed. Accessing the measurements via a measurement accessor is thread safe. The following sections describe the methods and the properties to implement. Notice that the constructor should not perform any operation since the library selection has to construct an instance of each analyzer, measurement accessor and graphical user interface to be able to display them in its lists.

void Init(IMeasurementAccessor)

This method is called when the user clicked "Ok" in the libraries selection window. A reference to the actual measurement accessor is passed as parameter. Put your initialization code here.

string DescName (get)

This property is used to retrieve the name of the analyzer, which should be a string not longer than about 20 characters.

string DescDescription (get)

This property is used to retrieve a short description of the analyzer. It should be a string of about 100 characters.

void Start()

This method is called when the graphical user interface wants the analyzer algorithm to start. Usually you should put code that starts your analyzer thread in this method.

void Stop()

This method is called when the graphical user interface wants the analyzer algorithm to shut down. Usually you should put code that kills your analyzer thread in this method.

void Pause()

This method is called when the graphical user interface wants the analyzer algorithm to pause. Usually you should put code that suspends your analyzer thread in this method.

void Resume()

This method is called when the graphical user interface wants the analyzer algorithm to resume after it has been paused. Usually you should put code that resumes your analyzer thread in this method.

void Join()

This method must make the caller wait until your analyzer thread is stopped. Notice that if you are not running your analyzer algorithm in a separate thread, this method can return immediateley. Usually the code for this method resemble as the following one, where *myTread* is your analyzer thread.

```
1  public void Join()
2  {
3      myThread.Join();
4  }
```

AnalyzerState State (get)

This method should return the current state of the algorithm. The different states are

- Running
- Stopped
- Paused

Events

An analyzer can raise 4 kinds of event. Thus it has to expose the corresponding delegates that it is calling when an event occurs. The event listeners usually register to an event handler this way:

```
1  // register a new event handler
2  myAnalyzer.AnEventHandler +=
3      new EventHandler(myDelegate);
```

The operator += forces the analyzer to implement both get and set for this property. The following list shows the events that can be raised by the analyzer, and that have therefore to have a public property that allows to register to them.

- DataReadyEventHandler SliceDataReadyEvent : allows to send a set of interesting measurement points.
- TextEventHandler TextEvent : allows to send some text.
- ProgressEventHandler ProgressEvent : allows to indicate a progress.
- GrooveDataEventHandler GrooveDataEvent : allows to send a set of interesting groove points.

A.3.2 Adding a Graphic User Interface

To create an new graphical user interface you have to create a class library, which contains a class that implements the interface "IGui" defined in "Globals.dll". The following sections describe the methods and the properties that have to be implemented in order to make it work properly. Notice that the constructor should not perform any operation since the library selection has to construct an instance of each analyzer, measurement accessor and graphical user interface to be able to display them in its lists.

void Init(IAnalyzer, IMeasurementAccessor)

This method is called when the user clicked "Ok" in the libraries selection window. A reference to the actual analyzer and measurement accessor are passed as parameter. Put your initialization code here.

string DescName (get)

This property is used to retrieve the name of the graphical user interface, which should be a string not longer than about 20 characters.

string DescDescription (get)

This property is used to retrieve a short description of the graphical user interface. It should be a string of about 100 characters.

A.3.3 void StartGui()

This method is called when the user clicked "Ok" in the libraries selection window and after "Init()" was called. This method should not terminate before the user exits your graphical user interface. If your *IGui* implementation extends *System.Windows.Form* you may probably want to call *this.ShowDialog*(), which is actually blocking.

A.3.4 Adding a Measurement Accessor

To create an new measurement accessor you have to create a class library, which contains a class that implements the interface "IMeasurementAccessor" defined in "Globals.dll". **Getting and setting measurements must be thread safe.** The following sections describe the methods and the properties to implement in order to make it work properly. Notice that the constructor should not perform any operation since the library selection has to construct an instance of each analyzer, measurement accessor and graphical user interface to be able to display them in its lists.

string DescName (get)

This property is used to retrieve the name of the measurement accessor, which should be a string not longer than about 20 characters.

string DescDescription (get)

This property is used to retrieve a short description of the measurement accessor. It should be a string of about 100 characters.

string File (get/set)

This property gets or sets the measurements file name. Since there is no dedicated method to open the file before the user might call "GetHeight" for example, it should open the file when a property set is performed. Do not forget to close the previous file if a property set was called before.

bool IsPreprocessed (get)

This property is used to know if the measurement file has been preprocessed or not. Preprocessed files have the extension ".pre" and raw files have the extension ".dat". It has to return *true* if the file is preprocessed or $false$ otherwise.

GrooveAccessor GrooveAccessor (get)

This property allows to get a reference to the singleton groove accessor.

float GetHeight(int, int)

This method is called to retrieve the value of a measurement given its x (parameter 1) and y (parameter 2) indices. It has to return a floating point value corresponding to the height of the measurement point in micrometers.

float[] GetHeights(Point[])

This method is called to retrieve a set of measurement point values given their x and y indices wrapped in a *System.Drawing.Point* structure. It has to return an array of the heights in micrometers corresponding to each point in the same order.

void SetHeight(int, int, float)

This method is called to set the value of a measurement point given its x (parameter 1) and y (parameter 2) indices. The new height is given in micrometers.

float this[int x, int y] (get/set)

This property defines the operator [] on the measurement accessor class you are coding. It allows to access the measurement point in a more convinient way.

```
1  // this code
2  myAccessor.SetHeight(3, 5, 133.1f);
3  float x = myAccessor.GetHeight(10,4);
4  // is equivalent to this one
5  myAccessor[3,5] = 133.1f;
6  float x = myAccessor[10,4];
```

int SizeX (get)

This property is used to know the number of measurement points in the x direction.

int SizeY (get)

This property is used to know the number of measurement points in the y direction.

double UnitSizeX (get)

This property is used to know the distance in micrometers in the x direction between two adjacent measurement points.

double UnitSizeY (get)

This property is used to know the distance in micrometers in the y direction between two adjacent measurement points.

bool IsMeasurementError(int, int)

This method is called to know if a measurement value, given its x (parameter 1) and y (parameter 2) indices is an error or not. This has to return *true* if the measurement point is marked as an error or $false$ otherwise. If you are not using measurement errors you can always return *true*.

void SetMeasurementError(int, int, bool)

This method is called to mark or unmark a measurement point given by its x (parameter 1) and y (parameter 2) indices as an error. The 3rd parameter is set to *true* to mark the point as an error or $false$ to unmark. If you are not using measurement errors you can leave the code of this method empty.

Appendix B

Class Diagrams

B.1 Binaries

This section describes the classes and interfaces contained in each binary file (also called assembly) of Groovster.

Groovster.exe	DllSelection MainClass ParamConfigForm
Globall.dll	AnalyzerDataReadyEventArgs AnalyzerGrooveDataEventArgs AnalyzerProgressEventArgs AnalyzerTextEventArgs Functions GrooveAccessor GroovePoint IAnalyzer IDescriptible IGui IMeasurementAccessor IPointSet IniReader InputBox Resampling SGLConverter WavFileAccessor Waveform
DictabeltAnalyzer.dll	ApplyBadGroovesFilter BitMapWriter CircleFit CircleOp DictabeltAnalyzer OutputErrorImageFile OutputTxtFile OutputWavFile OutputXPosFile

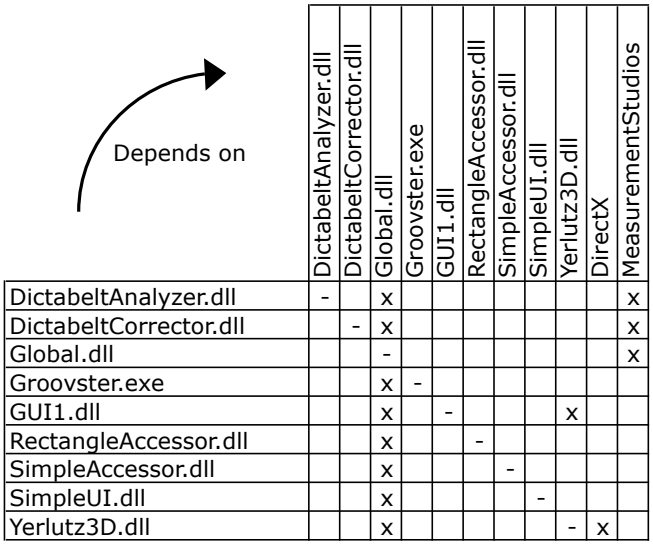
DictabeltCorrector.dll	DictabeltCorrector
GUI1.dll	Controller3D DictabeltAnalyzerView FullScreenView HoldableButton O3D_Grooves O3D_Meshed O3D_Plane O3D_Surface Progress
CmdUi1.dll	SimpleGUI
SimpleAccessor.dll	SimpleAccessor
RectangleAccessor.dll	RectangleAccessor
Yerlutz3D.dll	BoundingBox Frustum Globals IBoundBoxed Object3D Octree Scene3D ViewPort3D

B.2 Dependencies

This section describes the dependencies between the binaries (also called assemblies) of Groovster. The dependencies are represented in the figure B.1

B.3 Part Class Diagrams

This section shows the class diagrams for each part of Groovster.



	DictabeltAnalyzer.dll	DictabeltCorrector.dll	Global.dll	Groovster.exe	GUI1.dll	RectangleAccessor.dll	SimpleAccessor.dll	SimpleUI.dll	Yerlutz3D.dll	DirectX	MeasurementStudios
DictabeltAnalyzer.dll	-		x								x
DictabeltCorrector.dll		-	x								x
Global.dll			-								x
Groovster.exe			x	-							
GUI1.dll			x		-				x		
RectangleAccessor.dll			x			-					
SimpleAccessor.dll			x				-				
SimpleUI.dll			x					-			
Yerlutz3D.dll			x						-	x	

Figure B.1: Dependencies between the assemblies

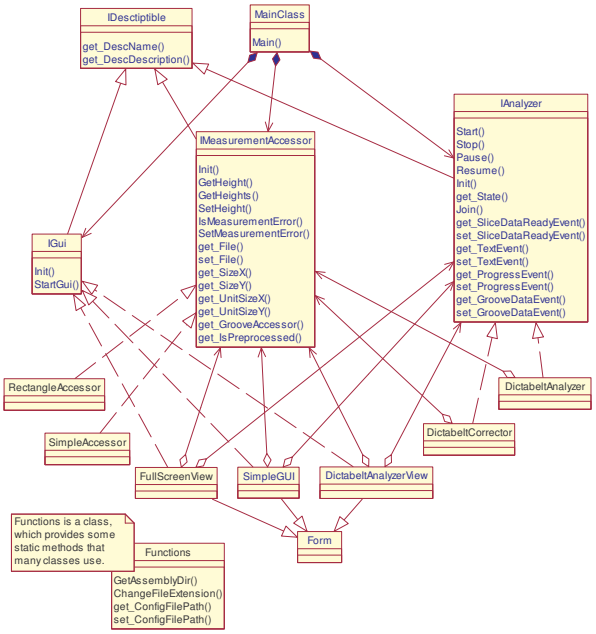


Figure B.2: Global class diagram *Shows the main architecture of Groovster.*

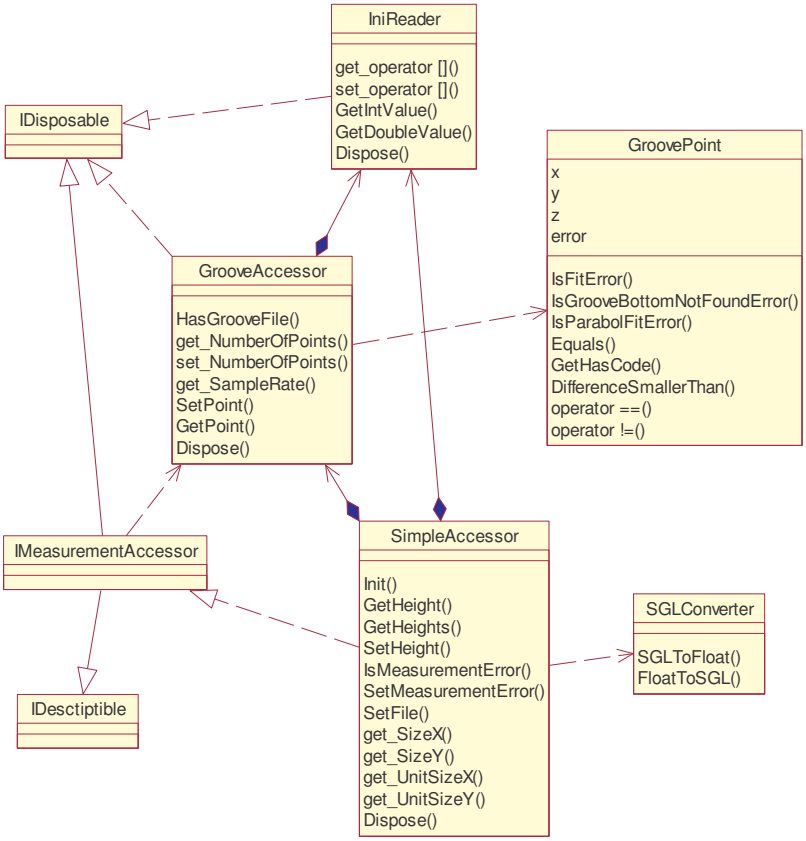


Figure B.3: Class diagram for Simple Accessor

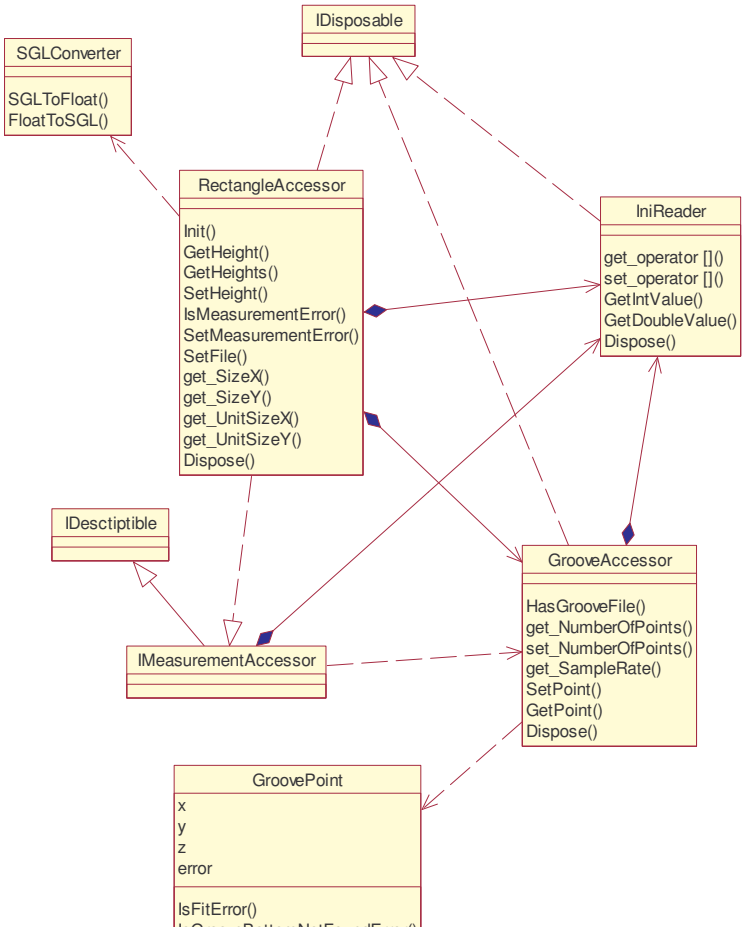


Figure B.4: Class diagram for Rectangle Accessor

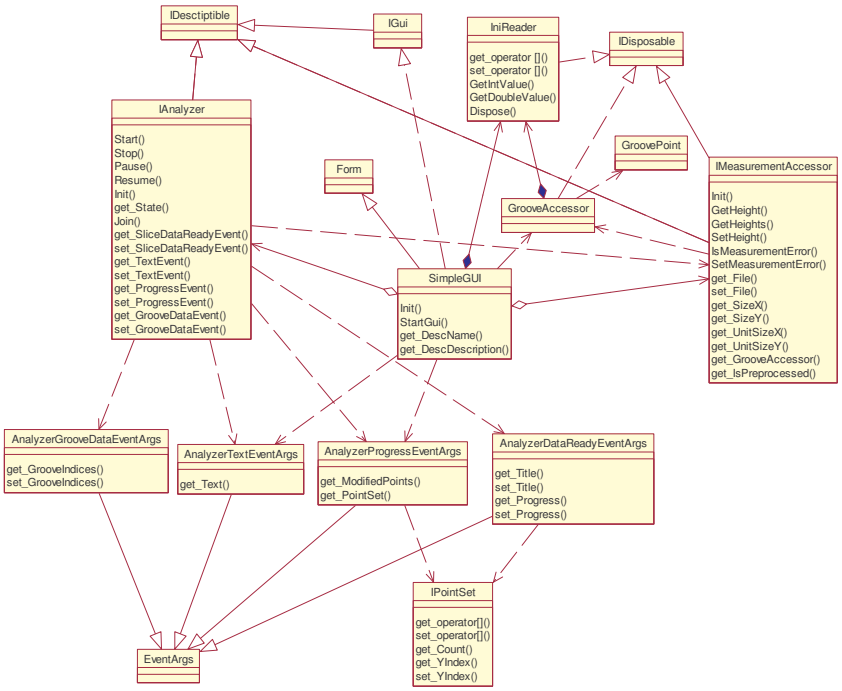


Figure B.5: Class diagram for SimpleUI

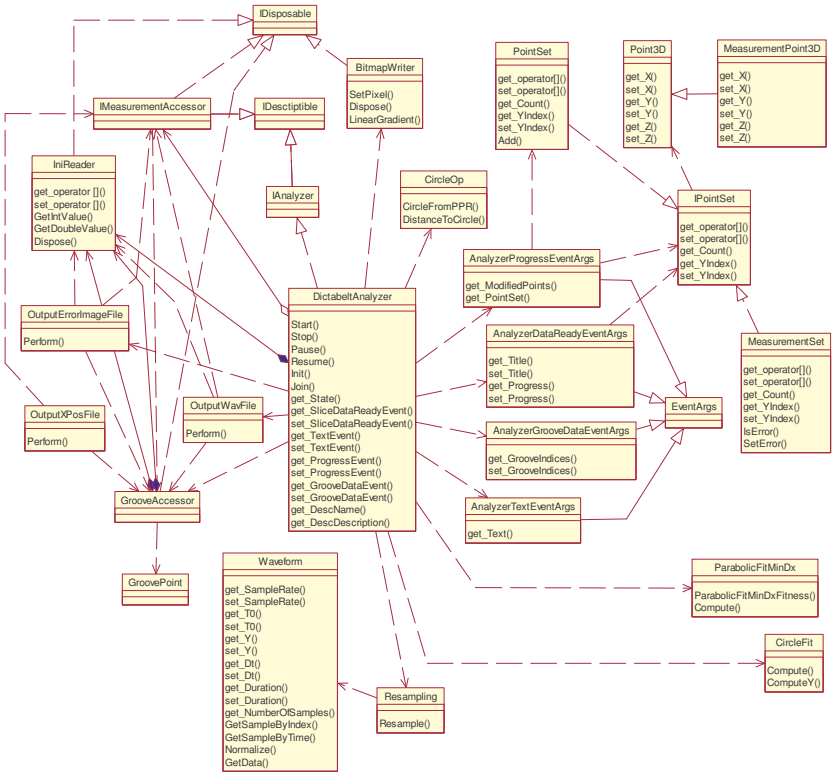


Figure B.6: Class diagram for DictabeltAnalyzer

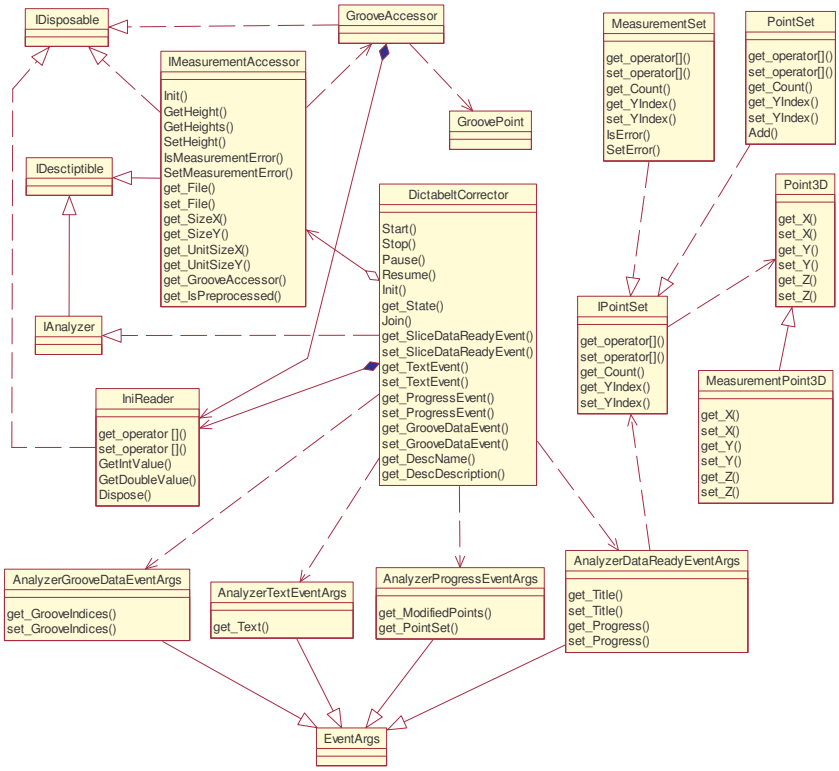


Figure B.7: Class diagram for DictabeltCorrector

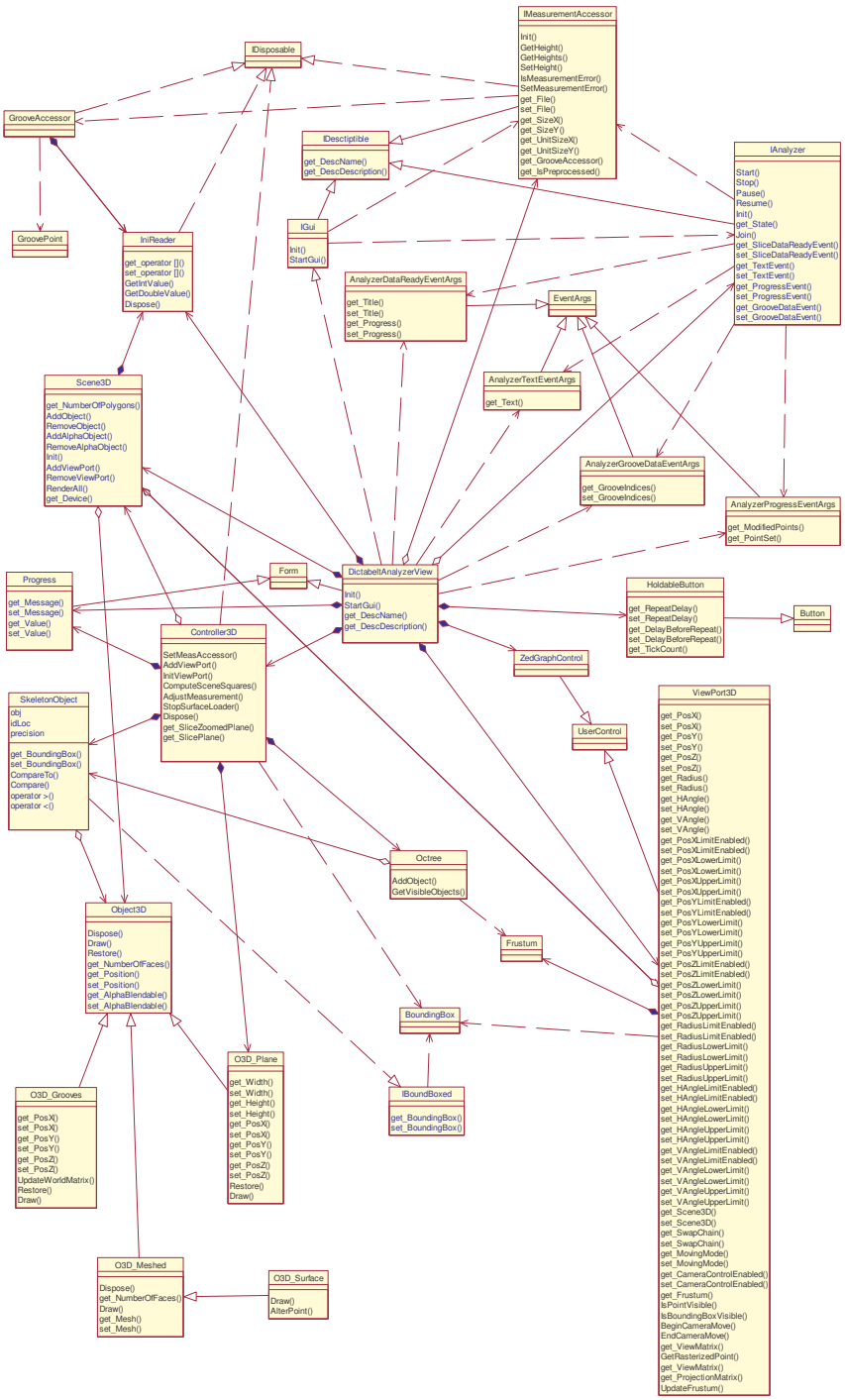


Figure B.8: Class diagram for GUI1

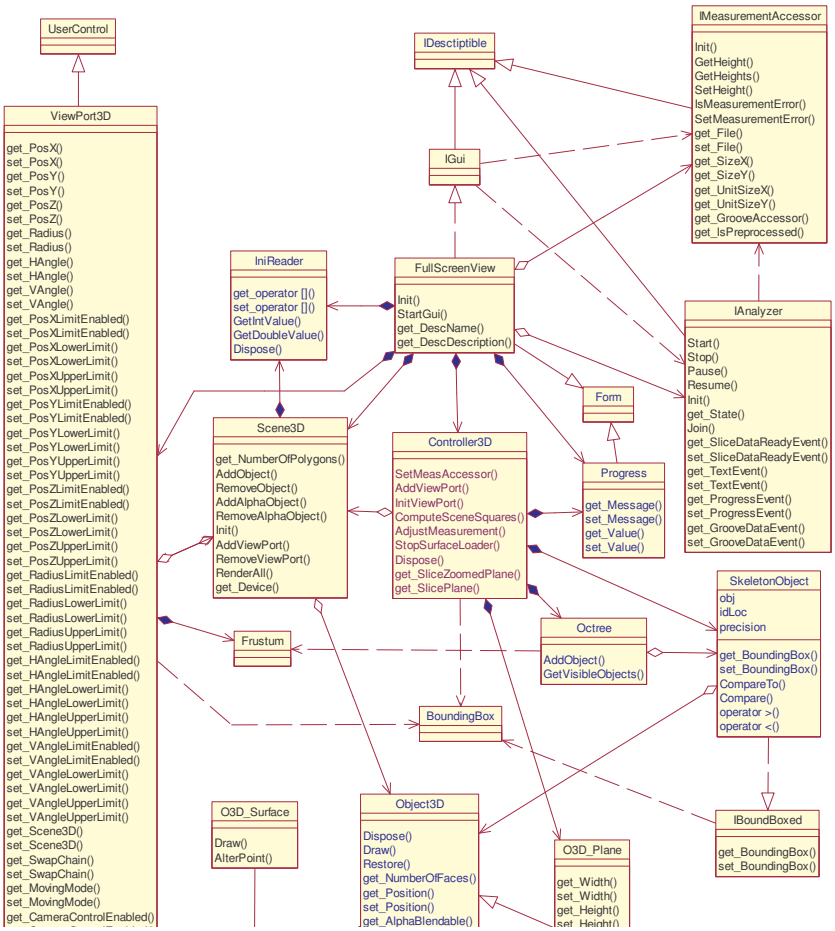


Figure B.9: Class diagram for Full3D

Appendix C

Notes on the 3D engine

C.1 DirectX 9 3D Pipeline

The 3D pipeline is the workflow Direct3D performs to transform a 3D model to a 2D image (see figure C.1). The first step is the transformation and lighting, which is also called "tessellation". After the transformation, all vertices have their final position in the world (not really because an optional vertex shader might move them afterwards). This is achieved by multiplying them all first by the world matrix, which positions the model in the space, rotates it and scales it, and second by multipling all of the resulting vertices by the view matrix, which rotates, translates and scales the whole world. Actually, to improve the performances of the transformation, the view and the world matrix are multiplied together and the vertices are multiplied by the resulting matrix. The lighting is computed using the normal vector of each vertex, which was transformed too. Then a vertex shader might be applied to create sophisticated effects. Now that the final positons of the vertices are known, Direct3D can rasterize them. That means that each vertex is multiplied by the projection matrix which basically defines the field of view. This matrix makes the further object looks smaller and the nearest one bigger. After this operation, the X and Y position of each vertex is exactly its position on the 2D screen and the Z position provides information about its depth, which will be used later to know which models are foreground and which one are background. To improve performances the culling operation aims at removing all the primitives (simple figures discribed by the indices and the vertices) that are offscreen or not visible. The pixel shader takes then care of filing the primitives according their color, lighting and texture and optionnally applying some fancy effects. Finally the primitives are renderend to the backbuffer and the Z-buffer, which contains information about the depth of each backbuffer's pixels to avoid a nearer pixel to be overwritten by a further one.

C.2 Frustum culling

The source code below is used to determine wether a axis aligned bounding box is fully inside, fully outside the viewing frustum or intersects with it. The return value is

- 0 : Fully outside
- 1 : Intersects
- 2 : Fully inside

```
1  public int ContainsBoundingBox(BoundingBox box)
2  {
3      bool intersect = false;
4      int result = 0;
5
6      Vector3 minExtreme;
7      Vector3 maxExtreme;
8
9      for (int i = 0; i < 6; i++)
10     {
11         Plane p = planes[i];
12         if (p.A <= 0)
13         {
14             minExtreme.X = box.XYZLower.X;
15             maxExtreme.X = box.XYZUpper.X;
16         }
17         else
18         {
19             minExtreme.X = box.XYZUpper.X;
20             maxExtreme.X = box.XYZLower.X;
21         }
22
23         if (p.B <= 0)
24         {
25             minExtreme.Y = box.XYZLower.Y;
26             maxExtreme.Y = box.XYZUpper.Y;
27         }
28         else
29         {
30             minExtreme.Y = box.XYZUpper.Y;
31             maxExtreme.Y = box.XYZLower.Y;
32         }
33
34         if (p.C <= 0)
35         {
36             minExtreme.Z = box.XYZLower.Z;
37             maxExtreme.Z = box.XYZUpper.Z;
38         }
39         else
40         {
41             minExtreme.Z = box.XYZUpper.Z;
42             maxExtreme.Z = box.XYZLower.Z;
43         }
```

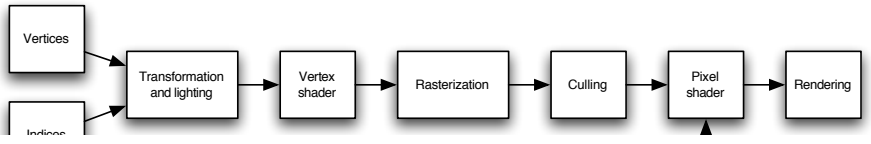


Figure C.1: DirectX 3D pipeline

```
44            if (Plane.DotNormal(p, minExtreme) + p.D < 0)
45            {
46                result = 0;
47                return result;
48            }
49
50            if (Plane.DotNormal(p, maxExtreme) + p.D <= 0)
51            intersect = true;
52        }
53
54        if (intersect)
55            result = 1;
56        else
57            result = 2;
58
59        return result;
60    }
```

Appendix D

Study of Gramophone Records

A test was conducted acquiring data using the 3D scanner to scan gramophone records. The results were not as good as expected. The light intensity reflected from the surface of the record is not enough when the confocal probe scans groove walls. This is with shellac records. On gramophone records the angle of the groove walls is about 45°. Although this is above the limit of the confocal metrology device (the indicated maximum measurable angle is 27°) earlier measurements of records done in the lab showed satisfactory results. After different tests the conclusion is that the light source intensity must be lower than it was before. The light source has caused other problems in the past. The manufacturer suggests that the probe is returned for maintenance.

In another test aluminum records were scanned. Since aluminum reflects light more than regular shellac discs, the light intensity of the reflected light might be strong enough. They indeed seemed to produce acceptable data. With "regular" records the CHR had to run at a low acquisition frequency, e.g. $33Hz$, to get the maximum light intensity. With aluminum records it was pretty pretty much the opposite. Because of the reflecting surface of the aluminum record too much light is reflected when running at low acquisition frequencies. After several tests the conclusion is that $1000Hz$ is a good value for aluminum discs.

The data acquisition process strategy used to scan the disc is the same as explained in [2]: the record is first scanned along the axis (radial) and then the aximuth (rotation) is incremented. This means that the data needs to be reorganized at the end of the scan to get sequential in time. For conventional purposes a scan along the axis is called a "slice". In this experiment a slice is $10mm$ wide, which correspond approximately to 25 grooves. A step along the axis is $5\mu m$ long. That means that 2000 points are acquired for each slice. The scan rotates over the whole disc, i.e. 360°, with an increment of 0.01285°. That corresponds to a sampling frequency of about $30KHz$. The confocal probe was configured to acquire data at $1KHz$. At lower frequencies a lot of noise was measured which is probably due to the aluminum surface that scatters the light.

The shape and geometry of the groove on aluminum discs are slightly different than the ones on other records. Usually the groove-to-groove distance is about $400\mu m$ whereas the groove depth is $60\mu m$ and the groove width (on

the top) is $150\mu m$. On the data taken from the aluminum disc, the groove is between $9\mu m$ and $15\mu m$ deep, depending on how you see the groove, $125\mu m$ wide and the groove-to-groove distance is about $250\mu m$. Figure D.1 shows a portion of a disc slice. It represents the points measured by the confocal probe. Note that the points are equidistant from each other. With some experience, one can "guess" the groove shape. Aluminum records are embossed using a cutting tool. The matter that was prior in the groove is "pushed" out and some of it stays on top of the groove and forms a hill. These hills are very well seen in the figure. Note that the groove on aluminum discs is a little narrower and especially less deep. Hence the slope of the groove walls is smaller. This might also explain why data could be acquire from the aluminum records.

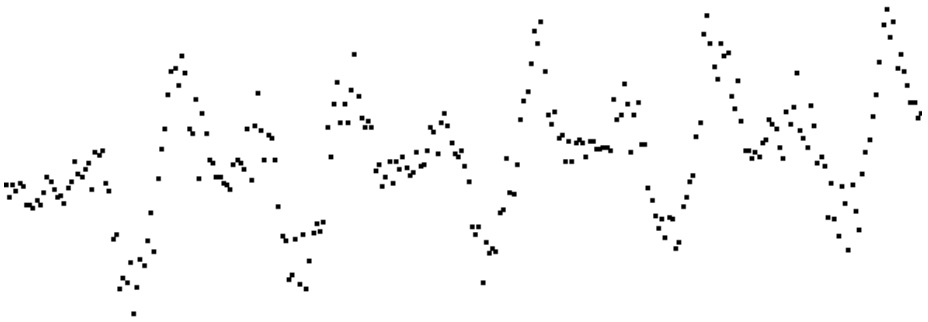


Figure D.1: Acquired data (points) from a aluminum record that represents five grooves within a slice. *Note that the points are equidistant from each other.*

Appendix E

Noise Measurement

E.1 Static Confocal probe

In order to measure the noise of the confocal probe the linear stage and the rotational stage were stationary. Two different positions on the dictation belt were considered for that measurement: a region with a certain slope, e.g. in the middle of a groove wall, and a flat region, e.g. between two grooves. Since the intensity of the measured light that is coming back into the CHR is less on regions with slope, the quality of the measurement might be affected. To position the linear motor at the right place, a LabView program was designed that plots measurement points in real time and lets the user move the linear stage to the right position. Using this tool it is fairly easy to position the probe to measure a groove wall or the top of a groove. In this measurement the points on the steep region were acquired from a groove wall which has a typical slope of approximately 27%. Once the linear motor was positioned, measurements of the same location were taken using different probe rates. The acquired measurement points are normally distributed around some mean value. Table E.1 shows the standard deviation in micrometer of the measurement points for the different locations and probe rates. For each measurement 10000 points were acquired.

Table E.1: Noise measurement for confocal probe.

Location	Probe rate $[kHz]$	Standard deviation $[\mu m]$
Flat region	0.4	0.0204
	1.0	0.0311
	2.0	0.0631
	4.0	0.1480
Steep region	0.4	0.0348
	1.0	0.0827
	2.0	0.1261
	4.0	0.1942

The values from Table E.1 suggest that the precision of the measured points

with the confocal probe varies with the probe rate. Lower probe rates such as 400 measurements a second give better results than higher rates. This is probably due to the fact that the light intensity that is going back to the CHR lowers as the probe rate gets higher. For the same reason a smaller standard deviation is measured on a flat surface compared to a surface with a certain slope. If the measured surface is not flat, the light intensity that goes back to the CHR is lower because the light is partially reflected away from the probe.

E.2 Moving Confocal probe

The previous section shows the measurement procedure that was used to identify the noise of the confocal probe when the probe is not moving. When the probe is moving two additional noise sources must be considered: The linear stage and the trigger that tells the confocal probe when to measure.

The linear stage is configured to move at constant velocity while the confocal probe is taking data at a constant rate. Measurement points are considered to be equally distant from each other, which is only accurate if the linear stage speed is constant. Since the linear stage is a servo motor its position, and speed, are constantly corrected using a PID controller. This helps to achieve a constant velocity but small fluctuations in the speed might still introduce errors at every measurement point since in the analysis the points are considered to be equally distant.

The trigger is sent from the motion controller to the confocal probe to start the data gathering whenever the linear stage is crossing a certain position. If the trigger is not precise, the probe will not always start the measurement at the same place. This would introduce a certain offset between consecutive data slices.

A measurement was set up to measure the combined error and noise for the three error sources: The linear stage, the trigger and the confocal probe. For that measurement, the same data slice was measured 500 times. That means that the rotational axes did not move at all. Each slice contains 2000 measurement points, which represents $10mm$ width on the dictation belt since each point is spaced by $5\mu m$. Several measurement points from flat regions and points from steep regions where picked out of the 2000 measurement points. For each of these points the standard deviation was computed over the 500 slices. Tables E.2 and E.2 shows the standard deviations obtained. The position represents the index in the 2000 point array and the values represent the standard deviation of the measurement in micrometer.

Measurements taken at $400Hz$ have a bigger standard deviation than measurements taken at $1000Hz$. This is not the case if the confocal probe is stationary. Since the probe is averaging the surface height over a distance of $5\mu m$ when moving, the light intensity is probably saturating at $400Hz$ which would explain the loss in precision. The best results are achieved at $1000Hz$.

Table E.2: Noise measurement in μm on steep regions for confocal probe in motion.

	Probe rate $[Hz]$			
Position	400	1000	2000	4000
88	0.26893	0.07173	0.23286	0.33383
110	0.48712	0.07145	0.16945	0.49850
212	0.43732	0.07572	0.18143	0.31779
315	0.65103	0.08875	0.24589	0.97616
569	0.40861	0.08346	0.39410	0.51620
672	0.55942	0.09619	0.17879	0.62353
646	0.52661	0.07468	0.20491	0.55398
638	0.25921	0.08674	0.11869	0.17156
620	0.53756	0.07940	0.16410	0.28876
596	0.43672	0.09685	0.40032	0.34072
570	0.88184	0.10250	0.25143	0.54005
442	0.35652	0.13397	0.30987	0.63991

Table E.3: Noise measurement in μm on flat regions for confocal probe in motion.

	Probe rate $[Hz]$			
Position	400	1000	2000	4000
119	0.13210	0.06564	0.10292	0.16859
172	0.17258	0.06409	0.11599	0.20330
273	0.19688	0.09565	0.11749	0.17562
325	0.15122	0.07993	0.14851	0.15216
349	0.13296	0.07962	0.11793	0.14234

Appendix F

Data Acquisition File Format

The data acquired with the LabView program are stored in binary files using the SGL format, which is the standard way to write binary data with LabView. For historical reasons the number in SGL files are stored using the Big Endian format. In Big Endian format, the most significant byte (MSB) of a multi-byte number is written first, then the second MSB, and so on down to the least significant byte (LSB). The header as well as all the data is stored as SGL precision numbers (4 Bytes).

The header of a data file is structured like follows:

Offset	Description
$0x00$	Scanning increment (μm)
$0x04$	Distance to scan (mm)
$0x08$	Scan starting position (mm)
$0x0c$	Angular increment ($^\circ$)
$0x10$	Number of angular steps

The *scanning increment* corresponds to the distance between two measurement points in a slice along the dictation belt axis. The *distance to scan* is the distance in millimeters along the dictation belt axis the confocal probe is taking measurements. The number of points taken along the axis is given by $\frac{distancetoscan*1000}{scanningincrement}$. The position on the belt where the scan starts is the *scan starting position* given in millimeters. This value is only relevant for a certain linear stage setup since this value represents a position relative to the home position of the linear stage motor. The *angular increment* gives the step in degrees of the rotational stage taken between each slice. The *number of angular steps* represents the number of slices measured. Usually the *number of angular steps* times the *angular increment* is equal to 360 °.

The rest of the file contains one measured slice after another. The measurement point that was measured first in the slice is stored first in the file. All the measurement points are therefore stored in the order they are measured.

Appendix G

File Listing

This chapter lists all the LabView VI's, the important data files and extracted wav files.

G.1 LabView Files

G.1.1 Data Analyzer

Filename: **Analzyer4.vi**

This is the first implementation of the data analysis software. It basically goes through the data slice per slice, finds interesting regions that might contain a groove bottom, fits a sixth polynomial function to each interesting region and takes the minimum of that sixth polynomial function as the groove bottom. The detected groove bottoms are then linked together to form a consecutive signal. The resulting signal is filtered, resampled, scaled and converted to a wav file. For more details about the data analysis process please refer to Chapter 4.

G.1.2 Output Sine Waves

Filename: **OutputSineWaves.vi**

This VI is used to generate sine waves and output the signal to the sound card that converts it to an analogue signal (PXI 4461). The output is a series of sine waves at different frequencies. The user can indicate the lowest and highest frequency. The program will generate sine waves starting at the lowest frequency and doubling the frequency at each step until the higher frequency is reached. Silence of 0.5 seconds is inserted between each frequency. The amplitude, sampling frequency and the duration of the signal can be specified as well.

This VI was used to generate test patterns that were recorded on dictations belts and scanned with the optical scanner afterwards.

G.1.3 Output Swept Sines

Filename: **OutputSweptSine.vi**

This VI generates a swept sine signal used by the National Instruments Sound and Vibration Toolkit, which can be used to measure for example the harmonic distortion caused by the dictation belt recording machine. It basically generates a sine wave that is continuously increasing its frequency. The lowest and highest frequency, the sampling frequency and the output voltage can be specified in the VI.

This VI was used to generate test patterns that were recorded on dictations belts and scanned with the optical scanner afterwards.

G.1.4 Signal Acquisition With Sound Card

Filename: **CompareSignalsDAQ.vi**

This VI's reads the analog input of the acquisition card (PXI 4461) and writes the data into a SGL binary file. No filtering is applied. A small header is written containing two floating SGL files. The first is the sampling frequency and the second is the number of samples acquired.

This VI was used to measure the speaker output, the signal applied at the recording needle, and the signal measured at the playback needle. Wav files can be generate from data acquired with that VI using the SignalFiltering2 VI. The data file has to be previously converted using the Cmp2Xpos VI.

G.1.5 Compare Signals

Filename: **CompareSignals.vi**

With this VI the user can compare different waveforms that were either extracted with the optical method or measured using the CompareSignalsDAQ VI. Up to three waveforms can be loaded and compared. The offset and amplitude of each waveform can be modified in order to superpose them. The spectrum of certain regions of the waveform can also be compared, by indicating the regions of interest. If the user wants to compare a file extracted from the analyzer he needs to add a special header to the file that contains two values: the first value is the sampling frequency and the second is the number of samples.

G.1.6 Sound Filtering and Wav Generation

Filename: **SoundFiltering2.vi**

This VI was used before the C# code was able to output a wav audio file. The C# analyzer would output a file called XPos file that is an SGL binary file that contains the position of the groove bottom at any given time. This VI removes the slope of the input signal, it passes the waveform though a bandpass filter, resamples the signal, scales it and saves the result as a wav file at $11025Hz$.

G.1.7 Data Acquisition

Filename: **....vi**

This is the main data acquisition VI. The user has to first configure the measurement. Once the measurement is configured it can be run. This VI will output a data file that contains the measured distances by the confocal probe.

The format of the file is described in details in Chapter F. The structure of this VI is presented in Chapter 3.

G.1.8 Additional VI's

Other VI's have been used to measure the noise level of different measurements, measure the total harmonic distortion of a signal or simply perform some basic operations used in VI's described earlier. These VI's are not described more in detail here.

G.2 Optical Measurement Files

Several data files were acquired with the optical method during the project. This sections lists all these files and indicates their names, the configuration parameters and their purpose. The measurements are listed in their chronological order. Note that for all measurements the distance between two measurements along the axis is $5\mu m$. The sampling frequency indicated is the sampling frequency of the corresponding sound on the belt:

$$f_s = \frac{num\ angular\ increments \cdot rpm}{60s} \tag{G.1}$$

For example if 18000 samples were taken, the sampling frequency would be $\frac{18000 \cdot 42}{60} = 12600Hz$. The CHR sampling rate is the number of samples what were taken per seconds with the confocal probe. The recording amplitude mentioned in the configuration parameters corresponds to the amplitude of the signal outputted by the sound card (PXI 4461) that was then recorded using the dictation belt recorder. The typical sound patterns recorded on dictation belts are eight two second long sine waves with frequencies between $50Hz$ and $6.4kHz$. The frequency is doubled at each frequency change ($50Hz$, $100Hz$, $200Hz$, ..., $6.4kHz$). Between each frequency a $0.5s$ of silence was recorded. On some belts there is a swept sine recorded as well.

G.2.1 Men Talking $12.6KHz$

Filename: **DB-12K-7mm-5um-20050829.dat**

This file is the first measurement. It is an old belt on which an insurance guy is talking. This measurement was made to make sure the measurement process and the data acquisition is working properly.

Configuration parameters

Parameter Name	Value
Sampling frequency	$12.6KHz$
CHR sampling rate	$1000Hz$
Distance to scan	$7mm$
Angular increment	$0.02°$
Number of angular increments	18000
Date	29 August 2005

G.2.2 High Amplitude Sine Waves

Filename: **DB-12K-11mm-5um-20050902.dat**

This file contains the recording of sine waves at different frequencies. The amplitude used to record these signal was $1V$, which is much higher than the amplitude generated by a regular microphone. This was not known at the time this belt was recorded. The dictation belt recorder has an automatic gain control that reduced automatically the amplitude of the signal.

Configuration parameters

Parameter Name	Value
Sampling frequency	$12.6KHz$
CHR sampling rate	$1000Hz$
Distance to scan	$11mm$
Angular increment	$0.02°$
Number of angular increments	18000
Recording amplitude	$1V$
Date	2 September 2005

G.2.3 Lower Sine Waves at $25.2KHz$

Filename: **DB-25K-6mm-5um-20050909.dat**

This file was recorded at a lower amplitude using $50mV$, which is approximately the output voltage generated by the dictation belt recorder microphone when someone is dictating something. The test pattern is composed of sine waves at different frequencies with a swept sine.

Configuration parameters

Parameter Name	Value
Sampling frequency	$25.2KHz$
CHR sampling rate	$1000Hz$
Distance to scan	$6mm$
Angular increment	$0.01°$
Number of angular increments	36000
Recording amplitude	$50mV$
Date	9 September 2005

G.2.4 High Sampling Frequency

Filename: **DB-50K-5mm-5um-20050916.dat**

The same dictation belt that was scanned at $25.4kHz$ was scanned again at a higher sampling frequency. In order for the scan to finish in one weekend the CHR sampling rate had to be raised to $2kHz$ instead of $1kHz$. The number of measurement errors is higher at a higher sampling rate. There is a tradeoff between the sampling frequency and the quality of the measurement since the time for a scan is limited.

Configuration parameters

Parameter Name	Value
Sampling frequency	$50.4KHz$
CHR sampling rate	$2000Hz$
Distance to scan	$5mm$
Angular increment	$0.005°$
Number of angular increments	72000
Recording amplitude	$50mV$
Date	16 September 2005

G.2.5 High Sampling Frequency Lower Gain

Filename: **DB-50K-5mm-5um-20050930.dat**

Sine waves at different frequencies were recorded with a lower gain. The gain of the external microphone can be adjusted with a knob on the dictation belt recorder. This belt did not show good results. The noise level was too high compared to the level of the sound signal.

Configuration parameters

Parameter Name	Value
Sampling frequency	$50.4KHz$
CHR sampling rate	$2000Hz$
Distance to scan	$5mm$
Angular increment	$0.005°$
Number of angular increments	72000
Recording amplitude	$50mV$
Date	16 September 2005

G.2.6 High Sampling Frequency Lower Gain

Filename: **DB-50K-5mm-5um-20050930.dat**

Sine waves at different frequencies were recorded with a lower gain. The gain of the external microphone can be adjusted with a knob on the dictation belt recorder. This belt did not show good results. The noise level was too high compared to the level of the sound signal.

Configuration parameters

Parameter Name	Value
Sampling frequency	$50.4KHz$
CHR sampling rate	$2000Hz$
Distance to scan	$5mm$
Angular increment	$0.005°$
Number of angular increments	72000
Recording amplitude	$50mV$
Date	30 September 2005

G.2.7 High Sampling Frequency Lower Gain

Filename: **DB-50K-4mm-5um-20051009.dat**

The waveform recorded on that belt has been played once before it was scanned with the optical method. While it was played back with the dictation belt player the data acquisition card (PXI 4661) recorded the voltage at the playback needle. The filename of the output waveform from the reading needle is: **needle-50mV-Scan-out.dat**.

Configuration parameters

Parameter Name	Value
Sampling frequency	$50.4KHz$
CHR sampling rate	$2000Hz$
Distance to scan	$4mm$
Angular increment	$0.005°$
Number of angular increments	72000
Recording amplitude	$50mV$
Date	9 October 2005

G.2.8 Men Talking $50.4kHz$ First Try

Filename: **DB-50K-5mm-5um-20051014.dat**

This was a first try to measure the old dictation belt. The computer crashed in the middle of the measurement. This file still contains some data but now a whole measurement.

Configuration parameters

Parameter Name	Value
Sampling frequency	$50.4KHz$
CHR sampling rate	$2000Hz$
Distance to scan	$4mm$
Angular increment	$0.005°$
Number of angular increments	72000
Recording amplitude	$50mV$
Date	14 October 2005

G.2.9 Men Talking $50.4kHz$ Second Try

Filename: **DB-50K-5mm-5um-20051018.dat**

Measurement of the man talking.

Configuration parameters

Parameter Name	Value
Sampling frequency	$50.4KHz$
CHR sampling rate	$2000Hz$
Distance to scan	$5mm$
Angular increment	$0.005°$
Number of angular increments	72000
Recording amplitude	$50mV$
Date	18 October 2005

G.3 Needle and Speaker Measurement

The dictation belt machine was modified in order to be able to measure the voltage at the recording needle, playback needle and the speaker output. These files were acquired with the sound card (PXI 4461) using the same sampling frequency used by the optical method ($50.4kHz$). All the data files store the measured values in SGL binary format (see Chapter F for more information on SGL binary files). The first two values in the file are the sampling frequency and the number of samples measured.

All these files contain the same, usual test patterns: ten sines waves at different frequencies from $50Hz$ to $6.4kHz$ recorded one after another. The frequency is doubled after each sine wave. Each sine wave was recorded for two seconds with half a second between each frequency change.

G.3.1 Recording Needle

Filename: **needle-50mV-Scan-in.dat**

This file was recorded by measuring the voltage around the recording needle while recording test patterns using the data acquisition card (PXI 4461).

G.3.2 Playback Needle

Filename: **needle-50mV-Scan-out.dat**

This file was recorded by measuring the voltage around the playback needle while playing a belt that contains test patterns.

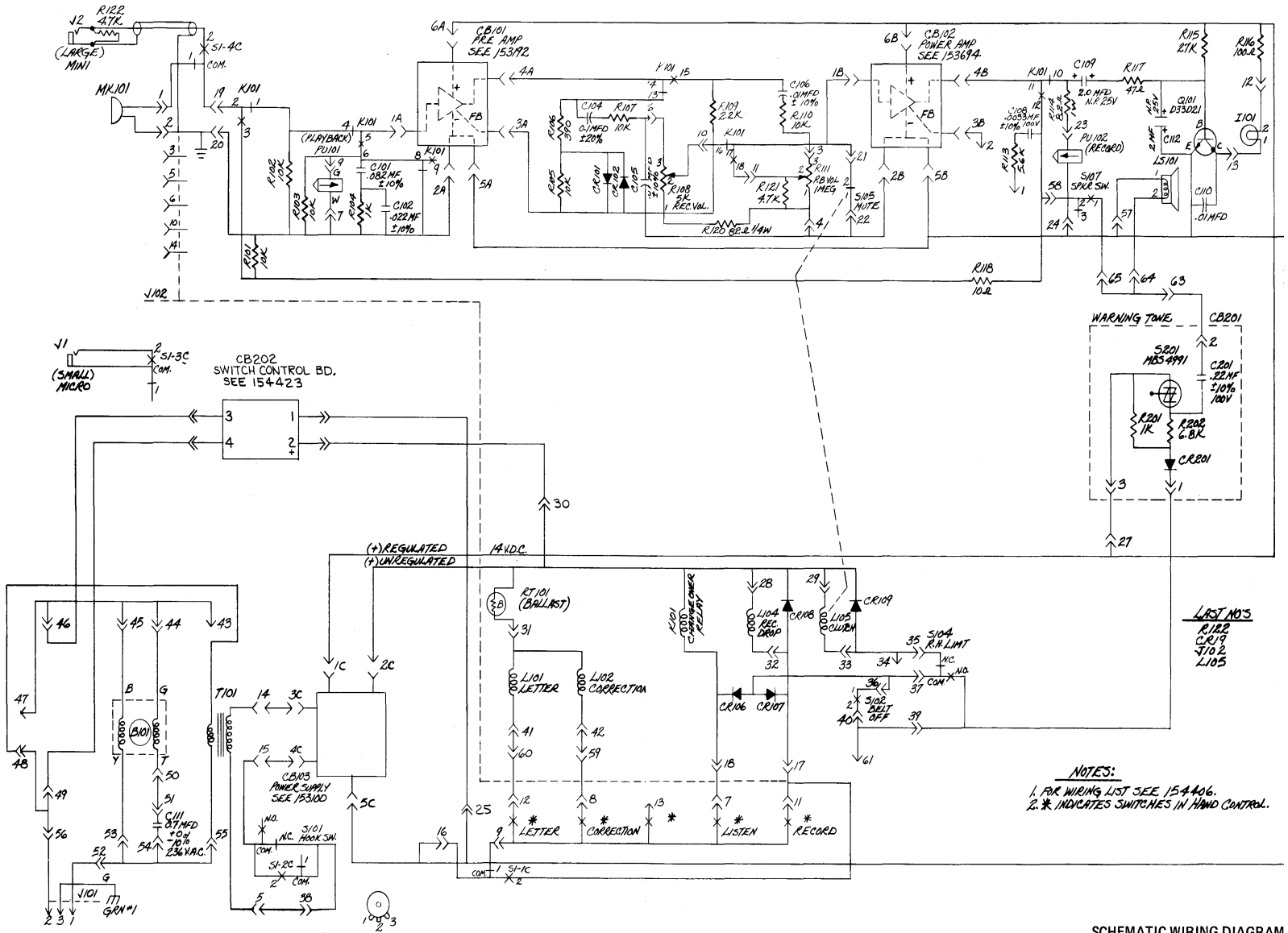
G.3.3 Speaker output

Filename: **speaker-50mV-Scan-out.dat**

This file was recorded by measuring the speaker output of the dictation belt player while playing test patterns using the data acquisition card (PXI 4461).

Appendix H

Dictation Belt Machine Schematics



SCHEMATIC WIRING DIAGRAM
8811 MODEL #154400
SD-1952 ISSUE 1 2-10-7E

Appendix I

Hardware Specifications