Development of an Optical Positioning System for 3D Ultrasound

A Thesis submitted to the faculty

of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for

The Degree of Master of Science

in

Electrical and Computer Engineering

by

Carsten Poulsen

October 2005

APPROVED

Prof. Peder C. Pedersen, Major Advisor

Prof. R. James Duckworth, Committee member

Prof. Brian King, Committee member

Prof. Fred J. Looft, Department Head

Abstract

Ultrasound has developed from 2D into 3D ultrasound in recent years. 3D ultrasound gives enhanced diagnostic capabilities and can make it easier for less trained people to interpret ultrasound images. In general there are two ways of getting a 3D ultrasound image : By using a 2D array scanner (giving 3D images directly) or by using a series of 2D scans and combine these scans to build a 3D volume. The only practical scanning technique that can be used for portable systems is freehand scanning that combines a series of 2D images.

3D images acquired by using a conventional ultrasound transducer and the freehand scanning technique are, however, often misaligned laterally and have unevenly spacing. These errors can be corrected if the position associated with each 2D image is known. Commercially available positioning systems use magnetic or optical tracking, but these systems are very bulky and not portable.

We have proposed another way to get the position by tracking on the skin surface. This is done by obtaining digital images of the surface at a very high rate and then cross correlating each image to reveal the change in position. Accumulating these changes will then give the correct location (in two dimensions) relative to a starting point. Correct volume and surface rendering can therefore be achieved when a scan is done.

A custom-made housing was made to mount an optical sensor to the ultrasound transducer. The optical sensor was placed in the housing and the hardware circuit from an optical mouse was used to interface to a USB interface. An implementation with an optical fiber was also made since this could fit easily to the transducer handle.

In Windows a custom-made mouse driver was used to extract the position information from the sensor. This driver allowed multiple mouse devices in the system and removed the acceleration of the mouse, giving a correct transfer of the position.

A DLL (Dynamic Link Library) was used to interface to a 3D ultrasound software called Sonocubic. Using the DLL and a custom modified version of Sonocubic 3D construction software has allowed a correct compensation of the acquired ultrasound images.

To validate the accuracy of the optical sensor an optical mouse was placed in an XY-recorder to compare the acquired position with the actual position. The test revealed that the accuracy of the optical sensor is very high. A 55 mm movement of the sensor gave a deviation of 0.56 mm which is well within the expected result.

A computer generated phantom was made to see if the compensation algorithm was working. The test revealed that the compensation algorithm and the software is working perfectly. Next a vessel phantom was scanned to see that the compensation algorithm (lateral compensation) was working in real life. The test showed that a correct lateral compensation was made. Finally 3D phantoms were custom made to test the accuracy of the system by estimation of a known volume. The system was able to estimate the volume in a phantom within an accuracy of 6 %.

Performance of the system with direct imaging, using the optical sensor and a lens, was compared to an implementation with an optical fiber, two lenses and the optical sensor. The optical fiber was difficult to implement since the image contrast was degraded severely through the optical fiber and the lenses. This made it difficult for the correlation algorithm to function correctly and tracking could therefore not be done on a skin surface.

Code for an FPGA was made in VHDL to extract the actual images from the optical sensor and display them directly on a computer screen. This was necessary to see how well the sensor was in focus. This proved to be a really useful tool for adjusting the optical system for maximal contrast.

The optical tracking on a skin surface is a good way to assist a user doing a freehand scanning to get images without geometric distortion. Furthermore, it is the only real positioning system for a portable system. One requirement for this system is, however, that the object being scanned is flat and does not curve or vary vertically. For most applications this is not the case, and we are therefore proposing an implementation with microgyros that is able to give angle information as well. This would give the system a total of up to 5 instead of just 2 degrees of freedom. The status of this is currently that it can be easily implemented in the DLL, but it is not implemented in the 3D reconstruction software, Sonocubic.

Acknowledgements

I would first of all like thank my advisor, Prof. Peder C. Pedersen, for his big effort in helping me with this thesis. He has put many hours of work into this and I sincerely hope that the results from my work will benefit the ultrasound lab and WPI in the years to come.

For the development of the interface between the 3D positioning system and Sonocubic and for implementing the interpolation algorithm, I would like to thank Ricardo Gayoso for his big contribution to this work.

For the development of the ultrasound phantoms I would like to thank Matt Rowan for his excellent work on making really nice phantoms. These phantoms have been invaluable in evaluating the performance of the positioning system described in this thesis.

As an introduction to this thesis we developed a portable ultrasound system that was demonstrated at the ATA (American Telemedicine Association) meeting in Tampa, May 2005. I would like to thank both Prof. Peder C. Pedersen and Prof. R. J. Duckworth for their support in this work and for the many enjoyable hours we had demonstrating this system.

Finally I would also like to thank both Prof. Thomas L. Szabo from Boston University and Prof. Peder C. Pedersen for coming up with the idea for this thesis and for their contribution to the IEEE Ultrasonics Symposium 2005 in Rotterdam, where we presented a paper about the positioning system.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Fast treatment is critical in saving people's life when they are injured. Triage is therefore of the highest importance whether it is on the battlefield, on the road in a car accident or in an emergency room. Ultrasound can be used for this purpose, but often the ultrasound equipment is bulky and difficult to carry around. Some small systems do, however, exist on the market, which makes it possible to use ultrasound at places it is difficult to access. An example of such a system is the Terason 2000 ultrasound scanner shown in Fig. 1.

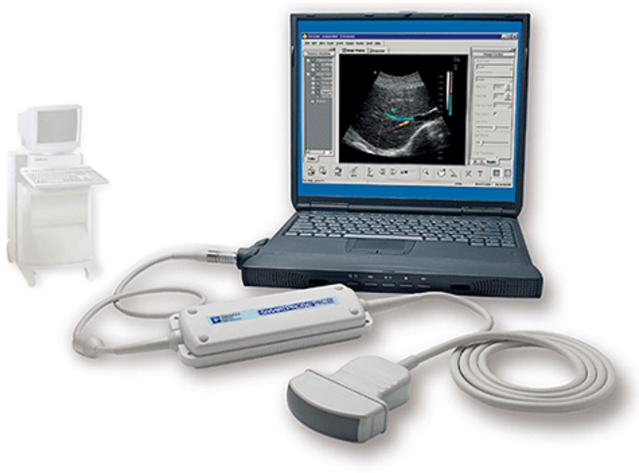


Fig. 1 - Terason 2000 ultrasound system

As part of my work from January to May 2004 here at WPI I was developing a portable ultrasound system for TATRC (Telemedicine and Advanced Technology Research Center), the so called 2nd generation portable ultrasound system. This was done with Prof. Peder C. Pedersen and Prof. R.J. Duckworth. The work resulted in an ultrasound system integrated into a vest (see Fig. 2) that it was possible to carry around on the battlefield or in an ambulance.

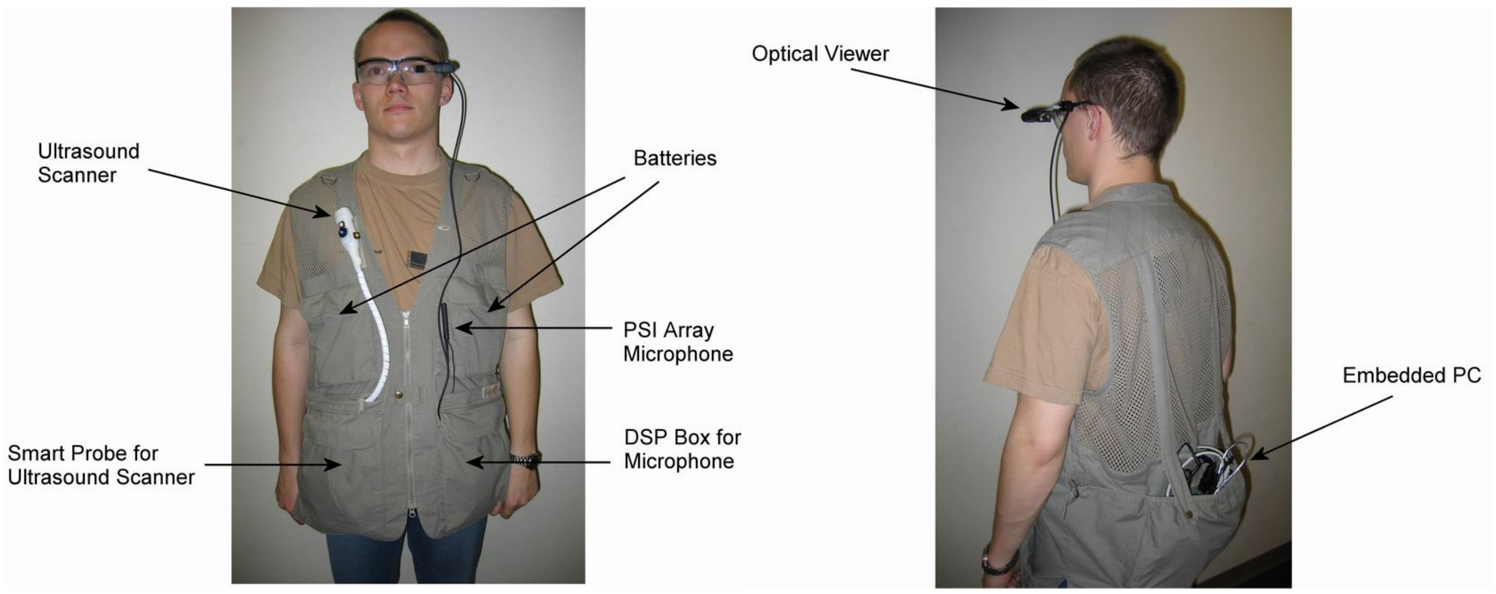


Fig. 2 - Portable 2nd generation ultrasound system

The system is based on the Terason 2000 ultrasound scanner. To this we have added a small embedded PC that is placed in a pocket in the back. A microphone is used together with a speech recognition system that was customized for this system by Dalys Sebastian [1]. Finally there is an optical viewer through which it is possible to see the ultrasound image that is acquired by system. The work presented in this thesis is a continuation of the results we achieved with this system. In short, the idea is to add 3D capabilities to the portable system.

Traditionally experienced ultrasonographers have visually imagined how a 3D representation of a scanned object or pathology looks based on a series of 2D ultrasound scan planes put together in their head. We took this idea a step further and provide an actual 3D representation on a computer screen, making it easier for less trained people to interpret ultrasound data. The 3D representation of the data gives a lot of redundancy and makes it easier for the mind to visualize what is scanned. This makes it possible to get a fast diagnosis of a patient.

There are a number of different methods that can be used to obtain a 3D ultrasound image. One of these is shown in Fig. 3. As it can be seen this implementation needs a large mechanical motor to move the transducer, which makes it a bit impractical. There are, however, implementations where the motor is built into the transducer itself. Another problem is that the size of the volume that can be scanned is limited to the volume that is swept by the transducer when it is rotating. This limits the ultrasound scans to small volumes.

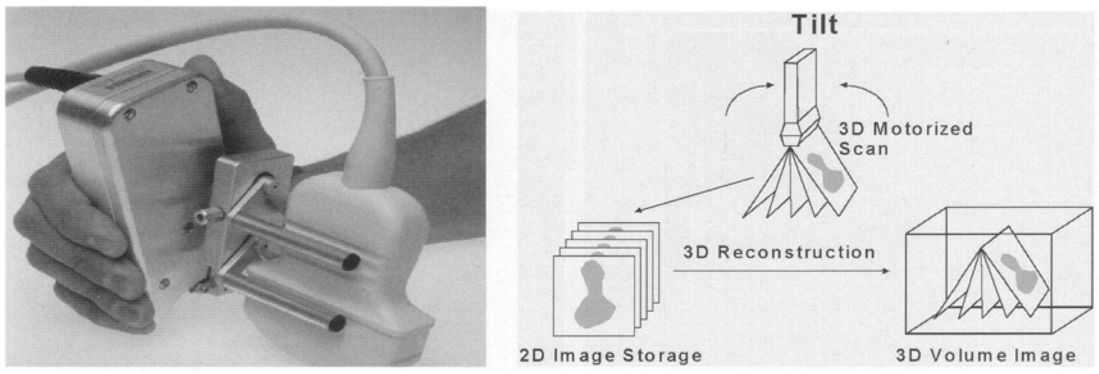


Fig. 3 - 3D Rotational Motorized Scan [4]

In recent years a new technique using a 2D array transducer has been developed. The idea is the same as with the mechanical rotation, but instead of having a motor mechanically move the transducer, the direction of the ultrasound beam is steered electronically. The advantage with this method is that it is very fast, which makes it possible to capture 3D ultrasound images in real time - also called 4D ultrasound. But the same limitation exists for the field of view as described above. Another limitation in terms of portable ultrasound is that the available 3D implementations on the market, which control the sweep of the scan planes, are not particularly suitable for portable ultrasound.

The only real solution on the market for portable 3D ultrasound is to use a scanning technique called free-hand scanning. The goal is to move the ultrasound transducer in a straight scan path with a constant speed, as if the transducer was mounted in a motor as shown in Fig. 4. In practice, a straight scan path at a constant speed is difficult to achieve.

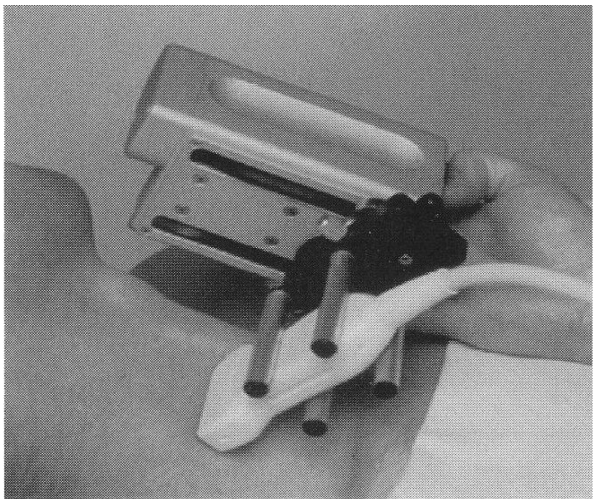


Fig. 4 - Ultrasound transducer mounted in a motor driven linear fixture [4]

For practical purposes the motor is unwanted and instead a position system can be used to determine the position of the transducer. By using the position information the necessary compensations can be made and a correct 3D image can be obtained. How this can be done will be described in detail in this thesis.

The position systems used today are bulky and impractical, especially for portable ultrasound [2],[3]. An example called "Flock of Birds" is shown in Fig. 5. As it can be seen, the position system is quite big, and it requires a number of fixed devices to be placed in the room where the scanning takes place, making it far from portable. The "Flock of Birds" gives 6 degrees of freedom.

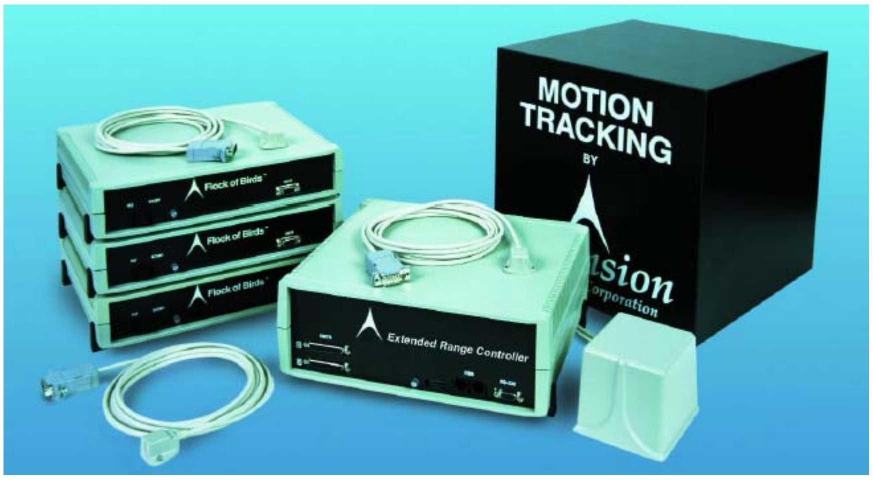


Fig. 5 - Flock of Birds by Ascension Technology [2]

We therefore decided to develop a new and much smaller position system to assist a user trying to make a free-hand ultrasound scan. The system is based on an optical sensor that is found in optical mice. The sensor is tracking on the skin surface and gives 2 degrees of freedom. In the following chapters it will be described how this system was developed and interfaced with the ultrasound scanner software that we used. Finally the system's ability to do volume estimations and visualize scanned objects will be shown.

1.2 Overview of this thesis

Chapter 2 - System overview : This chapter gives an introduction to how this system is designed (and implemented). The idea is to give a basic understanding of the system before describing the details in the following chapters.

Chapter 3 - The Optical Position Sensor : This chapter describes the basic functionality behind an optical mouse and the accuracy of it. It is described how images can be extracted from the sensor for visualization, which is important for verifying that the images captured by the optical sensor are in focus. Two different implementations, one with an optical fiber and one without, is tested. Finally a comparison between the different implementations is made.

Chapter 4 - Software and interfacing : This chapter describes how the optical sensor was interfaced to the software. It also describes how the data transfer is made between the applications used. Finally a test is made to see if the system is working on computer generated image data.

Chapter 5 - Volume Estimations : In this chapter the complete system is tested in a real life test. This is done on a customized 3D ultrasound phantom that was created in our lab. The system is tested for it's ability to do volume estimations and to make a 3D reconstruction.

Chapter 6 - Conclusion : This chapter contains the conclusions for the work presented in this thesis and gives recommendations for future work.

The appendix part of this thesis contains :

Appendix A - Detailed information about the CCD-array viewer, which is used for capturing the image that the optical sensor is photographing.

Appendix B - The code that was made for interfacing the optical sensor to the software.

Appendix C - The Matlab algorithm that was used for filtering and calculating the volume of the tested ultrasound phantom.

Appendix D - The customized Teratest code for testing

Appendix E - The customized Pos3D.dll code for testing

2 System Overview

2.1 Introduction

As I expect the reader to be quite unfamiliar with the architecture of the system described in this thesis I have made this section to provide a short overview of it. This should make it easier to see the "big picture" in this thesis.

The development of an ultrasound system with 3D capabilities is described in this thesis. The system is based on a PC and a Terason 2000 ultrasound transducer that is commercially available at a price around $30,000. The Terason 2000 ultrasound system is shown in Fig. 6.

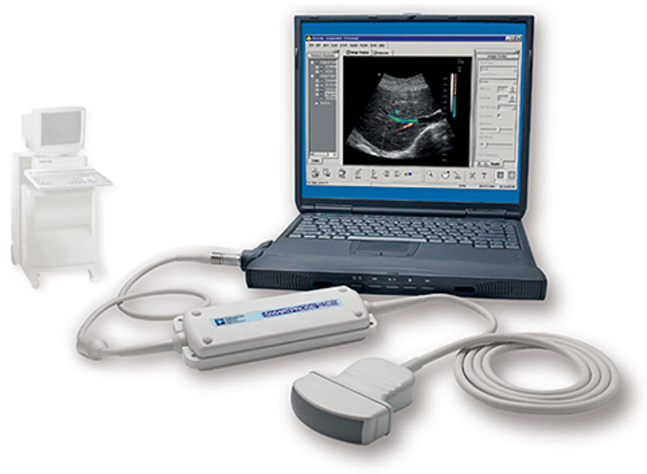


Fig. 6 - Terason 2000 Ultrasound Scanner connected to laptop

We used a Terason 2000 system to developed a second generation wearable ultrasound system, which was fitted into a photographers vest. This was made from January-April 2004. The key components in the system are shown in Fig. 7.

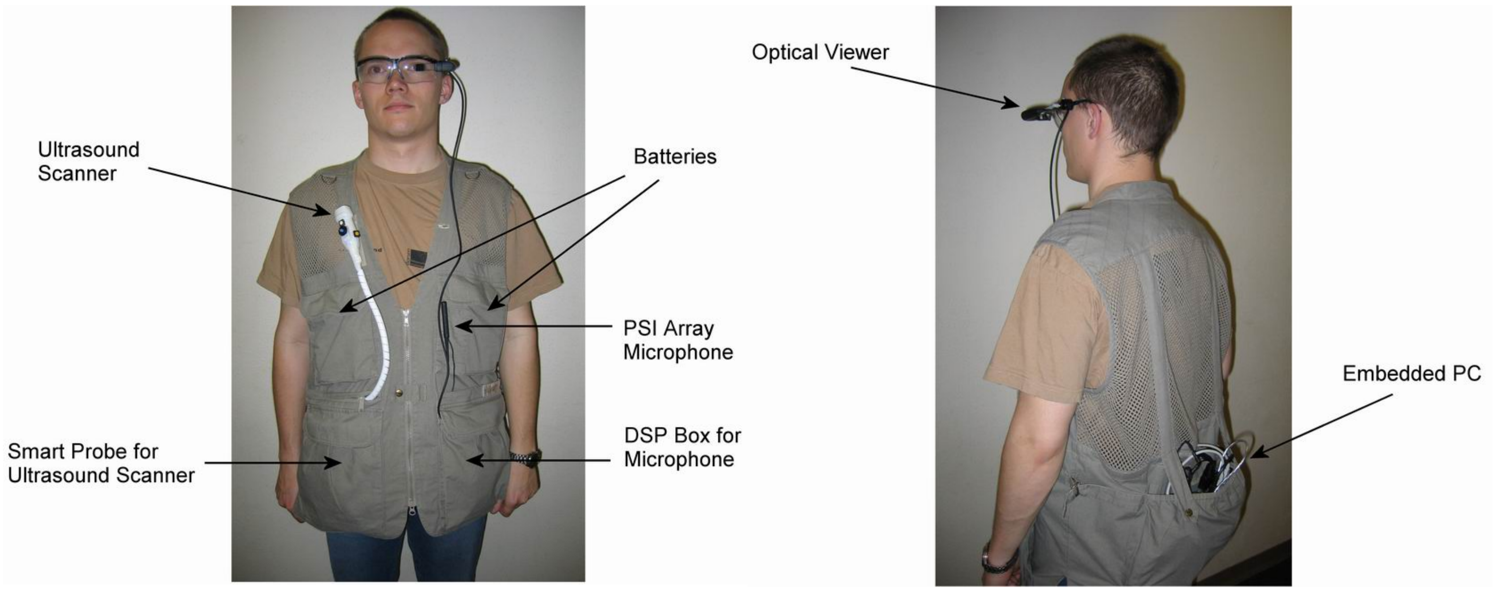


Fig. 7 - 2nd generation wearable ultrasound system

Based on this system we want to implement the 3D capabilities. As explained in the introduction (chapter 1), 3D makes the interpretation of ultrasound images easier. The development of an optical position sensor is described in this thesis. The position sensor is used to track the position of the ultrasound transducer (in the Terason 2000 system) on the surface of the skin. This is basically done in the same way as a regular optical desktop mouse tracks on the surface of a desktop. A block diagram of the hardware is shown in Fig. 8. A PC running on Windows XP is used as the platform. The Terason ultrasound scanner and a position sensor is connected to this PC. Inside the PC there are programs which take care of acquiring and displaying the ultrasound data.

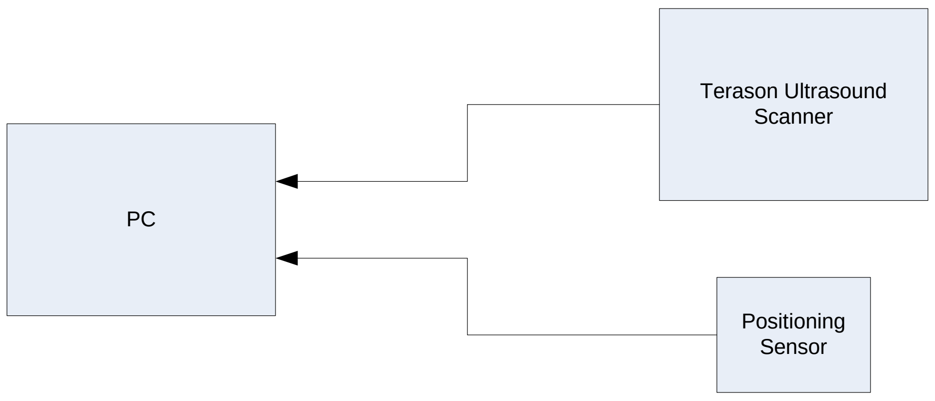


Fig. 8 - Block diagram of the hardware

2.2 Software Setup

The positioning device we are using is able to track in two dimensions (that is, it has two degrees of freedom). But any kind of device that is able to give a position can be used. The software consists of three applications that all run in Windows XP :

- Terason - Ultrasound scanner software
- Sonocubic - 3D rendering software
- Mouse Filter Driver - Mouse driver with a so called filter

They are interconnected as shown in Fig. 9. The Terason application acquires ultrasound data from a Terason ultrasound scanner, processes the data and creates a video image. This image or scan plane is then sent to Sonocubic for three dimensional rendering. Every time Sonocubic receives a scan plane it asks a so called mouse filter driver for the current position of the position sensor and thereby the scanner. With the positioning information it is possible for Sonocubic to position the scan planes correctly so a correct 3D image can be displayed. In the following I will describe each of the three software applications.

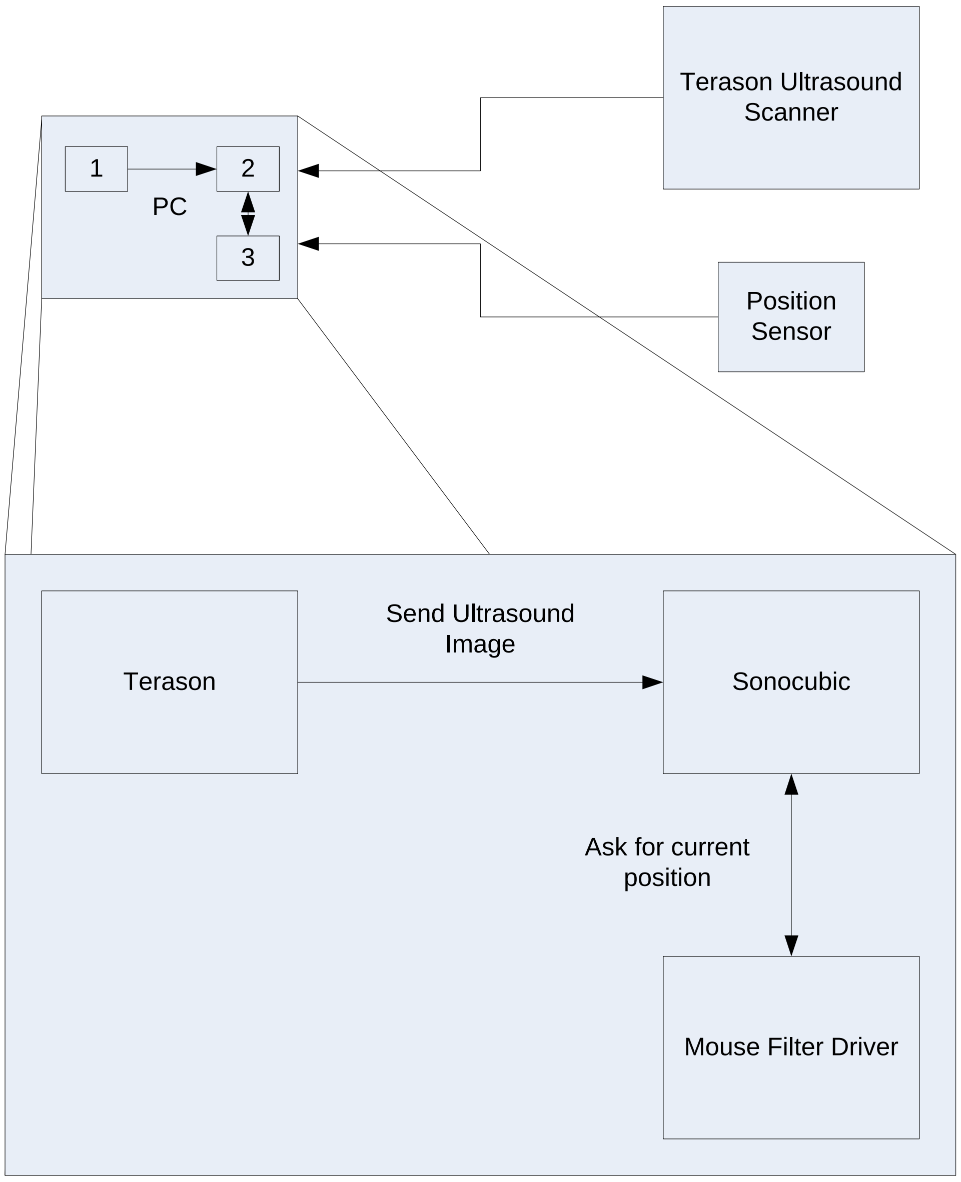


Fig. 9 - Simple block diagram of the 3D imaging system

2.3 Terason

Terason is the manufacturer of the ultrasound scanner. With the scanner comes a software program that is also called Terason. It interfaces to the hardware in the ultrasound scanner. It takes care of transferring the data obtained by the ultrasound scanner and by means of a scan converter it translates the data into a video format. This video information can then be transferred to Sonocubic for 3D rendering.

2.4 Sonocubic

Sonocubic is basically a 3D rendering software program. It is made so it captures images from an analog video input through a frame grabber and stores them in a memory location. After a sequence of scan planes or video images has been recorded, a 3D rendering can be done. But instead of having an analog video interface and a frame grabber, images are transferred digitally from Terason using a shared memory location (a memory where both programs can write and read to/from). This is done using a series of BMP (bitmap) images, that are transferred uncompressed and without loss.

2.5 Mouse Filter Driver and Interface

The mouse filter driver and interface is not an application as such. It consists, as the name indicates, of a mouse driver and a small program code called a dynamic link library (DLL). The driver captures the position information from the mouse and filters it out from the Windows driver stack so that the cursor is not moved on the screen. The dynamic link library then interfaces the driver to Sonocubic. Sonocubic therefore loads this DLL at runtime (in the start up phase). In this way Sonocubic has direct access to get the scanner positions whenever it is necessary.

3 The Optical Position Sensor

3.1 Introduction

After giving a short introduction to the system in the previous chapter, this chapter will describe how the optical sensor works and how we have used it.

A standard optical mouse can be used for tracking movements on a surface. The requirements for the surface is that it has to be uneven or to have an optical pattern. These requirements are indeed fulfilled for the human skin. A simple test where the optical mouse is moved over the skin reveals that it is possible to track on the skin. If the optical sensor circuitry inside the mouse is taken out and mounted on the side of an ultrasound transducer, it can therefore be used to track the movement of the transducer with respect to the skin. This can be utilized to create a 3D image using freehand scanning even if the scan speed is not constant or the scan path is not straight on the surface.

3.2 Design of an Optical Computer Mouse

In Fig. 10 the hardware setup of an optical mouse is shown. The main components are the sensor, the lens/light pipe and the LED. The LED is illuminating the surface with light that comes from an angle. This reveals discontinuities in the roughness of the surface since shadows are created. These shadows can be used to track on. The mouse works by acquiring images of the

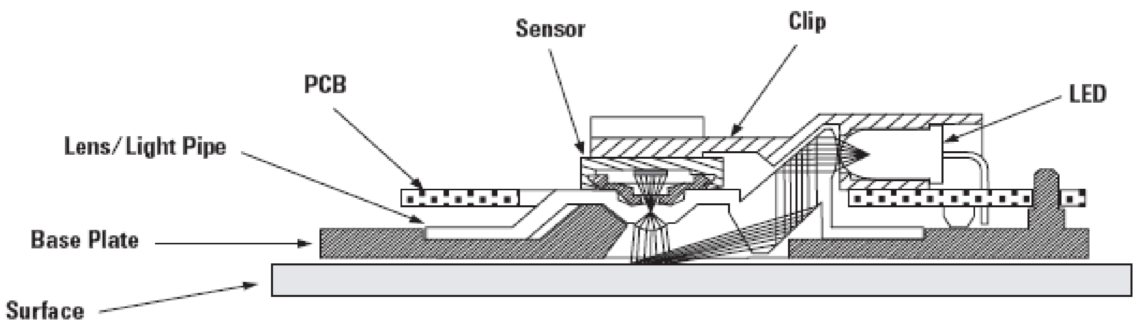


Fig. 10 - Components in an optical mouse [9]

surface on which it moves at a very high rate. This is done through the lens that puts the surface being tracked on in focus. Each captured image is called a frame, and the speed at which the surface is photographed is therefore given in frames per second (fps). In the Agilent ADNS-2610 chip [9], which is the optical sensor used in this project, this takes place at a rate of 1,500 fps. The resolution of the optical sensor is very low, since each image only consists of 18x18 pixels.

After each image has been captured a cross-correlation between consecutive images takes place in order to determine the direction the mouse has moved in. This is done in hardware inside the optical sensor chip and the details of this algorithm is not known.

The concept is illustrated in Fig. 11, where (a) is the previous frame captured and (b) is the current frame. Here the X-coordinate of the mouse (not the image) has changed with -3 while the Y-coordinate has changed with -2 since the last frame was captured.

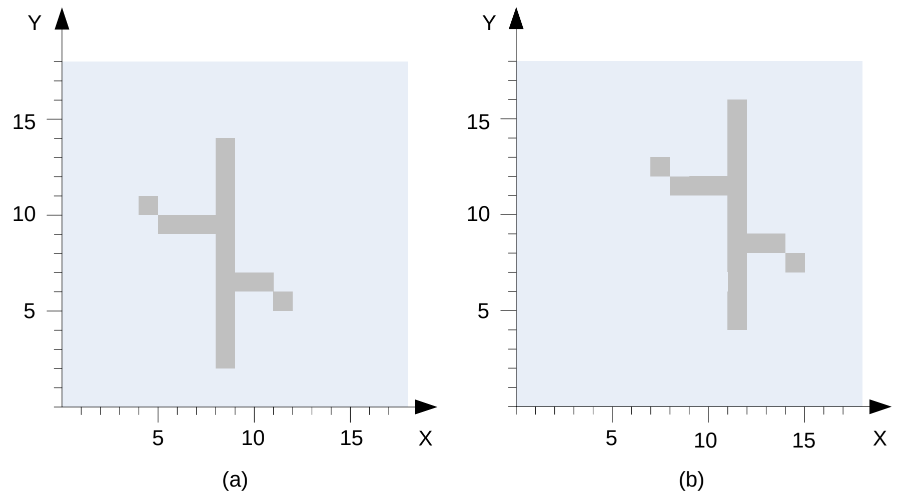
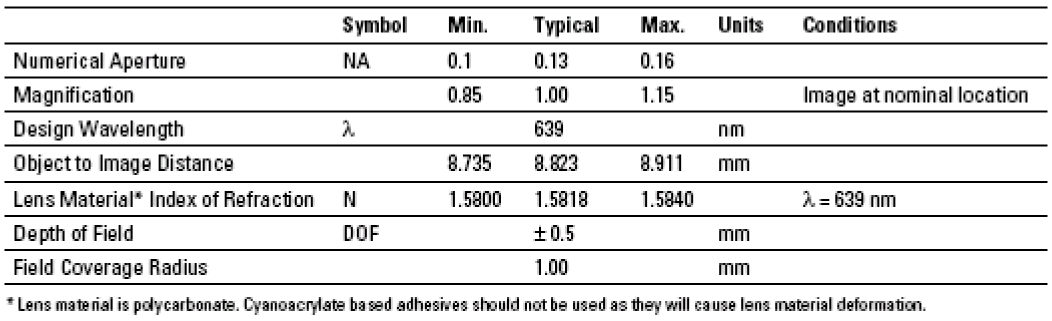


Fig. 11 - Concept of an image captured before and after a movement

3.3 Pixels and Real World Movements

The relationship between the actual physical movement and the corresponding pixel movement can be calculated. This relation does not only depend on the CCD-array, but also depends on the lens being used in front of it. With the HDNS-2100 lens [10] that is designed for the Agilent mouse sensors, the resolution of the CCD-array ends up being 400 cpi (counts per inch). This is a measure of how many counts or pixel changes the sensor can register and is equivalent to the dpi (dots per inch) that is a well-known measure of the resolution of a printer or a scanner. Since the HDNS-2100 lens has a 1.00 magnification, 400 cpi is also the resolution of the CCD-array, but as it can be seen in Fig. 12 this magnification can vary between 0.85 to 1.15, giving a 340 to 460 variation in the cpi. The background for this variation is based on the mechanical assembly requirements that can be found in the HDNS-2100 datasheet [10].



	Symbol	Min.	Typical	Max.	Units	Conditions
Numerical Aperture	NA	0.1	0.13	0.16		
Magnification		0.85	1.00	1.15		Image at nominal location
Design Wavelength	λ		639		nm	
Object to Image Distance		8.735	8.823	8.911	mm	
Lens Material* Index of Refraction	N	1.5800	1.5818	1.5840		$\lambda = 639$ nm
Depth of Field	DOF		± 0.5		mm	
Field Coverage Radius			1.00		mm	

* Lens material is polycarbonate. Cyanoacrylate based adhesives should not be used as they will cause lens material deformation.

Fig. 12 - HDNS-2100 Lens design optical performance specifications [9]

From the 400 cpi we can calculate the height and width of a single pixel to be 1/400 inches. In mm this is 1/400*25.4 = 0.0635 mm. Thus the CCD-array is covering an area that has the size $(0.0635 * 18)^2 = 1.31 \text{ mm}^2$. In order to verify the image size a testpattern was made, and the images from the mouse were captured. This was done using a CCD-array image viewer (appendix A). This image viewer was done as a part of VHDL-class EE574 with Prof. R. J. Duckworth.

The results can be seen in Fig. 13. As it can be seen there is a good correspondence with the size of the area that is actually covered and the theoretical size (1.31 mm^2). This is equal to a square with the size 1.14 mm x 1.14 mm.

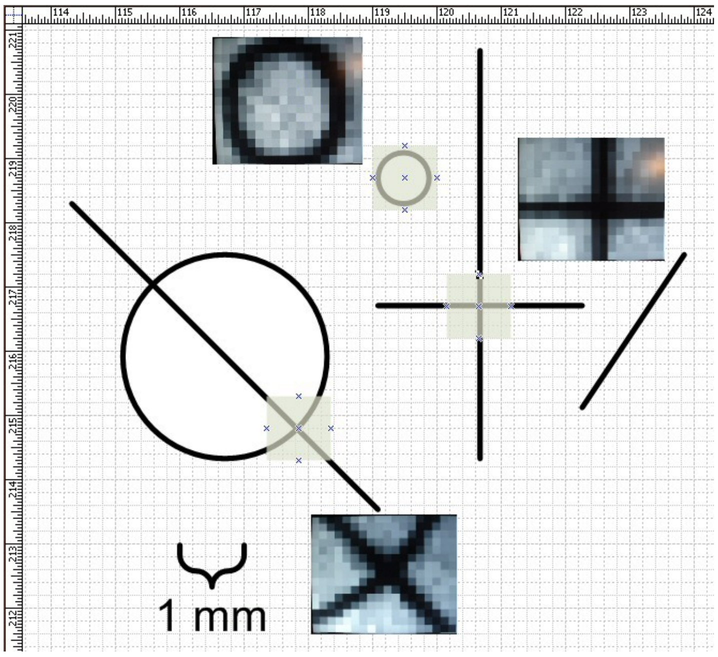


Fig. 13 - Test patterns and captured mouse images

In terms of position accuracy the worst case occurs when a 45 degree movement takes place, that is the same x and y speeds or movement between frames. In this case the change would be 0.0635 * sqrt(2) = 0.0898 mm. It is therefore impossible to achieve an overall higher accuracy than this.

3.4 Mouse Movement in Windows XP

In order to obtain the movement of the mouse it is necessary to find a way to extract the pixel movements detected in the CCD-array. In our case this is done in Windows XP. In a normal Windows environment a movement of the mouse is translated into a movement across pixels on the screen without any direct real world connection.

Three major problems occur if the pixel movements on the screen are used directly :

1 - Hitting the edge of the screen makes the mouse cursor stop

2 - Increasing the speed of the mouse makes the cursor move faster than it should

3 - Only one mouse is allowed to be used in Windows

Ad. 1 - The fact that the mouse is confined to the screen area is called mouse clipping. It is an unwanted effect in our application since we are only interested in getting the position of the mouse and do not care whether the mouse is within a certain area or not.

Ad. 2 - In the Windows environment, the so called "acceleration curves" are stored in the Windows registry. These are curves with an non-linear characteristic between mouse (real world) and pointer (on screen) velocity.

In Fig. 14 an example of this can be seen with four different mouse acceleration curves. The curves are taken from a Microsoft document describing this [11].

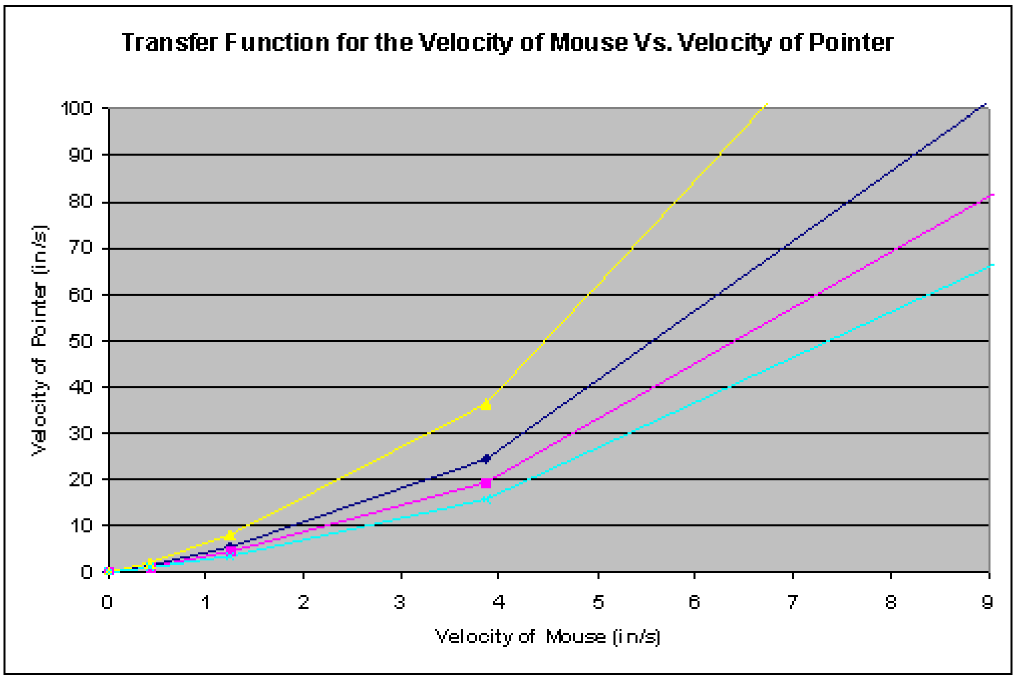


Fig. 14 - Mouse acceleration curves

When the mouse is used as a pointing device this feature is useful, since it makes it easier to navigate on the desktop. We require, however, a completely linear translation from mouse velocity to pointer velocity.

After doing some research I found out that it was fairly easy to overcome both problem 1 and 2 from the previous page by using the Microsoft DirectX technology. DirectX is a technology that gives the programmer control of the actual hardware at a low level. The great advantage is primarily speed, and it was therefore designed to be used by game programmers. An example of the DirectX's use of a Windows driver stack can be seen in Fig. 15. In this example Direct Input, that is a part of the DirectX technology, is used to get inputs directly from the hardware. This is routed directly into the application where we want to use the positioning information.

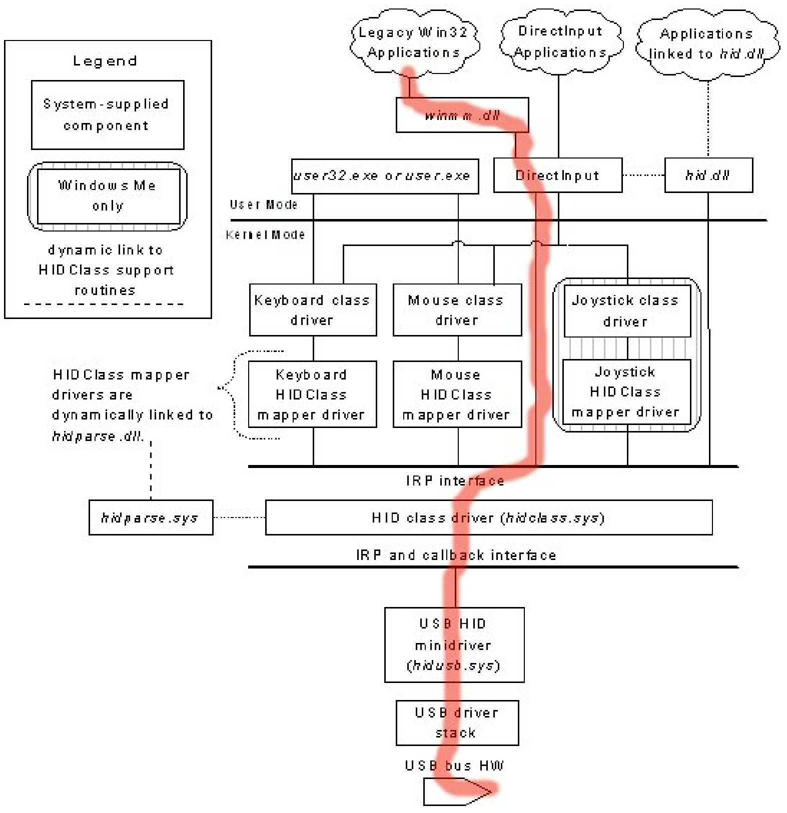

Legacy Win32 Applications
DirectInput Applications
Applications linked to hid.dll
Legend
System-supplied component
Windows Me only
dynamic link to HID Class support routines
winmm.dll
user32.exe or user.exe
DirectInput
hid.dll
User Mode
Kernel Mode
Keyboard class driver
Mouse class driver
Joystick class driver
HID Class mapper drivers are dynamically linked to hidparse.dll.
Keyboard HID Class mapper driver
Mouse HID Class mapper driver
Joystick HID Class mapper driver
IRP interface
hidparse.sys
HID class driver (hidclass.sys)
IRP and callback interface
USB HID minidriver (hidusb.sys)
USB driver stack
USB bus HW

Fig. 15 - DirectX rerouting of positioning information in Windows driver stack

The requirement of having several mouse devices operate simultaneously was, however, not easily implemented and it seems like it is forbidden in the Windows world. Windows appears to be a single mouse environment!

However, I found that the solution to my problems was to use a mouse filter driver. A group of researchers at University of Aarhus in Denmark had already developed a mouse filter driver that could solve my problems [7]. Furthermore they had uploaded all the source code for the mouse filter driver to a website called sourceforge.com. Sourceforge is a place where people share projects that can be used free of charge. This project is called *cpnmouse* and in the following I will refer to it using this name.

Using the *cpnmouse* driver made it possible for me to reroute the input from the mouse, like it also was done in Direct Input as described earlier. In this way the mouse information never reaches the top of the Windows driver stack and the mouse movement does not affect the cursor position - just what is needed in our case. It also eliminates the need for developing a new interface to the computer where for example a microcontroller could have been used to interface to the computer instead.

3.5 Verification of Position Accuracy

In order to use the mouse as a positioning device it is necessary to verify how accurate it is. To do this an experiment was made where an optical computer mouse is mounted on an X-Y recorder. The X-Y recorder is designed to carry a pen and to record on a piece of paper the voltages on the X and Y inputs. The pen was removed and the mouse was mounted. Paper (which has an irregular pattern) was placed in the recorder and is assumed to be a reasonable representation of the human skin.

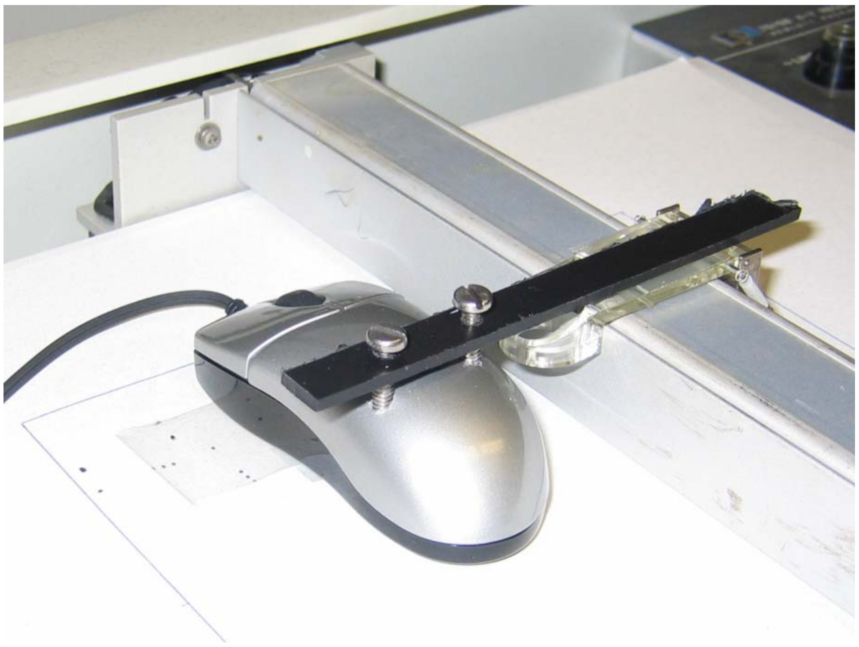


Fig. 16 - XY recorder with mouse mounted

The mouse was moved in a predefined pattern in order to record it's accuracy. Since the arm that holds the mouse is very long, the orientation of the mouse is affected when the recorder changes the direction of movement. Another problem is the yawing (a rotation about the vertical axis) of the mouse, since it is very difficult to align it 100% with the vertical axis. Although the mouse could be mounted in a more rigid way, this setup can still be used to test whether the accuracy of the mouse is good enough or not.

Three patterns were recorded : A horizontal, a vertical and a slanted curve. The movement measured in CCD pixels was stored in the mousetracker program using the DirectInput feature of Microsofts DirectX technology (note that it is not necessary to use multiple mice for the testing and I therefore do not need the cpnmouse driver here). A CCD pixel movement is the number of pixels that the CCD array in the mouse has detected due to the mouse movement since the last image was detected. The CCD array has a resolution of 400 cpi (counts per inch). This means that sweeping the mouse over a distance of one inch will register a movement of 400 pixels.

After the mouse movement has been stored in the program memory it was written to the disk. Thereafter it can be read into Matlab and the curve in Fig. 17 can be plotted.

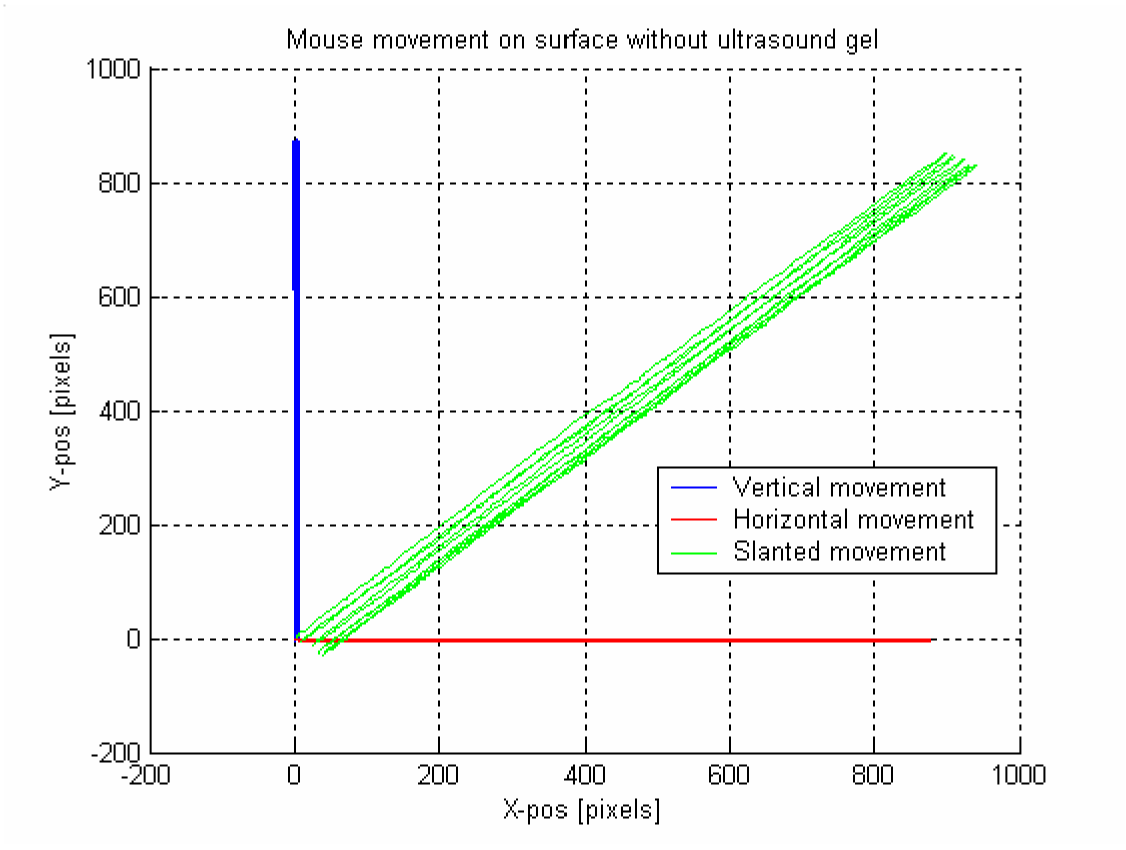


Fig. 17 - Mouse movement using XY recorder

In Fig. 17 the mouse has been moved 55 mm in the X, Y and both directions. In the graph a movement of 55 mm corresponds to 875 pixels (this exact value was found in the datasets I recorded and used to plot the graphs in Fig. 17). This value deviates only 0.56 mm from the expected result :

$$\text{Scale} = \frac{1''}{400\ \text{cpi}} = \frac{25.4\ \frac{\text{mm}}{\text{inch}}}{400\ \text{cpi}} = 0.635\ \frac{\text{mm}}{\text{count}} = 0.635\ \frac{\text{mm}}{\text{pixel}}$$

$$P_{\text{Expected}} = \frac{55\ \text{mm}}{0.635\ \frac{\text{mm}}{\text{pixel}}} = 866\ \text{pixels} \qquad \text{Eq. 1}$$

$$d_{\text{Expected}} - d_{\text{Measured}} = 55 - 875\ \text{pixels} \cdot 0.635\ \frac{\text{mm}}{\text{pixel}} = -0.56\ \text{mm}$$

The errors are likely to be due to the variation in magnification described in the datasheet for the HDNS-2100 lens [10]. This introduces an error of up to +/- 15% or +/- 130 pixels:

$$P_{ExpectedMin} = 0.85 \cdot 866 = 736 \text{ pixels}$$
$$P_{ExpectedMax} = 1.15 \cdot 866 = 996 \text{ pixels}$$

So the error coming from the mounting of the mouse is much smaller than the error found in the specifications for the lens. As it is explained later in this chapter we are using a different lens than the HDNS-2100 lens.

Looking at the blue curve an example can be seen of the inaccuracy of the positioning. After one run the X-displacement is 1 pixel, which corresponds to 0.06 mm. The arm that holds the mouse is affected by the movement. As the mouse is moved up and down this angle changes and influences the result. This results in an accumulated error for each run, which makes the blue curve move a little as the movement is repeated. The maximum deviation occurs on the 5th repetition and is 7 pixels corresponding to a 0.4 mm error.

Looking at the green curve the problem with the mounting of the mouse can be seen. Every time the mouse changes direction its angle is changed a little bit. Since it is now moved in both directions a spiralling hysteresis effect can be seen.

Although the test was not perfect one can say that the deviations are within a reasonable margin of what would be needed to determine the position an ultrasound scanner. And since the test showed that it was mainly a mounting problem that gave rise to the errors we decided to continue our work on using the mouse.

3.6 Extracting Images from the CCD-Array

By using a serial command it is possible to acquire the image that the CCD-array is acquiring. Doing this is quite important in order to achieve the best tracking possible. In appendix A a hardware design for extracting information over the serial link to the position sensor is shown. This makes it possible to qualitatively compare the contrast in the images that are captured by the CCD-array and to adjust the setup to get the maximum contrast possible.

3.7 SQUAL Value

A good measure of whether the CCD-array is in focus or not is the so called SQUAL value that can be read. The SQUAL value is an abbreviation for Surface QUALity and is, according to the datasheet, a measure of the number of features visible by the CCD-array. If the image is out of focus it is more blurred and the SQUAL value is lower, which makes good sense. If there is no surface beneath the sensor the SQUAL value goes toward zero.

The Surface QUALity (SQUAL) value can be found for combinations of different surfaces and different colors of LED light for the optical mouse. According to the datasheet for the ADNS-2610 [9] optical sensor, the SQUAL value is a measure of the number of features visible to the optical sensor. Our interpretation of this is that the more non-uniform the surface is physically and optically, the higher is the SQUAL value. And the higher the SQUAL value is, the easier it is to track on the surface. The goal must therefore be to find a way to get the optimum SQUAL value in order to get the optimum performance.

One thing that is effecting the SQUAL value is whether the sensor is in focus or not. A measure of that can be seen in Fig. 18.

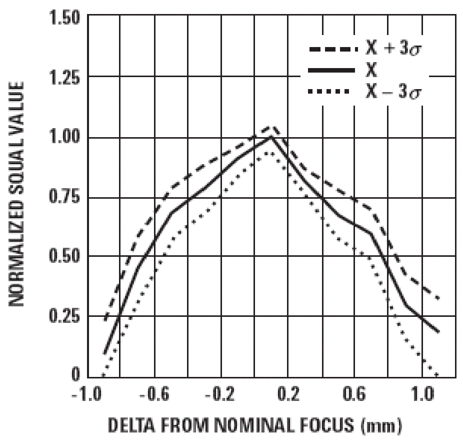


Fig. 18 - Typical mean SQUAL vs. height (on white paper)

As it can be seen, a variation of just 0.4 mm will decrease the SQUAL value by 25%, so it is really critical that the sensor is in focus. Note that this deviation has been measured with the HDNS-2100 lens, which has a shorter focal length (approx. 4.8 mm) than the lens I have used in front of the optical sensor (6.0 mm) that is mounted on the transducer. The smaller focal length gives a smaller depth at focus, and makes it more sensitive to height variations.

In order to see if there was any change in the SQUAL value when using different colors of LED light a measurement of the SQUAL value was done on different surfaces. The surfaces that were selected were my table (that has a color very similar to the skin), white paper, black electric tape (that is smooth and non reflecting), my skin and the CIRS phantom that has a polyurethane surface with a very dark grey color. I choose these surfaces since they seemed to be a good representation of the surfaces on which we want the system to be able to scan. The white paper was included as a reference, since the SQUAL value is given for white paper in the ADNS-2610 datasheet [9]. The results from my measurements can be see in Table 1.

Surface / Color	Red 639nm	Orange 620nm	Yellow 590nm	Green 502nm	Blue 470nm
Table	72	70	79	59	73
White paper	68	68	73	72	6C
Black electric tape	00	00	00	00	00
Skin	46	40	30	8	29
CIRS Phantom surface	41	0C	1A	5	8

Table 1 - SQUAL value on different surfaces when illuminating with different colors

The initial observation from looking at the table is that the red color is the best. But during the test I realized that it is extremely difficult to maintain focus on the skin and on the CIRS phantom surface with the factory mounted mouse since the surface is soft. So because of the high fluctuations in the measurements it can not be concluded that one type of LED is better than the other. One problem is that there are 4 plastic standoffs under the mouse. These standoffs provide an adequate distance to the surface, but will only work on a flat hard surface. It is therefore only the first three rows that provide accurately enough results, and from these rows it can be concluded that no color can be preferred over another.

For the black electrical tape that is very smooth and where there is a very low amount of light reflected it is not possible to observe a SQUAL value at all. A surface with a zero SQUAL value is impossible to track on. And since the surface of our custom made phantom is smooth and black, it is a big problem for testing out the system. A pattern therefore has to be placed on top of the phantom in order to get it to track correctly on the surface (see chapter 5).

My initial thought was that red would be the preferred color since the ADNS-2610, according to the datasheet, was designed to be used with a red LED (639 nm). Also the so called responsivity for the ADNS-2610 can be seen in Fig. 19.

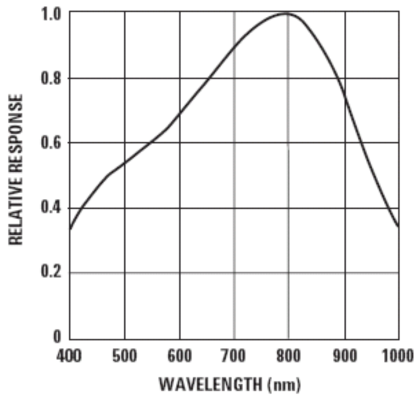


Fig. 19 - Wavelength responsivity

We can see that the optical sensor has the best response (highest sensitivity) at a wavelength of approximately 800 nm, equivalent to the near infrared region of the spectrum. This response should however not affect the SQUAL value, as long as there is enough light reflected from the surface. So the darker the surface gets the more important is the wavelength and the incident light intensity because of the responsivity.

3.8 USAF 1951 Test Pattern

For testing the resolution of an optical system a 1951 USAF test pattern can be used [13]. The pattern was originally developed to be used with a military standard (MIL-STD-150). The test pattern is depicted in Fig. 20.

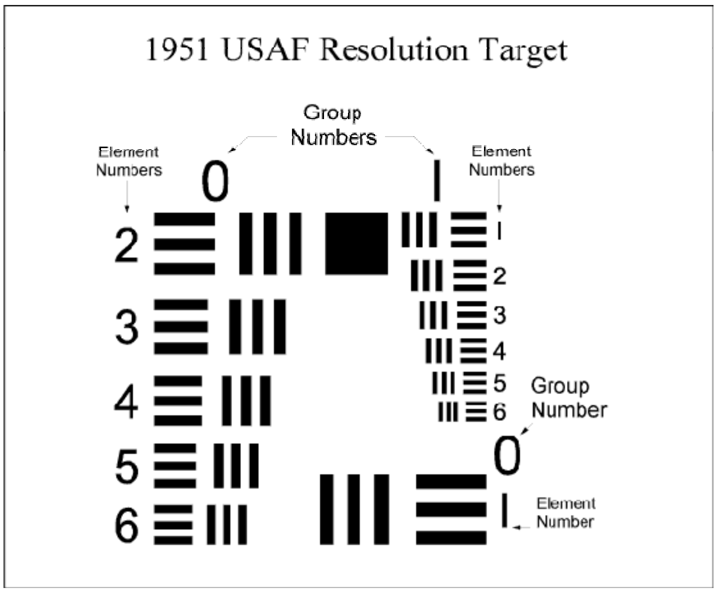


Fig. 20 - 1951 USAF Test Pattern [13]

The chart consists of groups and elements. Each group contains six elements labeled in ascending order from the top to the bottom. However the first group (with the largest elements) has the first element no. 1 placed at the right bottom of the chart. The next smaller group no. 1 can be seen in the upper right corner and in between group no. 0 and 1 there is a black square of the same size as element 2 in group 0.

The chart is used by placing it in front of the optical system that has to be measured. The resolution limit of the system can then be found by examining the element that is indistinct or blurry. And because the chart has both a horizontal and vertical pattern for each element, a horizontal and vertical resolution can be found. Often a smaller pattern is repeated inside the bigger groups. An example of this is shown in Fig. 21.

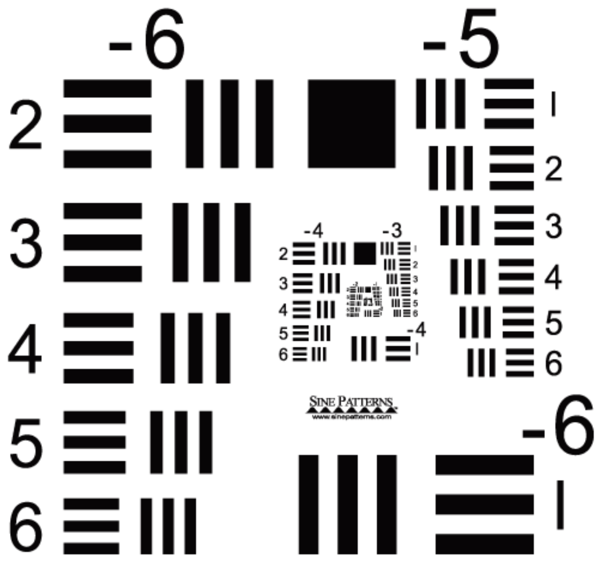


Fig. 21 - USAF 1951 test pattern (40"x40" on photographic paper) with repeated patterns

3.9 Optical fiber

In order to make a more compact and better ergonomic design we investigated if it was possible to couple the image of the skin surface via an optical fiber bundle to the CCD-array. By using an optical fiber bundle it would be possible to bend it so it would form along the shape of the transducer. For this reason we bought a so called fiber optic image conduit from Edmund Optics. The specifications for the image conduit are shown in Table 2.

Table 2 - Specifications for optical image conduit

Image conduit diameter	3.2 mm	Fiber diameter	12 µm
Image conduit length	76.2 mm	Numerical aperture	0.55
Core refractive index	1.58	Cladding refractive index	1.48
Approx. transmission*	35-45%	Conduit resolution**	42 lp/mm

*in the range of 400 to 750 nm **lp/mm = line pairs per mm

An image conduit consists of a lot of small optical fibers that have been glued together to form a bundle of small optical fibers. Using the image conduit an image of a surface can be transmitted and shown in the end of the fiber. The concept is demonstrated in Fig. 22 where I am holding the image conduit on top of a USAF optical test pattern. As it can be seen it is important to get the top of the image conduit in focus in order to be able to track on it.

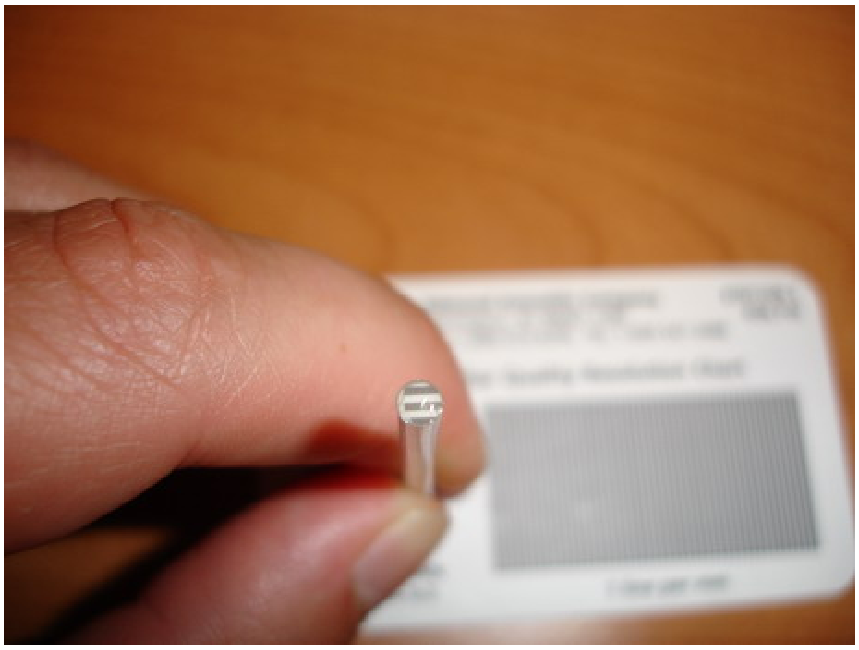


Fig. 22 - Fiber optic image conduit on 1951 USAF test pattern

However, illuminating the surface is a difficult challenge. One way to do this is by illuminating the surface from the same end of the fiber where the image is captured by the CCD-array. This works pretty well on a flat smooth and optically highly varying surface, but if the image conduit gets just a small distance (about 0.5-1.0mm) from the surface it gets out of focus. Another problem is that the surface is not illuminated with light under a small angle relative to the paper surface (since it travels through the fiber), making it more difficult to reveal discontinuities in height on the surface (see Fig. 23). These discontinuities are otherwise revealed by shadowing on the surface, if it is illuminated with light from an angle.

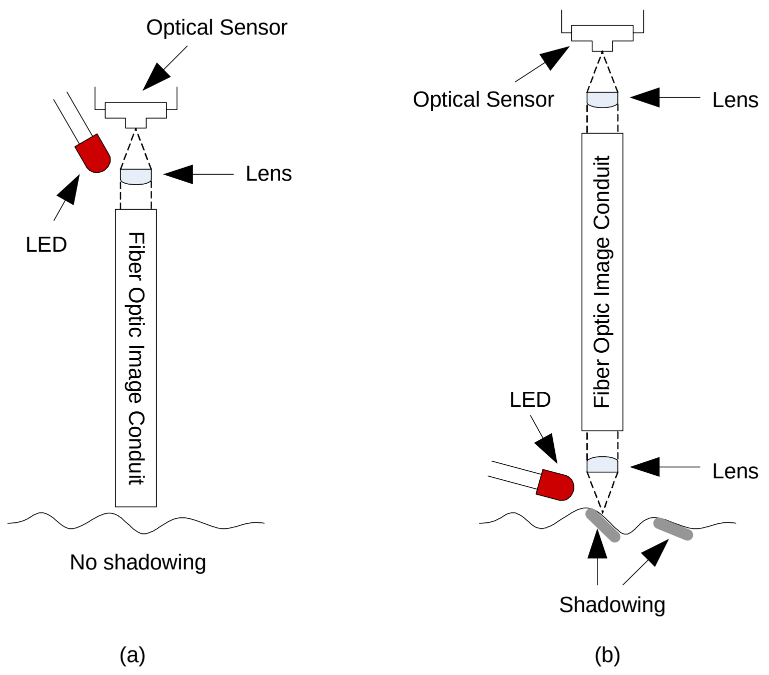


Fig. 23 - Two ways of illuminating surface with optical image conduit

For this reason it was decided to try to mount a lens in front of the fiber. The lens gives a little more freedom in varying the distance from the surface, since the lens has a certain focus depth. The ability to do this is directly proportional to the focal length, meaning that the longer the focal length the better the lens will be to keep the image in focus.

In this way it is also possible to illuminate the surface with an LED mounted at the surface. This LED can illuminate the surface from an angle and gives shadows on the surface at discontinuities, as shown in Fig. 23 (b).

It was decided to try to mount the lenses and the fiber together. However, before this could be done it was necessary to calculate what type of lens and at what distance it should be mounted in order to give a 1:1 magnification and transmission of the surface. Before calculating this, the basic theory behind the lens calculations will be presented and then this theory will be used to calculate the dimensions of the lens holder.

A simple lens can be described as a the intersection of two circles with each other as depicted in Fig. 24. This will create a biconvex lens with the thickness D, radii R_1 and $-R_2$ and the refractive index n.

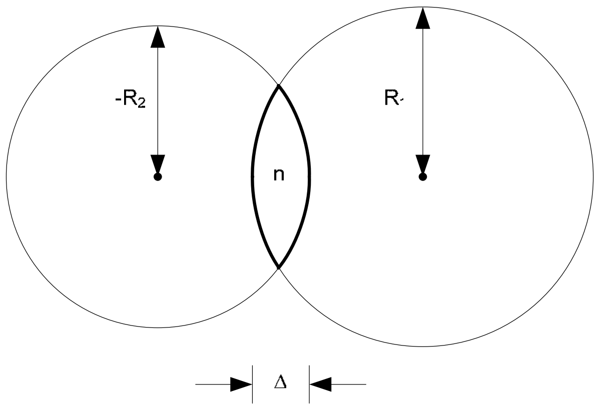


Fig. 24 - A biconvex spherical lens

Using a concept called raytracing it can be described how an image that is transmitted through a lens is formed on the other side of it. In Fig. 25 a ray leaving P_1 goes into the lens at an angle θ_1. In the air-glass interface it is refracted at an angle θ (not shown) and then hits the back side of the lens at the glass-air interface. If the lens is thin it can, however, be assumed that the incident ray emerges at about the same height at which it enters and with the angle θ_2.

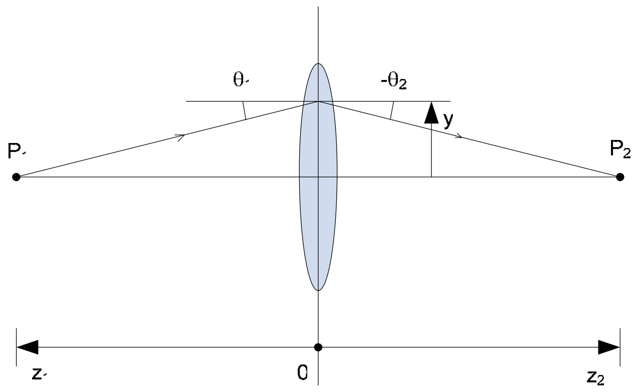


Fig. 25 - Raytracing through a lens

Using this assumption the incident and refracted rays are related as

$$\theta_2 = \theta_1 - \frac{y}{f} \qquad \text{Eq. 2}$$

where f, the focal length, is given as

$$\frac{1}{f} = (n-1)\left(\frac{1}{R_1} - \frac{1}{R_2}\right) \qquad \text{Eq. 3}$$

Note that R_2 is negative, so the two terms add up giving a positive focal length. As shown in Fig. 25 the rays are originating in $P_1(y_1,z_1)$ and ends up at $P_2(y_2,z_2)$. From here we have two equations, one for the imaging,

$$\frac{1}{z_1} + \frac{1}{z_2} = \frac{1}{f} \qquad \text{Eq. 4}$$

and one for the magnification,

$$y_2 = -\frac{z_2}{z_1} y_1 \qquad \text{Eq. 5}$$

Note that the two z-values are from a different coordinate system, being positive in the left and right direction from the origin as shown in Fig. 26. The image will therefore be mirrored, which is shown in Eq. 5 with the minus sign and magnified with factor z_2/z_1.

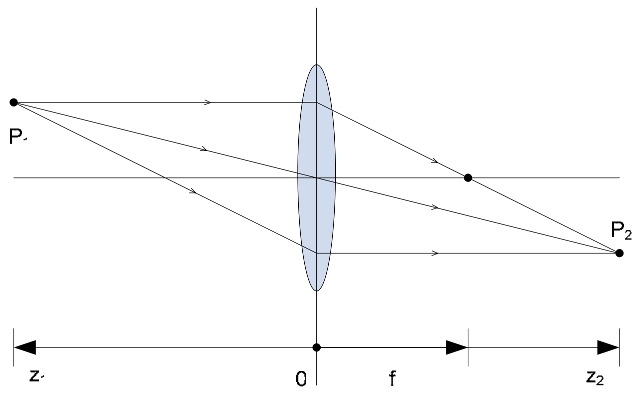


Fig. 26 - Image formation by a thin lens

However, a big problem is how to mount these lenses and the image conduit together. The diameter of the image conduit is only 3.2 mm and the diameter of the lenses are just 3.0 mm. So we are dealing with very small dimensions.

The type of lens that was used is a plano-convex lens. Three of these lenses were purchased for experimental purposes before much was known about the theory behind lenses.

It was decided to try to mount the lenses in plexiglas, since this material is very easy to work in. The necessary holes were milled and the lenses mounted, a process that best can be described as trying to solder SMD (Surface Mounted Devices) to a PCB (Printed Circuit Board) with a very big soldering iron.

A plano-convex lens is flat on one surface and curved on the other. Or, in other words, R_2 in Fig. 27 is going to infinity and R_1 is as given in Table 3. Note that if this is the case the focal length will be the same on both sides if the thin lens approximations are used.

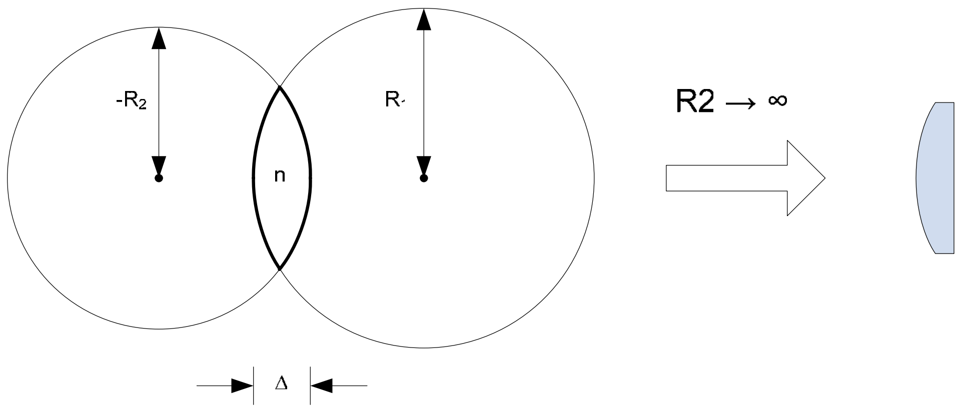


Fig. 27 - Lens radii

Table 3 - Lenses (all dimensions in mm) (FL = Focal Length)

Lens no.	Dia	Eff. FL	Back FL	Center Thickness	Edge Thickness	Radius R_1
1	3.0	4.5	3.50	1.80	1.47	3.62
2	3.0	6.0	4.81	1.80	1.41	3.10
3	3.0	9.0	8.01	1.50	1.25	4.65

Since lens no. 3 has the longest focal length (FL) it would be the preferred one to use. But because of the dimensions of the ultrasound transducer and the fact that I cannot change the length of the optical fiber I decided to use lens no. 2 with an effective focal length of 6.0 mm.

Using lens 2 I can calculate the dimensions of the plexiglass housing that holds the lens in place in front of the optical fiber. For the milling in the plexiglas I have to calculate 3 lengths, shown as 'A', 'B' and 'C' in Fig. 28. The diameters used were 2.9, 3.0 and 3.2 mm for 'A', 'B' and 'C' respectfully.

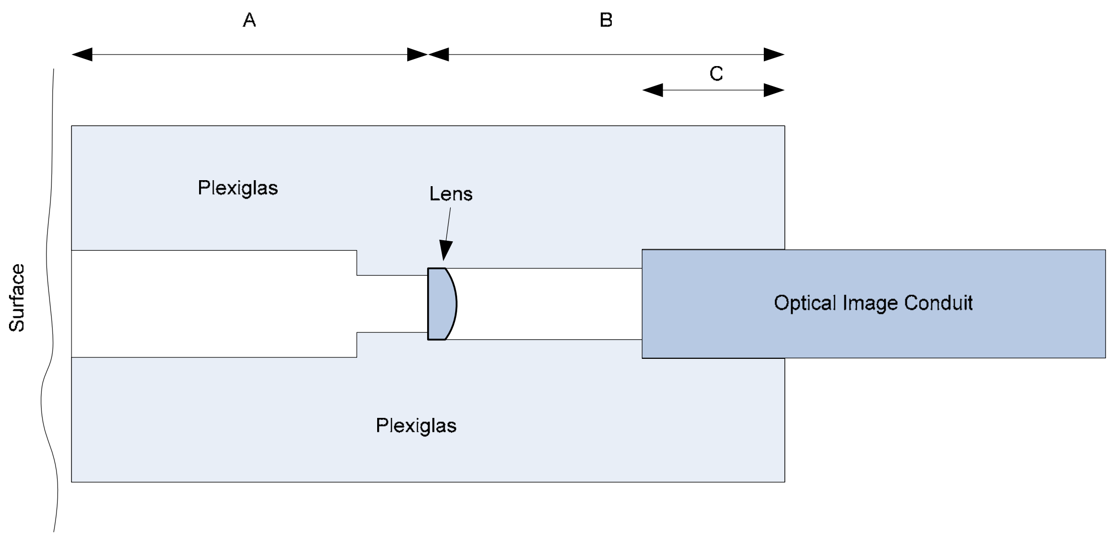
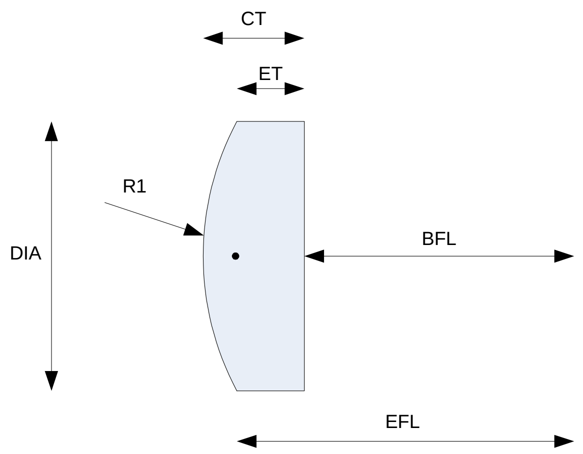


Fig. 28 - Dimensions for lens holder

From the specifications for the chosen lens no. 2 I can find the effective focal length (EFL), the back focal length (BFL), the edge thickness (ET) and the center thickness (CT).



CT	Center Thickness
ET	Edge Thickness
BFL	Back Focal Length
EFL	Effective Focal Length
R1	Radius
DIA	Diameter

Fig. 29 - Dimension of lens

If we want a 1:1 magnification (that is, no magnification) we should place the surface at 2x the EFL. For lens 2 'A' will therefore be :

$$\begin{aligned} A &= BFL + EFL \\ &= 4.81 + 6.0 = 10.81\ \text{mm} \end{aligned} \qquad \text{Eq. 6}$$

'B-C', the distance between the lens and the optical image conduit will be

$$\begin{aligned} B - C &= 2 \cdot EFL + (EFL - BFL) \\ &= 2 \cdot 6.0 + (6.0 - 4.81) = 13.19\ \text{mm} \end{aligned} \qquad \text{Eq. 7}$$

If the overall distance 'A+B' is picked to be 30.0 mm, 'B' and 'C' can also be found. I need these distances to calculate the depths of the holes in the plexiglass which I have to drill.

$$A + B = 30.0\ \text{mm}$$

$$\begin{aligned} B &= 30.0 - A \\ &= 30.0 - 10.81 = 19.2\ \text{mm} \end{aligned} \qquad \text{Eq. 8}$$

$$\begin{aligned} C &= B - 13.19\ \text{mm} \\ &= 19.2 - 13.19 = 6.0\ \text{mm} \end{aligned}$$

As it can be seen the drilling has to be very accurate since the components used are very small. For that reason the only way possible to do this accurately enough is by using a milling machine and do the drilling here, which I ended up doing. It is a very difficult process and it takes a long time to make the lens holder, because of the small tolerances and dimensions.

After making the lens holder and mounting the lens, a simple test verified that the image I viewed was in focus on the USAF test chart. Also there was no magnification. Visibly the image was identical to if the image conduit was placed directly on top of the chart. The contrast in the image was, however, a bit lower. I believe that part of the reason for the lower contrast is because light goes directly from the LED through the plexiglass and interferes with the image reflected from the surface as shown in Fig. 30.

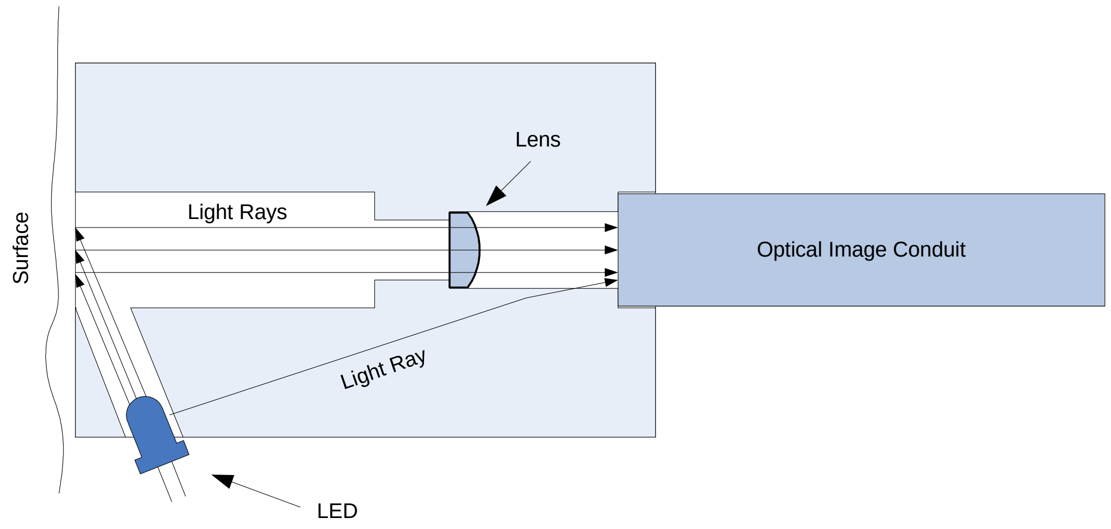


Fig. 30 - Light going from LED both along the intended paths and directly through plexiglas and into optical image conduit

I came to this conclusion by holding the lens holder with the LED and optical image conduit out in the air (where no light is reflected). Here it was possible to observe a dark red color in the image conduit. This color was quite close to the color of the black parts on the USAF chart, indicating that the dark red color of the black parts comes from interfering light rather than from the surface. To avoid this I tried to paint the inside of the holes in the plexiglass black. I was not successful in doing that, since the paint changed the diameter of the hole, making it impossible to mount the lens again.

I therefore decided to make another version of the lens holder in aluminum instead of plexiglas. This version gave a bit more contrast, but the black areas were still a little dark red instead of black. I think that the problem comes from reflections and scattering as the light goes through the lens. Some of these reflections come from the aluminum wall and painting the walls black should, at least theoretically, lower the interference from these reflections. But one would then have the same problem as with the plexiglass that the diameter is changed. The scattering in the lens is however impossible to overcome, since the light going through the lens will always be slightly scattered. The scattering illuminates some of the optical fibers that otherwise should have been dark, creating an image with a lower contrast.

A third problem I found was that light appears to be coupled between the fibers in the image conduit. This problem increases with the angle at which the light enters the image conduit. Light coming from the scattering will therefore not only interfere as described above, but also create interference inside the image conduit, since this light enters with an angle.

Another thing that could cause problems using the image conduit is that it has a “chicken-wire” pattern. The chicken wire occurs because small optical fibers make up the image conduit and at the boundary of each optical fiber there will be a shadow. This problem is described in [4]. Here it is suggested to apply a low pass filtering to the image to overcome this. But this would not be a very good idea since all the details in the image hereby disappear; and it is mainly the details that the mouse is using to track it’s position on the surface. Furthermore this is not possible for us to do, since all the imaging is done inside the optical sensor chip.

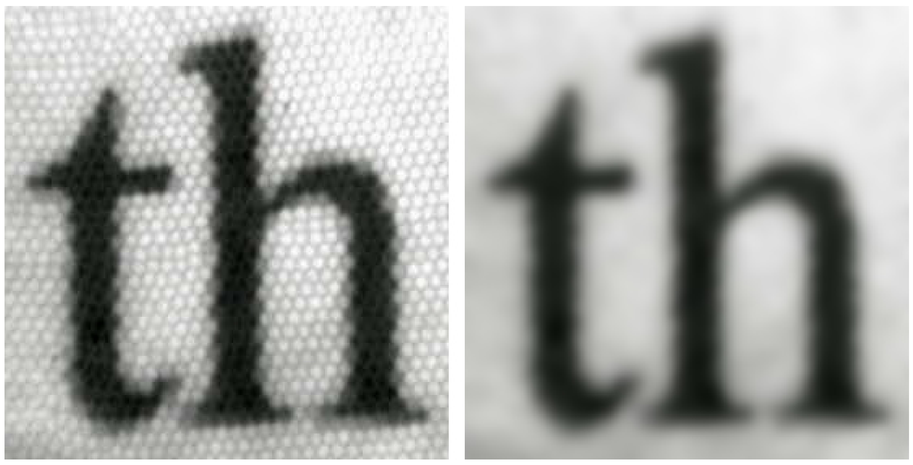


Fig. 31 - Chicken wire pattern in image captured (before and after low pass filtering)

3.10 Testing of the optical fiber with lenses

After assembling the optical fiber with the associated lenses, I did some testing of the system to see how good it was at tracking. It was my first impression that it was not as good as the optical mouse where I took the optical sensor from. It was, however, possible to track on the USAF test pattern (see later) and it was also possible to track on the table in my office that has a highly unregular pattern, as seen in Fig. 32.

Fig. 32 - Table pattern

Tracking on the skin, however, presented greater difficulties. It felt like the sensor could not pick up the movements. I therefore made a comparison on the USAF chart between the different mounting methods. In Fig. 33, Fig. 34 and Fig. 35 these images are shown.

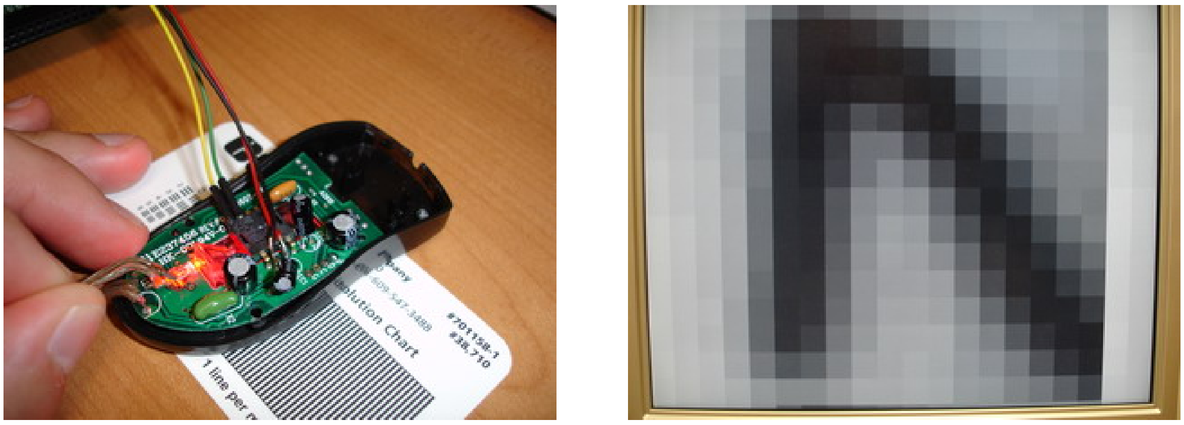
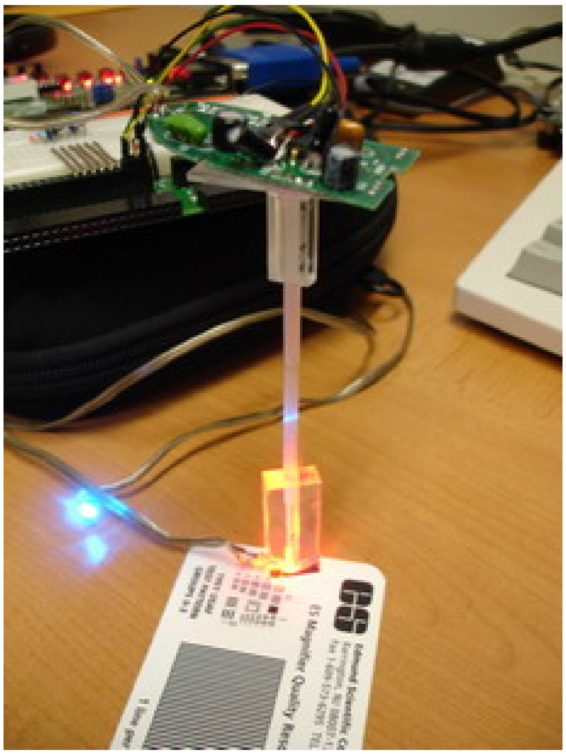


Fig. 33 - Standard (factory) mounting and red LED



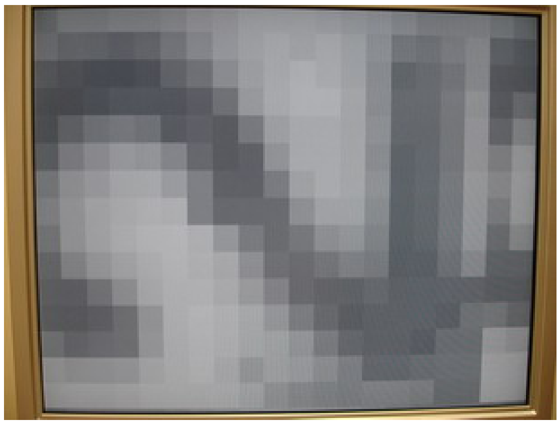


Fig. 34 - Mounting with optical fiber, two lenses and red LED

As it can be seen the image taken with the optical image fiber conduit is more blurry than the others. It can also be seen that the image is not mirrored, as expected, since two lenses are used together with the optical fiber, making the mirroring to happen twice.

In Fig. 35 it can be seen that the image is clear and sharp in the customized aluminum housing. The image looks a bit different from the factory mounting, but this is because there is no lens that spreads the light out in front of the LED. This makes the light a bit more uneven on the surface.

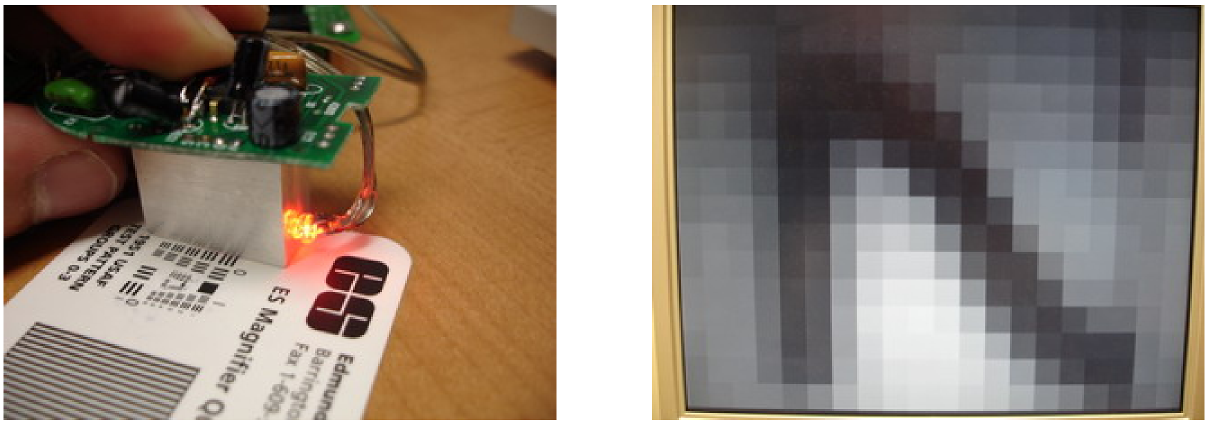


Fig. 35 - Customized housing in aluminum and red LED

3.11 Optical Position Sensor based on Laser Mouse or Optical Mouse

At the time I began my research for this thesis Logitech and Agilent developed a new type of mouse. This new mouse is using the same principle as the optical mouse, but instead of an LED a laser diode is used to illuminate the surface.

Fig. 36 - Logitech Laser Mouse MX-1000

The advantage of this is that the contours of the surface appears much better as opposed to a regular optical mouse. An example of a smooth surface captured with a regular optical mouse and a laser mouse can be seen in Fig. 37.

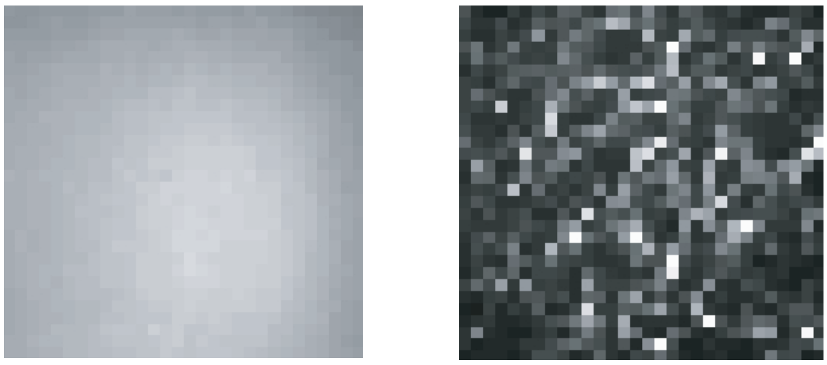

(a) (b)

Fig. 37 - Smooth surface captured by the CCD array in an optical (a) and a laser mouse (b) [8]

As it can be seen the contrast in the image captured from the laser mouse is much better. This makes it possible for the laser mouse to track more accurately on surfaces and it is also possible to track on black surfaces, something that is quite relevant for testing purposes since most ultrasound phantoms are black. Tracking on glass and transparent surfaces remains a problem, but that is not a limitation for the intended application. Unfortunately, I observed that the laser mouse, as opposed to the optical mouse, does not work through a transparent surface mounted in front of the sensor.

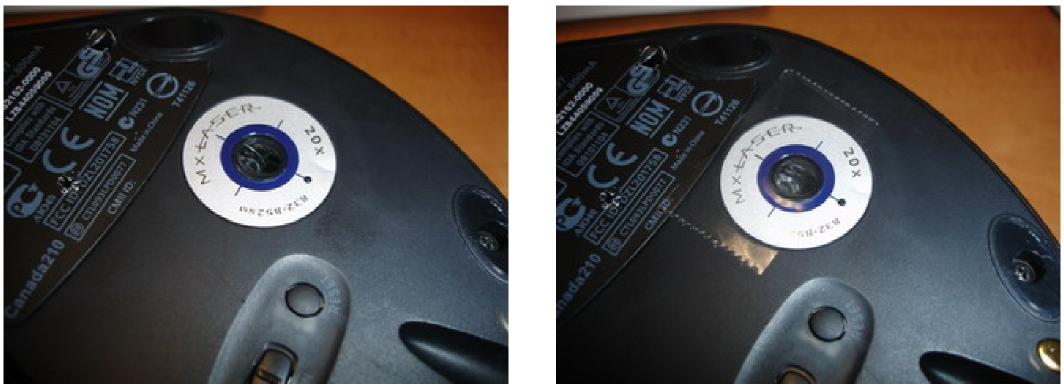


Fig. 38 - Laser mouse without and with transparent tape mounted in front of sensor

It is necessary to place a transparent surface in front of the sensor since the ultrasound gel otherwise will clutter up in the sensor opening. It also looks as if the intensity of the laser is getting lower when the tape is placed in front of the sensor. I have not looked into the details of this.

3.12 Conclusion on Optical Mounting Method

I think that the optical sensor, as it is mounted on the ultrasound transducer, is suitable for testing in a real life application. It is a good proof of the concept that an optical mouse can be used to provide limited tracking (2 degrees of freedom) in a portable system. With the optical fiber it should be possible to create an ergonomically better design. But it will never be fully as good as when only the optical sensor and a single lens is used. My tests convinced me that an optical fiber could be used on a surface with a highly uneven pattern, like my table or the USAF test pattern. But the skin, although it is quite uneven, is not varying enough and the blurring that takes place through the fiber is enough to degrade the image so much that it is not possible to track on it.

I suggest that this work can be continued by integrating the optical sensor in a better way. This could be done in a way where the optical sensor is lying down and a mirror is used as shown in Fig. 39. This implementation is more flat and would fit better on the ultrasound transducer. I have seen this method being used in a so called pen-mouse.

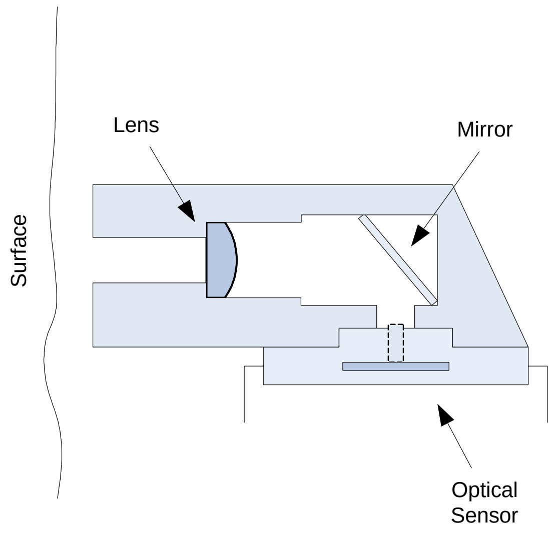


Fig. 39 - Housing for optical sensor with mirror

4 Software and interfacing

4.1 Introduction

The previous chapter described how the hardware for the optical sensor was made. This chapter describes the relevant aspects of the Terason and Sonocubic software and how it is interfaced to the positioning system. First it is described how images are transfered from Terason to Sonocubic. Then it is described how the position system was interfaced to the existing software.

4.2 Transfer of Images between Terason and Sonocubic

Images are transferred between Terason and Sonocubic through a shared memory space. In the programming world's terminology such a space is called a memory segment. This memory segment contains a series of ultrasound images, so it can also be described as a 3D matrix that acts as a buffer between Terason and Sonocubic. This shared memory segment is created by using a small program piece called a DLL (Dynamic Link Library) to be described in detail later. The ultrasound images are then written to the shared memory by Terason and read by Sonocubic. The concept is depicted in Fig. 40.

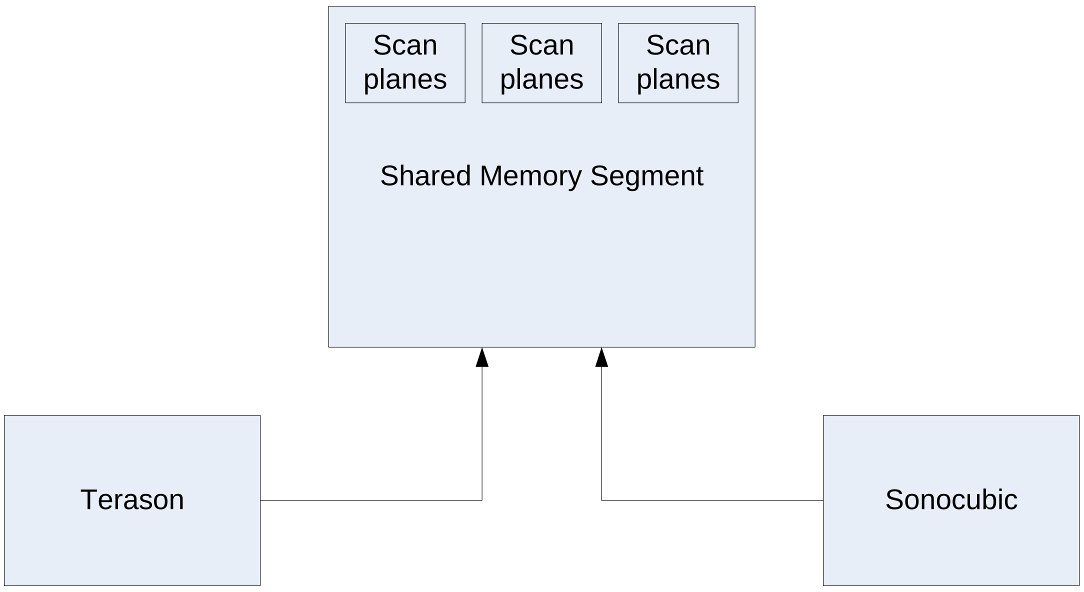
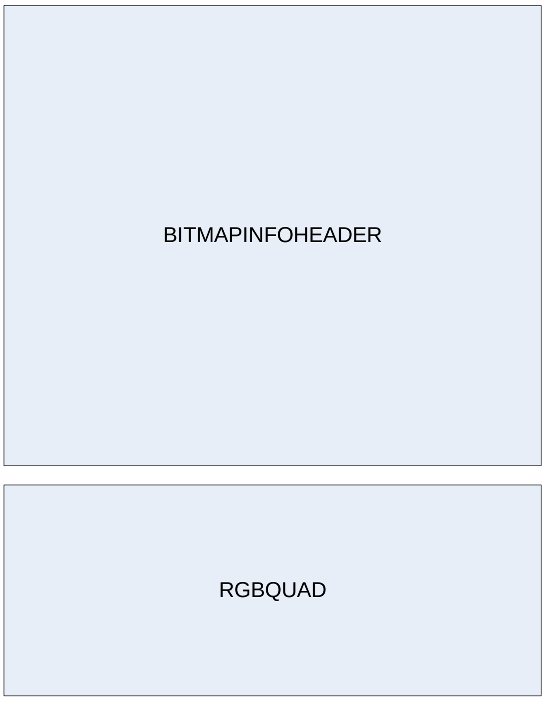


Fig. 40 - Concept of transfering scan planes between Terason and Sonocubic

The images are transferred digitally as opposed to the way Sonocubic works with other ultrasound scanners (through an analog video input and a frame grabber). The transferring of images is done continuously, meaning that only a subset of the number of 2D slices that will make the entire 3D image are in the shared memory at a given time (I have been told by Ricardo Gayoso, a software developer at Sonocubic that typically only one or two images are in the shared memory).

Each scan plane is stored in a standard format called a DIB (Device Independent Bitmap). The DIB consists of two distinct parts : a BITMAPINFO structure followed by an array of bytes defining the pixels in the bitmap. The BITMAPINFO structure can be found in the documentation for the Windows GDI (Graphics Device Interface) and is shown in Fig. 41.

```
typedef struct tagBITMAPINFO {

   BITMAPINFOHEADER bmiHeader;     //see structure below
   RGBQUAD          bmiColors[1];  //see structure below

} BITMAPINFO, *PBITMAPINFO;

//Followed by an array of bytes describing the pixel values
```



```
typedef struct tagBITMAPINFOHEADER{

  DWORD  biSize;
  LONG   biWidth;
  LONG   biHeight;
  WORD   biPlanes;
  WORD   biBitCount;
  DWORD  biCompression;
  DWORD  biSizeImage;
  LONG   biXPelsPerMeter;
  LONG   biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;

} BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

```
typedef struct tagRGBQUAD {
  BYTE    rgbBlue;
  BYTE    rgbGreen;
  BYTE    rgbRed;
  BYTE    rgbReserved;
} RGBQUAD;
```

Fig. 41 - BITMAPINFO structure

As it can be seen in Fig. 41 the structure contains a bitmap header that describes information about the image such as width, height, vertical and horizontal resolution and the number of colors used. Here it is worth to notice the resolution that is given in pixels per meter. This number, which is defined and placed in the scan plane by Terason, is important for measuring the distance and therefore also the area and volume calculations that has to be carried out. Depending on the number of colors used, a color table that defines the color associated with each pixel value, can follow the header.

4.3 The Sonocubic Coordinate System

When Sonocubic has read the images from the shared memory they are stored in a three dimensional matrix. This is done scan plane by scan plane; note that Sonocubic has no knowledge about the distance between scan planes or the offset from the center (this is where our positioning system comes into the picture).

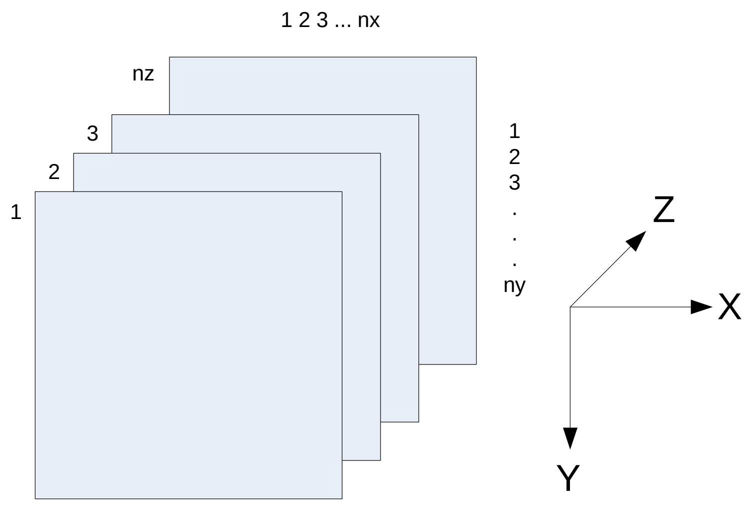


Fig. 42 - Scan planes in Sonocubic

As can be seen in Fig. 42 the scan planes are stored in an [nx,ny,nz] matrix. This is done so the first plane in time is stored at position 1. Note the orientation of the coordinate system where normally the Z-axis points into the body and the X- and Y-axis describe the coordinates on the surface.

The ultrasound data can be stored in Sonocubic in two different modes : a high resolution and a low resolution mode. In the high resolution mode the matrix the ultrasound data is stored in has the dimensions nx = ny = 512, and nz denotes the number of scan planes obtained from Terason through the shared memory interface. I have understood from Ricardo Gayoso at Sonocubic that the maximum value for nz is also 512. In the low resolution mode nx = ny = 256.

Since the images are transfered digitally through the shared memory there should be less degradation relative to an analog interface with a frame grabber. Unfortunately, an interpolation might take place when a scan plane is transfered into Sonocubic, depending on the size of the received scan plane. This is because Sonocubic works on a pre-defined 3D grid or matrix in which the number of pixels in each scan plane are for example 512 x 512 or 256 x 256 depending on the resolution mode selected. So if a scan plane received from Terason has a size of for example 388 x 388 pixels, then it has to be interpolated onto a 512 x 512 scan plane that makes up the 3D matrix that is visualized in Sonocubic. Nonetheless, the 2D window in Sonocubic that loops through all the received scan planes, is still displayed in the original image size (388 x 388). This creates a small problem when doing volume estimations, since an AVI-file (a video file) of the scan planes is saved in Sonocubic in the 388 x 388 resolution. So, in short, an interpolation is done twice when scan planes are entering and exiting from Sonocubic. The details of this will be described in chapter 5 where volume estimations are done.

All the measurements in this thesis were carried out using the high resolution mode.

4.4 Obtaining the position and interfacing with Sonocubic

After describing how the transfer of scan planes is done between Terason and Sonocubic I will now describe how the interfacing of the positioning software to the existing software was performed. This development was done in collaboration with Ricardo Gayoso. It was decided to use the following scenario for including the positioning information :

1 - A frame is placed in the shared memory by the Terason application

2 - Sonocubic reads it from the shared memory

3 - Immediately thereafter the position is captured from the position device

The procedure using the numbers above is depicted in Fig. 43.

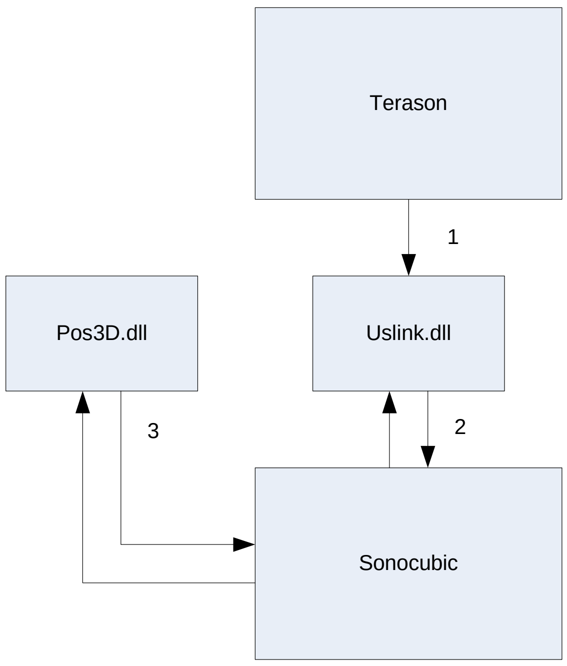


Fig. 43 - Algorithm for transferring images and obtaining position

Sonocubic consists of a lot of DLLs (there are 60 in the Sonocubic directory). As shown in Fig. 43 one of them is 'uslink.dll' that contains the code and functions that makes it possible to transfer images from Terason to Sonocubic.

'Pos3D.dll' is another DLL that was developed by us. The idea with 'Pos3D.dll' is that it interfaces to our hardware and gets the position when it queries the optical position sensor. One of the first questions that arose is how big the delay is from a scan plane is acquired in Terason until a mouse position is obtained from Sonocubic. How much can the mouse move during this time?

If we assume that the scanner is moved at a speed of 2 cm/sec and that frames are received from Terason every 30 ms we get a distance between each frame of 2 cm/sec * 30 ms = 0.6 mm. For testing purposes I recorded how often Sonocubic queried 'Pos3D.dll' for the position. I could see that this actually happened every 30 ms. I therefore think that it is reasonable to assume that the delay is smaller than 30 ms and that the shared memory buffer hardly ever contains more than one image. It would have been interesting to see how big the actual delay is, but since I do not have access to the software in Terason or Sonocubic I cannot get closer to a qualified guess than this.

4.5 Data Extraction from Optical Sensor

Data can be extracted from the optical sensor using a serial interface. Usually this is done with a microcontroller, but it can also be done with an FPGA. In a standard mouse, the microcontroller is extracting the change in position from the optical sensor. It then sends that information through the USB-interface and into the kernel of the operating system. Since the only information we are interested in is the position, it is therefore not necessary to build any new hardware. It is also not necessary to develop or port any microcontroller code for interfacing. All the hardware in the mouse can be reused, which makes the development a lot simpler.

4.6 CPN Mouse Driver

As mentioned in chapter 3 about there are a number of ways to extract the position from a mouse, but in this project I have decided to use the CPN mouse driver, since it provided us with all the features necessary. CPN is an abbreviation for Colored Petri Nets and is related to the application the CPN mouse was originally designed for.

The CPN mouse driver consists of a lower and higher level API (Application Programming Interface) as shown in Fig. 44. An API is the interface that is provided to programs (applications) in order to be able to communicate with the driver. The API can present the data needed in a nicer and more usable way for the application.

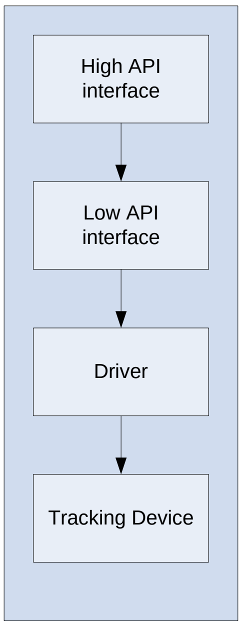


Fig. 44 - CPN-mouse driver stack

The mouse driver can be used in two different modes : by sending events from the driver to a Windows program or by polling it from a program. Based on the setup of our system we are interested in polling it, but I will describe both modes to give a full understanding of how the driver works and how the software is used. It will also be useful for future reference to know the different possibilities that the driver provides.

Using the high level API three functions are used :

```
int __cdecl hInitialise(int count, HWND window, HDC getscreen, unsigned int getevents);
void __cdecl hGetAbsolutePosition(int number, hPOINT* p);
void __cdecl hCleanup(void);
```

As the names indicate, ‘hInitialise’ initializes the driver. In addition to this a handle to a window can be passed by using the window argument. A handle is a void pointer that points to the window where data/events has to be sent. The positioning DLL is running in DOS-mode, so we are not using this feature, but if a Windows program were used it would be possible to pass mouse events directly to the program. The ‘getscreen’ parameter is used for passing the features of the screen being used. This can for example be used together with the clip feature that will be described later. In the ‘getevents’ argument the events of interest are specified. The arguments that can be supplied here are :

BUTTON | MOVEMENT | CURSOR | CLIP | ACCELERATE | SUSPEND

‘BUTTON’ indicates that we are interested in receiving events when a mouse button is pressed. ‘MOVEMENT’ is for receiving events when the mouse is moved. ‘CURSOR’ is for drawing the cursor on the screen. ‘CLIP’ is for keeping the coordinates within the screen size specified in the ‘getscreen’ parameter. ‘ACCELERATE’ is for accelerating the mouse movement in order to obtain a higher precision of the mouse cursor and ‘SUSPEND’ is for receiving events from a suspended mouse.

Since the driver is used in a polled mode, events are not sent to our DLL. It is therefore only necessary to make sure that we are not supplying the ‘CLIP’ and the ‘ACCELERATE’ parameters when we initialize the driver.

When the initialization is done our mouse driver is polled for the position every time we call the ‘hGetAbsolutePosition’ function. If the mouse is moved between the polls the change in position is accumulated inside the driver stack. So by using these two functions and calling them from an outside program (in this case a DLL) we can get the data needed to reposition the scan planes.

Finally 'hCleanUp' can be called to kill the listening thread inside the driver, free up the used resources and deallocate all the mouse devices we have initialized.

4.7 Dynamic Link Libraries

A DLL (Dynamic Link Library) is a way of encapsulating a piece of a program code in Windows. A DLL makes it possible to provide functionality to other programs without revealing the internal code, and it makes it possible to access (link) on demand, hence the name DLL. Several programs can use the same DLL simultaneously. Variables are as default local and not shared, but a shared memory segment can be declared in the DLL. This makes it possible to transfer data from one program to another.

In the case of Sonocubic and Terason a file called 'uslink.dll' is loaded at runtime. This DLL is loaded completely independent by either program and is in this case loaded twice, once by Terason and once by Sonocubic. There is no connection between the programs so far (data is by default local). Inside the DLL there is a declaration of the name of the shared memory. The operating system (Windows) allocates the memory space for the first program that loads the DLL. Since the memory space is shared, another program can also load the DLL and it will get access to the same memory space. Windows makes sure that none of the programs write to the memory space at the same time, so it is not necessary to provide any metaphores or other security features for this. After this setup it is possible to transfer data through the shared memory space. For transfering the scan planes the DLL called 'uslink.dll' is used. I have not been involved in developing the 'uslink.dll', and the code for it has not been modified by implementing the positioning system.

The code for the DLL that is used to interface to the mouse driver, 'Pos3D.dll', can be found in appendix B. 'Pos3D.dll' provides an interface between Sonocubic and the mouse driver. Using the DLL, Sonocubic can link it at runtime and access the functions that it provides. Below is an overview and a short description of the functions that it contains.

```
POS3DAPI int __stdcall Init(void)
```

This function initializes the driver stack using the ‘hInitialise’ function in the hapi (high level API) part of the CPN mouse driver described earlier.

```
POS3DAPI int __stdcall UnInit(void)
```

This function calls ‘hCleanUp’ and frees up resources used in the mouse driver.

```
POS3DAPI int __stdcall GetPosition(PositionDesc *pos)
```

This function calls ‘hGetAbsolutePosition’ and gets the current position for the mice we have initialized. I am calling ‘hGetAbsolutePosition’ twice with two different numbers, since I have designed the software so that two mouse devices can be attached to the system. One will give the XY-position (or XZ-coordinates Sonocubic terms) and another will give the tilt and angle information (in the form of a gyromouse). But the tilt and angle information can currently not be used in the 3D software and so far the mounting of gyroscopes for obtaining angle information is very bulky (more about this in chapter 6 - Conclusions and Future Work).

```
POS3DAPI int __stdcall LastPos(PositionDesc *pos)
```

This function was created in order to be able to monitor the positions acquired by ‘GetPosition’ (that Sonocubic is calling). It simply makes a lookup in a shared memory of the last position that was saved when ‘GetPosition’ was called.

```
POS3DAPI int __stdcall SetResolution(MouseResolution *res)
```

This function stores a number that converts the pixel count from the mouse, so it fits the scale that is needed in Sonocubic. In my own opinion I think this conversion factor should be located inside Sonocubic. But it is a little difficult to argue for, since the big picture of how the Sonocubic software is structured is unknown to me.

```
POS3DAPI int __stdcall GetResolution(MouseResolution *res)
```

‘GetResolution’ gets the number that I stored with ‘SetResolution’

```
POS3DAPI int __stdcall GetDLLVersion(int *DLLversion)
```

This function is intended to provide a number that indicates what version of the DLL that is being used.

4.8 Conversion Factors for Image Resolution

As mentioned in the description of the 'Pos3D.dll', Sonocubic is storing a conversion factor using the ‘SetResolution’ function. The conversion factor makes sure that the scan planes are positioned correctly inside the 3D voxel array in Sonocubic.

The value of the conversion factor is derived from the resolution of the ultrasound image that Terason transferred to Sonocubic. The resolution is found in the BMP-header and describes how many pixels that are used per meter.

Inside Sonocubic the data is collected scan plane by scan plane. Each scan plane is an image that contains pixels that describes the ultrasound image. If there is no positioning system attached to Sonocubic (no 'Pos3D.dll' in the Sonocubic directory) these scan planes are placed directly in a data matrix that we call a maingrid. In the maingrid the image data is describing a volume rather than an area, so they are referred to as voxels instead of pixels. The distance between the voxels in the maingrid are the same in all three dimensions.

If there is a positioning system attached to Sonocubic the scan planes are first placed in a finer subgrid and then projected onto the coarser maingrid. The subgrid is a matrix consisting of 65536 points in each dimension between the maingrid points in adjacent maingrid scan planes. In order to get a correctly dimensioned image it is therefore necessary to place the acquired scan planes correctly, and that is what the conversion factor is used for.

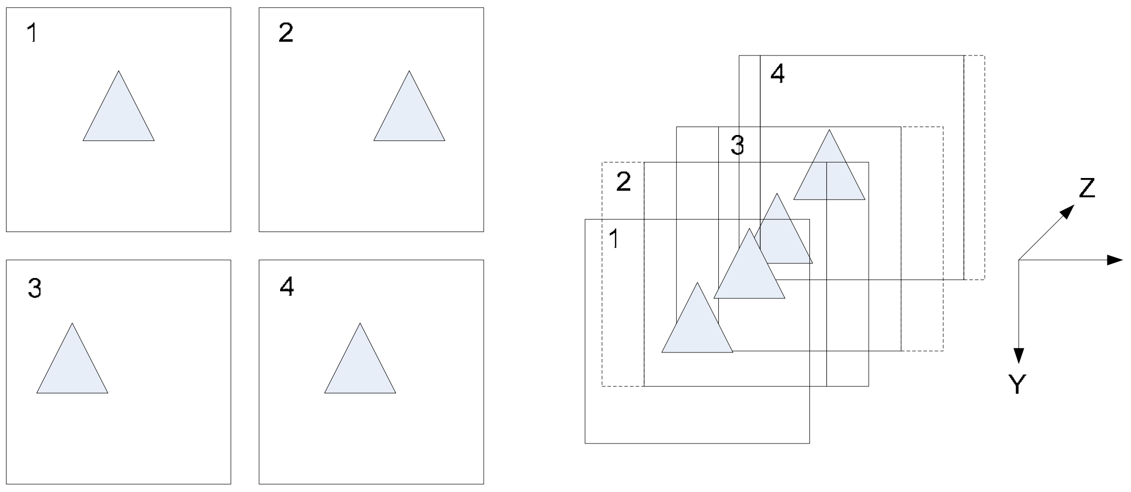


Fig. 45 - Repositioning of scan planes in the X-direction

In Fig. 45 the concept of placing the scan planes in the subgrid is shown. To the left four scan planes scanned with four different offsets and distance between them are shown. After reading the position and multiplying with the conversion factor they can be placed correctly in the subgrid, which is shown to the right.

When the scan planes are in place in the subgrid a nearest neighbor interpolation algorithm is used to find the value of the voxels in the maingrid. The concept is depicted in Fig. 46. 4 points are selected in each of the two subgrid scan planes closest to the maingrid scan plane, giving a total of 8 points being used for the interpolation.

The interpolation takes place directly after a complete scan has been acquired (that is all scan planes). This typically takes between 5-10 secs in the high resolution mode on an AMD Athlon 64 3500+ machine, but the interpolation time also depends on how many scan planes that were acquired during the scan or, said in another way, the duration of the scan. After doing the interpolation the image is represented by the data in the maingrid.

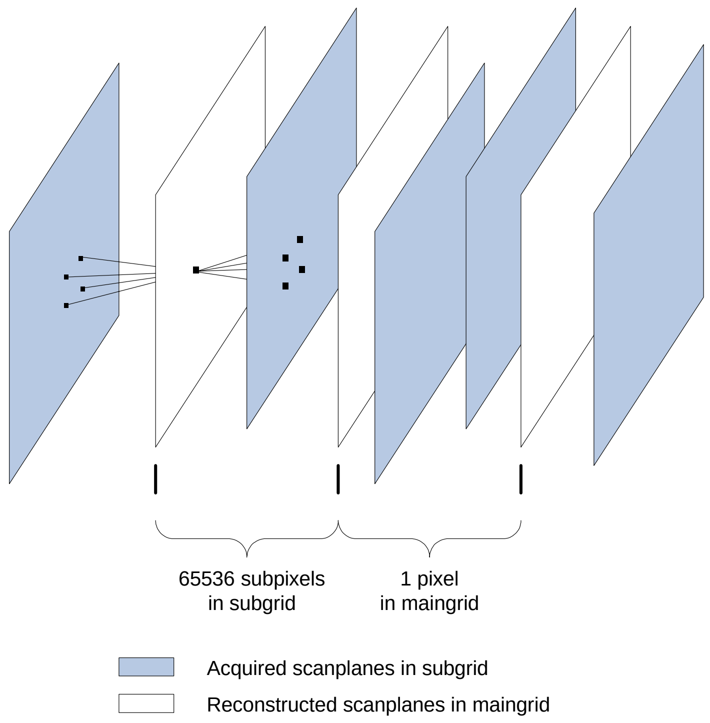

65536 subpixels in subgrid
1 pixel in maingrid
Acquired scanplanes in subgrid
Reconstructed scanplanes in maingrid

Fig. 46 - Interpolating voxels on maingrid from voxels on subgrid

Below is how the conversion factor is found for the maingrid:

$$\text{Mouse}_{\text{Resolution}} = \frac{25.4\,\frac{\text{mm}}{\text{inch}}}{400\,\frac{\text{counts}}{\text{inch}}} = 0.0635\,\frac{\text{mm}}{\text{count}} \qquad \text{Eq. 9}$$

$$\begin{aligned}\text{ConvFactor}_{\text{MainGrid}} &= \text{Mouse}_{\text{Resolution}} \cdot \text{Terason}_{\text{Resolution}} \\ &= 0.0635 \cdot \text{Terason}_{\text{Resolution}} \left[\frac{\text{mm}^2}{\text{count}\cdot\text{pixel}}\right]\end{aligned}$$

We are, however, interested in finding the conversion factor for the subgrid, and we therefore have to multiply the maingrid conversion factor with 65536. This conversion factor is the value that is stored in the 'Pos3D.dll' with ‘SetMouseResolution’ and extracted with ‘GetMouseResolution’. If, for example, the image resolution is 1000 pixels per meter, the conversion factor will be

$$\begin{aligned}\text{ConvFactor}_{\text{SubGrid}} &= \text{ConvFactor}_{\text{MainGrid}} \cdot 65536 \qquad \text{Eq. 10} \\ &= 0.0635\,\frac{\text{counts}}{\text{pixel}} \cdot 1000\,\frac{\text{mm}}{\text{pixel}} \cdot 65536 \\ &= 4161536.\end{aligned}$$

4.9 Testing Conversion Factors

As part of developing the interface between Sonocubic and Terason, Sonocubic created a test program called ‘Teratest’. This program is a scan plane generator that generates a scan plane and places an incrementing counter in it as shown in Fig. 47. This is good for testing purposes and makes it possible to identify each scan plane that is received in Sonocubic.

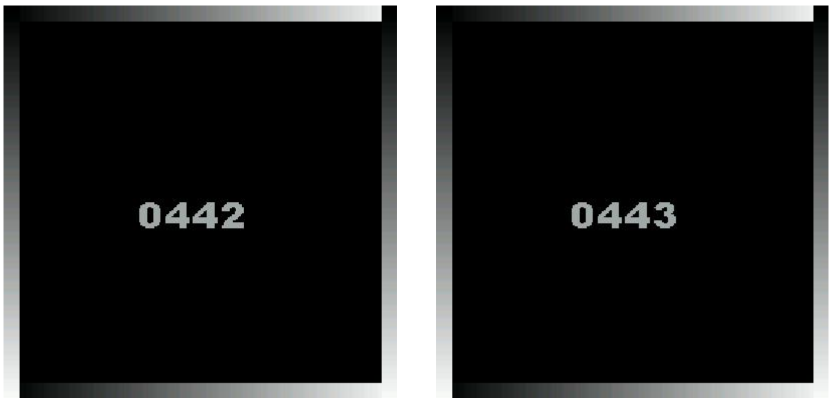


Fig. 47 - Images from scan plane generator, Teratest

In the customized version of Sonocubic that uses the positioning sensor, one cannot, however, because of the interpolation algorithm, assume that each scan plane is placed in the maingrid. The scan plane generator is therefore not useful for testing our algorithm and it is necessary to create another test pattern.

The source code for Teratest was made available to us by Sonocubic, so it was possible to customize it. In this way a computer phantom could be generated or simulated images from an ultrasound image simulator like *Field II* could be used. The latter approach was investigated, but not used because the generation of even a single scan plane in *Field II* takes several hours. The scan planes generated in Field II are also of a very high quality resembling how an actual ultrasound image would look, which is not necessary for testing. A program made in C-code, where ultrasound images were made on the fly, was therefore used to create a three dimensional computer phantom (see appendix D for the source code). This computer phantom was used to test if the conversion factors and the system was working as desired before carrying out a "real world" physical test.

Two different computer phantoms were made : a cuboid phantom, for testing the offset compensation (see Fig. 48(a)) and a pyramid phantom for testing both the offset and the compensation of the distance between the ultrasound scan planes (see Fig. 48(b)). If the algorithm is working the result in Sonocubic should be shapes that perfectly resembles a cuboid and a pyramid. These results will be presented later in this chapter.

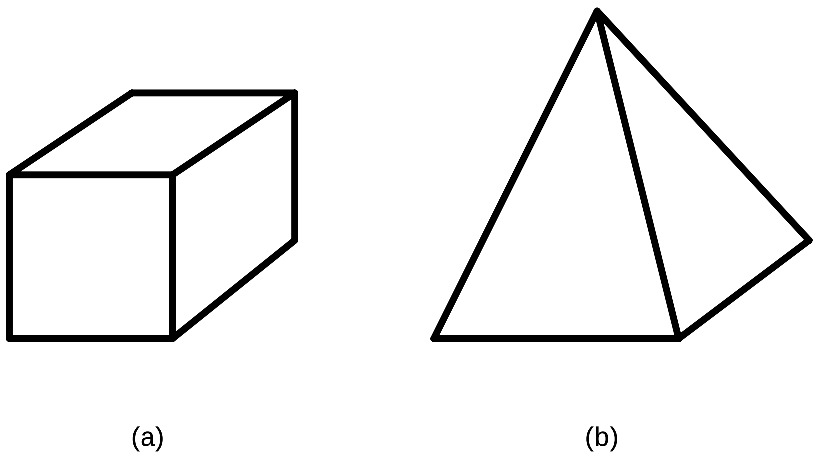


Fig. 48 - Computer generated 3D phantoms

The generated cuboid is in each scan plane a square with the size 60 pixels x 60 pixels. The pyramid starts out as a square with an area that equals 0 and ends having an area that equals 176 x 176 pixels. This gives room for the square in each scan plane to move 40 mm from the center without exceeding the boundaries of the scan plane.

An overview of the test setup can be seen in Fig. 49. The modified version of 'Teratest' is linking with two DLLs : 'Pos3D.dll' and 'uslink.dll'. Based on the longitudinal and lateral position that ‘Teratest’ finds by calling ‘CurPos’ in 'Pos3D.dll' a scan plane is generated. This scan plane has a certain size and offset based on the position. The scan plane is sent to Sonocubic by calling the function ‘SendDIBToSonocubic’ located inside 'uslink.dll' (DIB = Device Independent Bitmap). Sonocubic is then calling another function inside 'uslink.dll' that makes it possible to read the scan plane from the shared memory (the details about this is not known to us). Based on the header information in the first scan plane transferred, the conversion factor is calculated and placed in 'Pos3D.dll' by calling the ‘SetResolution’ function. The conversion factor is used by 'Pos3D.dll' to calculate the correct position of a scan plane in Sonocubic coordinates. Sonocubic thereafter calls ‘GetPos’ for each scan plane to find the correct position of it.

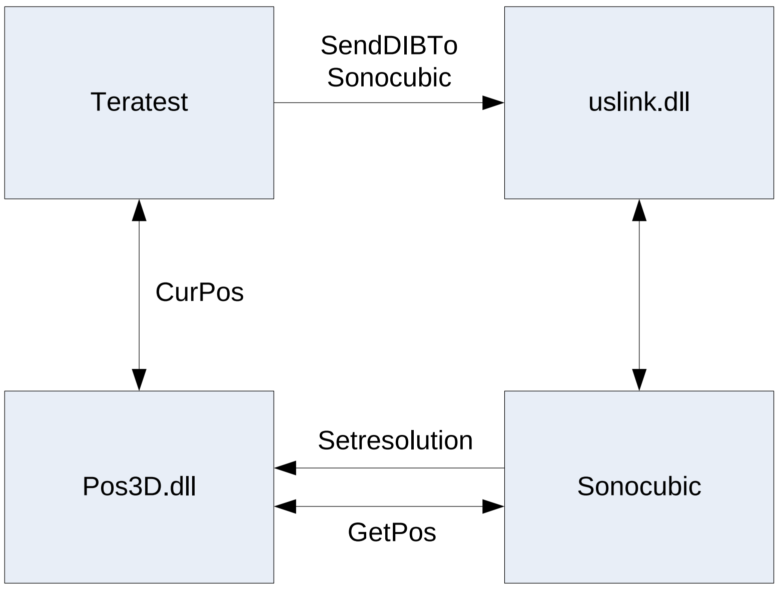


Fig. 49 - Test setup for testing conversion factors (DIB = Device Independent Bitmap)

The 'Pos3D.dll' used for testing is a modified version, since there is no connection to the CPN mouse driver (see appendix E). Instead the positions are generated based on two mathematical formulas, one for the offset and one for the distance.

The lateral offset is termed $s_x(t)$ and is defined in Eq. 11 :

$$\begin{aligned} s_x(t) &= 40\sin(\tfrac{2\pi t}{1000})\ [\text{mm}] \\ &= 1575\sin(\tfrac{2\pi t}{1000})\ [\text{counts}] \end{aligned} \qquad \text{Eq. 11}$$

The longitudinal velocity is termed $v_z(t)$ and is defined in Eq. 12 :

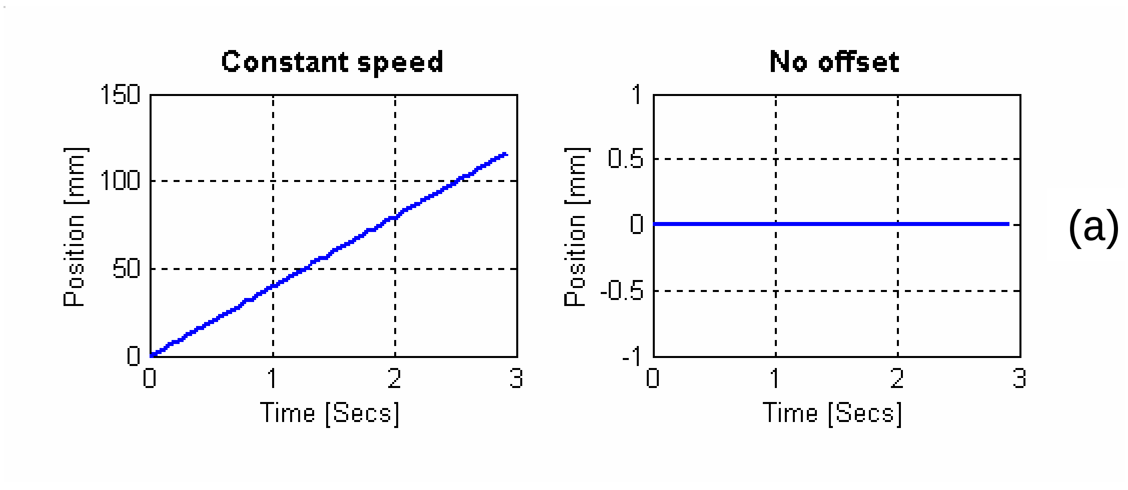
$$\begin{aligned} v_z(t) &= 40 + 20\sin(\tfrac{2\pi t}{1000})\ [\frac{\text{mm}}{\text{sec}}] \\ &= 1575 + 887\sin(\tfrac{2\pi t}{1000})\ [\frac{\text{counts}}{\text{sec}}] \end{aligned} \qquad \text{Eq. 12}$$

The longitudinal distance is $s_z(t)$ where s_z is found from v_z by integration :

$$s_z(t) = 1.575t - \frac{887}{2\pi}\cos(\tfrac{2\pi t}{1000}) + \frac{887}{2\pi}\ [\text{counts}] \qquad \text{Eq. 13}$$

Since the time is used in the formulas it is necessary to pass the current time in msecs every time the position is requested from ‘Teratest’. Based on the time and the formulas above, 'Pos3D.dll' generates a position that is returned back to ‘Teratest’. The generated position is stored in shared memory and is read by Sonocubic when it calls the 'GetPos' function in the DLL (note that this is a different shared memory than the one the scan planes are transferred in). In this way Teratest and Sonocubic gets the same position, and we can check if the compensation algorithm is working.

The reason for giving the time and not just to have an incremental counter is that Windows is not a real time operating system. It is therefore not guaranteed that a frame is generated every 30 ms in Teratest, just as it, in the existing software, is not guaranteed that Terason has finished acquiring an ultrasound image every 30 ms. Supplying the time solves this problem and gives a position that corresponds with the time.



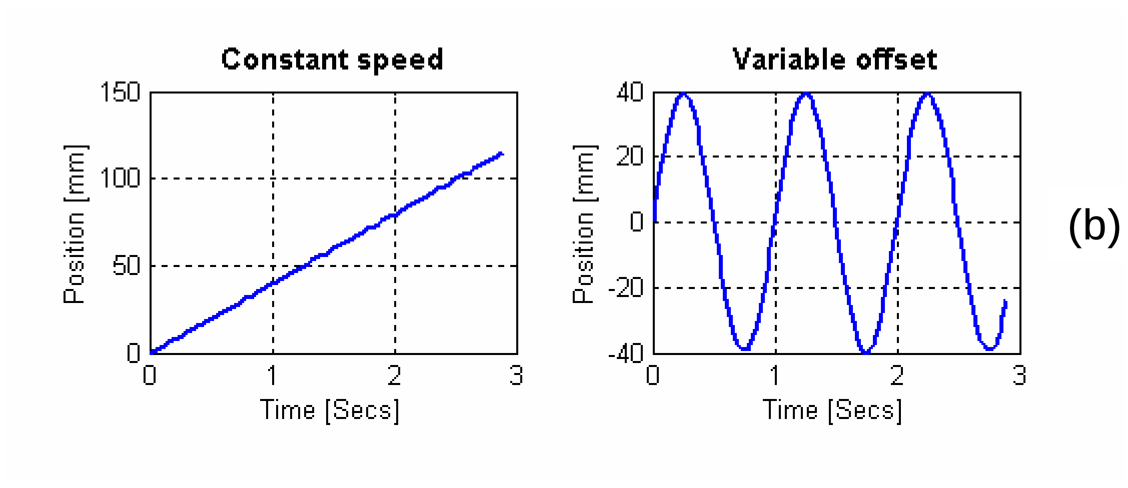
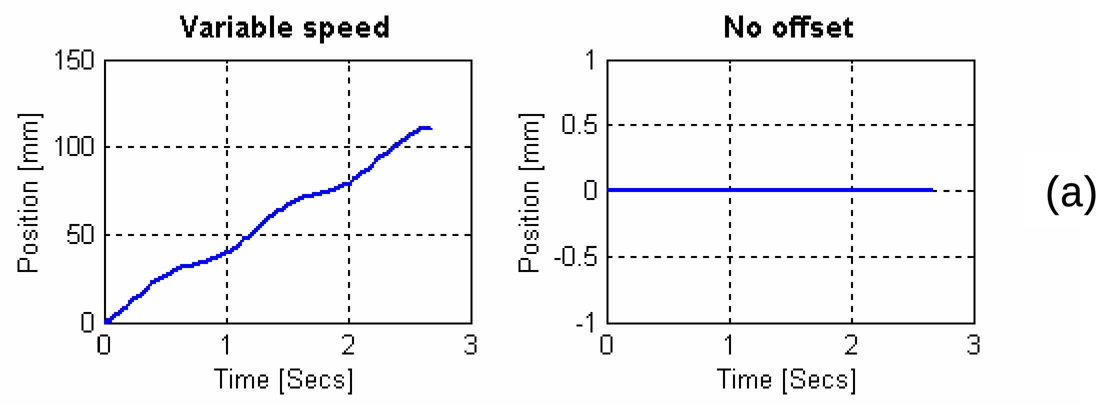


Fig. 50 - (a) constant speed and offset (b) constant speed and variable offset

In Fig. 50 the computer generated movements of the scanner can be seen. These graphs are the actual movements that were generated in 'Pos3D.dll' when the 'CurPos' function was called by 'Teratest'. The data for the position was saved for plotting in Matlab by writing to a file from 'Pos3D.dll'. In the top part of Fig. 50, a perfect movement is shown, that is, a movement without offset or change in scan-speed. This is the movement a user is trying to achieve when a freehand scanning is made. In the bottom part a movement with a constant speed but an offset varying as a sine wave is shown. A sine wave is probably not a very challenging movement for the algorithm, but it has to be remembered that the algorithm is made for assisting a user making a free hand scan. So a random scan would not be very interesting.



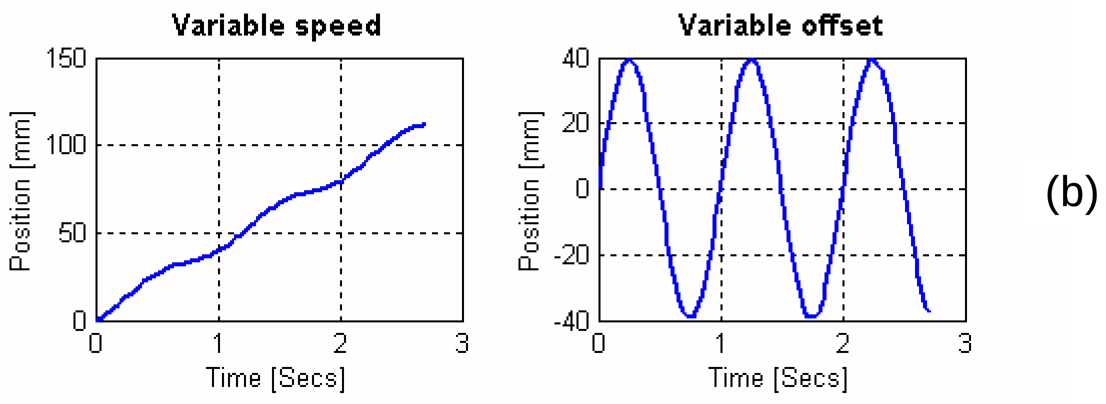
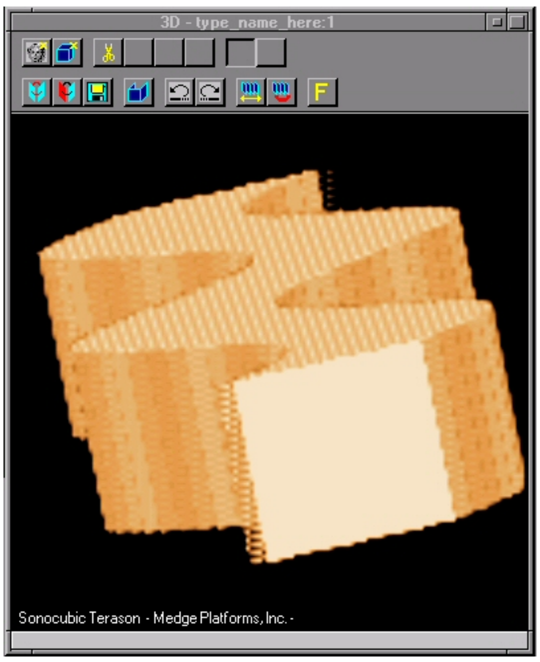
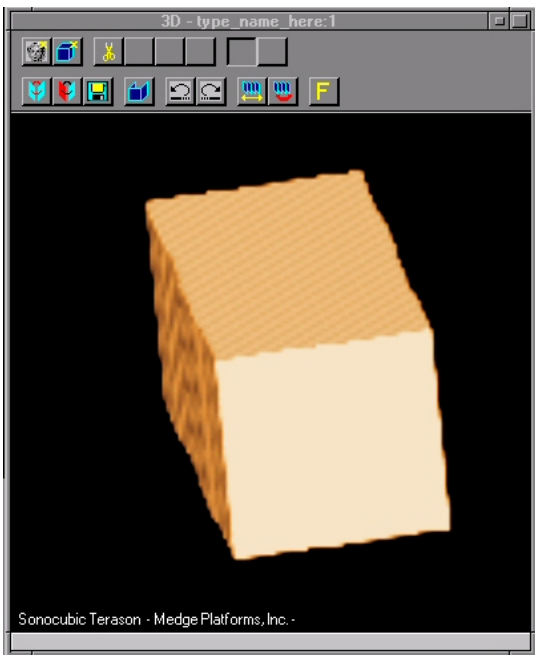


Fig. 51 - (a) Variable speed and no offset (b) Variable speed and variable offset

In Fig. 51 a movement with varying speed but no offset is shown in the top. And the worst case with varying speed and variable offset is shown in the bottom.



(a)



(b)

Fig. 52 - Uncompensated (a) and compensated offset (b)

In Fig. 52 the results can be seen where a scan without offset compensation and another with compensation is shown. It is clear that the correction process is working since it matches the intended volume - a cuboid. The long side of the cuboid is, however, not completely straight. The reason for this is that Sonocubic at this point can position the scan planes with a much higher accuracy than that with which the scan planes can be created. In the scan plane generator it is only possible to place the square on a 256 x 256 pixel grid, and this creates very small rounding errors.

In Fig. 53 the result of scanning the pyramid phantom without compensation can be seen. This corresponds to the movement shown in the bottom of Fig. 51. If a compensation is done only for the offset the result can be seen in Fig. 54 (a), and if both an offset and speed compensation is made the result can be seen in Fig. 54 (b). It is clear that the compensated volume matches the expected volume (a pyramid).

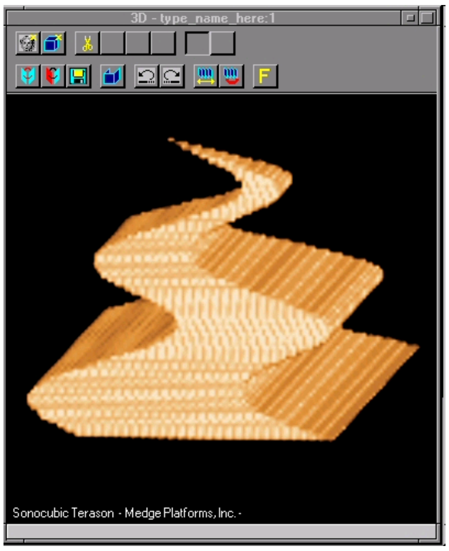
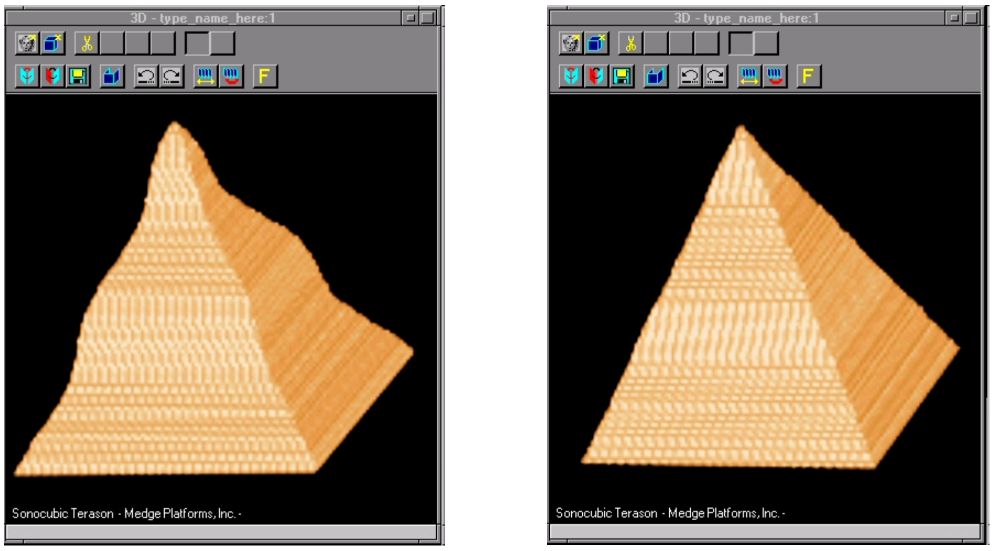


Fig. 53 - Distorted pyramid, no compensation


3D - type_name_here:1
Sonocubic Terason - Medge Platforms, Inc. -
3D - type_name_here:1
Sonocubic Terason - Medge Platforms, Inc. -

(a) (b)

Fig. 54 - Pyramid compensated for offset (a) and for offset and speed (b)

As it can be seen the algorithm is working on computer generated ultrasound images. If we therefore can get an exact position of the transducer it can also work in the real life. This will be tested in chapter 5.

5 Volume Estimations

5.1 Phantom Development

In the previous chapter we tested and proved that the algorithm is working on computer generated data. In this chapter an overall test of the system is done on 3D phantoms. A quantitative test was done by testing the ability of the 3D imaging system to calculate the volumes of inclusions contained in customized 3D phantoms. These 3D phantoms were designed in the Ultrasound Lab by Matt Rowan. The phantoms were made from an organic material called agar. The details that lies behind the development of the phantoms will not be covered in this thesis, but the ingredients used and the recipe that has to be followed in order to create a custom "homemade" phantom will be described.

For making agar, the following ingredients are needed :

1 - 600 mL distilled water

2 - 18 g agar

3 - 0.75g methyl paraben

4 - 54g graphite-powder

5 - 50 mL 1-propanol

For the parts of the phantom that has to resemble a cyst (a hypoechoic volume) the graphite powder is not included. It is the graphite powder that backscatters the sound waves and creates an ultrasound image that looks like an image of human soft tissue.

Using the ingredients mentioned above the following recipe has to be followed :

1 - Heat the distilled water to a minimum of 85°C

2 - Add the agar and mix well with water at the above temperature

3 - Add the methyl paraben and mix well

4 - Cool the mixture to 80°C

5 - At 80°C add the graphite powder and mix well

6 - Allow the mixture to cool to 70°C

7 - Add the 1-propanol.

8 - Degas the mixture under vacuum (to remove internal air bubbles) for 15 minutes while still maintained at 70°C.

9 - Pour solution gently into the mold

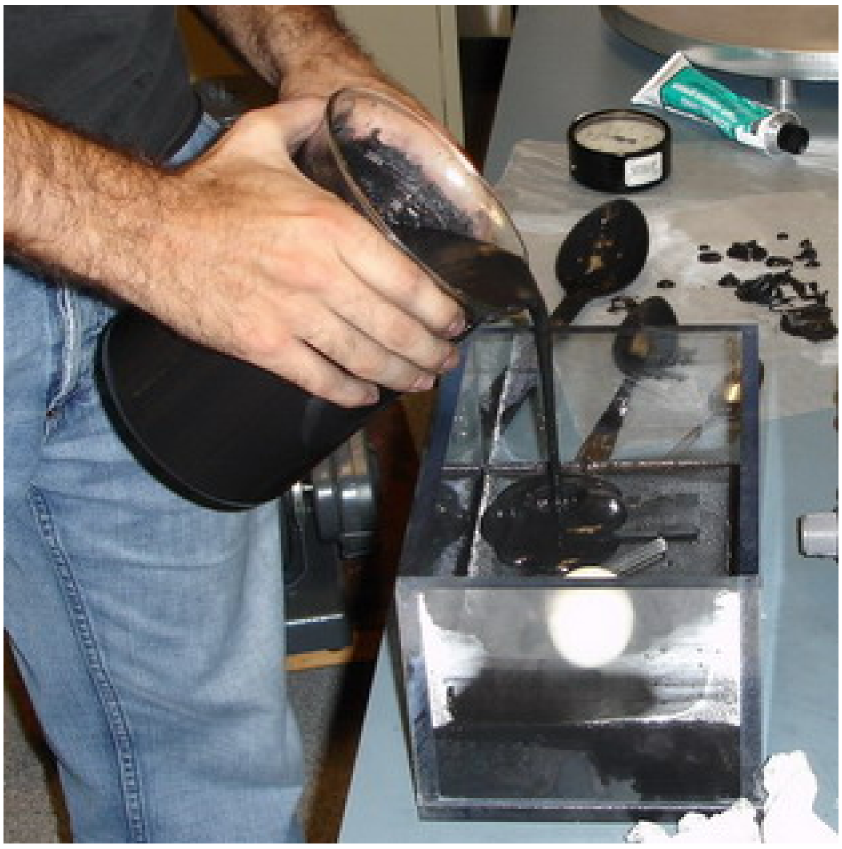


Fig. 55 - Matt Rowan pouring agar and graphite solution into the mold

10 - Cover the mold tightly to prevent ingassing

11 - Let sit for approximately 12 hours

12 - Store the phantom in a degassed environment.

A phantom made like this can be used for 2-3 weeks without losing its properties if it is placed in a cool place and covered to prevent the water from evaporating.

For optical tracking on the surface it is, however, a problem that the surfaces of the phantoms are black and smooth. A black smooth surface is one of the most difficult surfaces to track on, since there are no reflections of the light and no pattern to track on.

After creating several different phantoms we found out that by adding additional methylparaben close to the top surface, it was possible to create a black and white pattern. With this pattern the optical tracking on the phantom surface works properly, and the degradation in the ultrasound image is neglible.

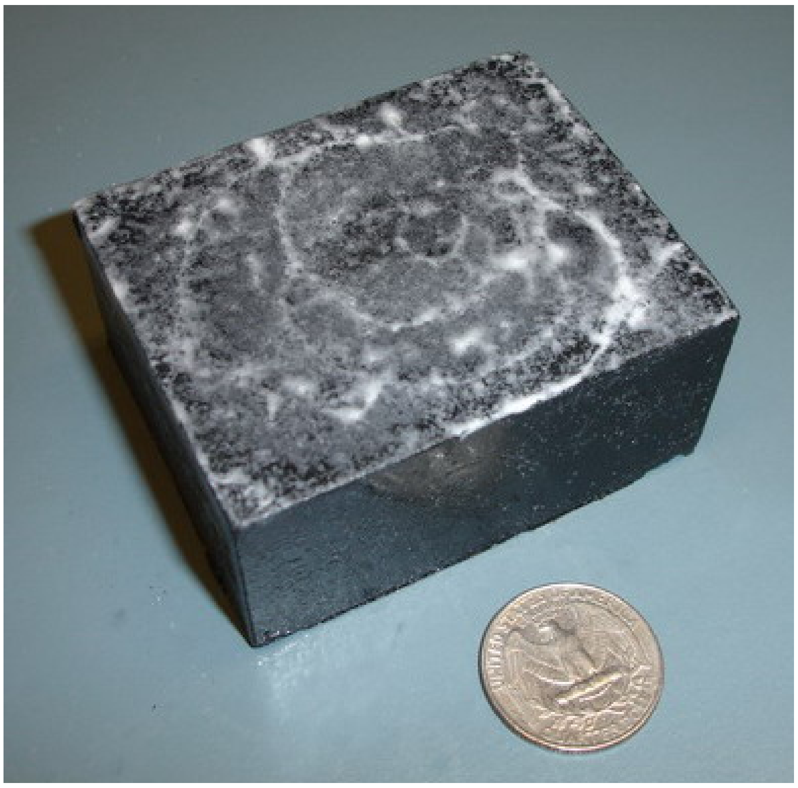


Fig. 56 - Picture of phantom with the surface pattern

5.2 Scanning a Small Inclusion Phantom

A phantom containing a small rod-shaped inclusion of pure agar (i.e., no graphite powder added) was made. The dimensions of the phantom can be seen in Fig. 57

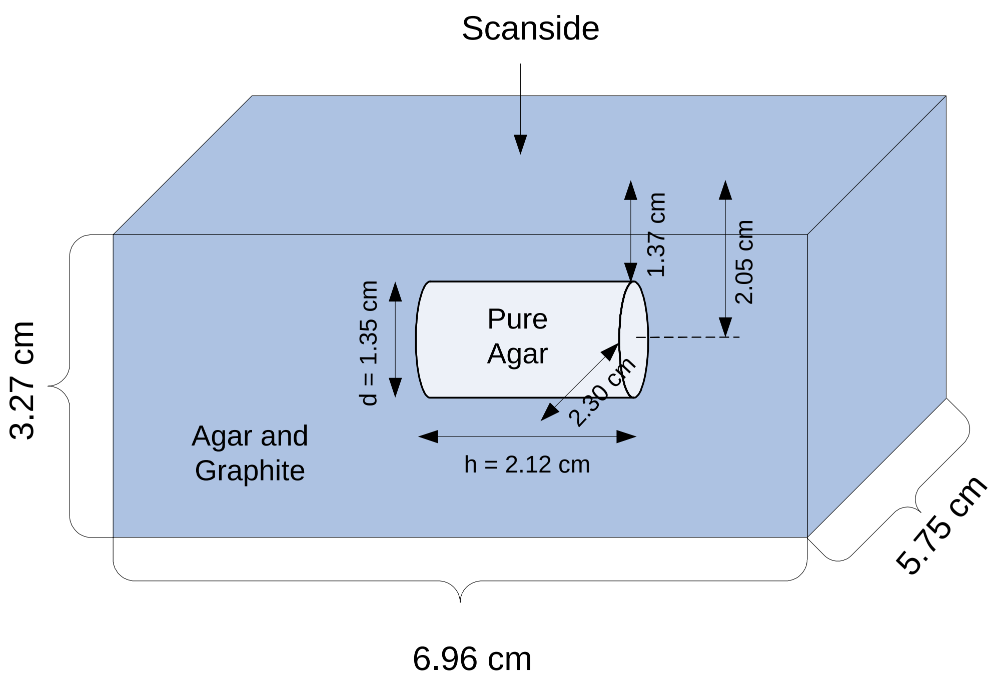


Fig. 57 - Dimensions of custom phantom

Surrounding this rod is an agar/graphite powder solution that ultrasonically resembles the human tissue. Near the top of the phantom is a pattern containing methylparaben that makes it possible to track on the surface. Using the dimensions in Fig. 57 the volume of the rod can be determined to be :

$$Vol_{PureAgar} = \frac{\pi}{4} \cdot d \cdot h = \frac{\pi}{4} \cdot (1.35\ \text{cm})^2 \cdot 2.12\ \text{cm} = 3.03\ \text{cm}^3 \qquad \text{Eq. 14}$$

This volume can therefore be compared to the volume that the system can determine in order to see how accurate it is for volume estimations. For this 5 scans were made, each consisting of between 350 to 414 scan planes or frames. The scans were done in Sonocubic, and after the interpolation algorithm had found the voxel data in the maingrid, a series of 2D scan planes (from the maingrid) were saved in an AVI-file. While working with Ricardo Gayoso on developing the system it was my understanding that all the voxel dimensions in the x, y and z directions were the same. I therefore assumed that if the resolution in the scan plane was known, then the distance between each scan plane was also known. This is in fact the case as long as the data is within Sonocubic, since the data in the main grid is stored in a 512x512x”no of scan planes” array. But a problem occurs when the data is stored in an AVI-file.

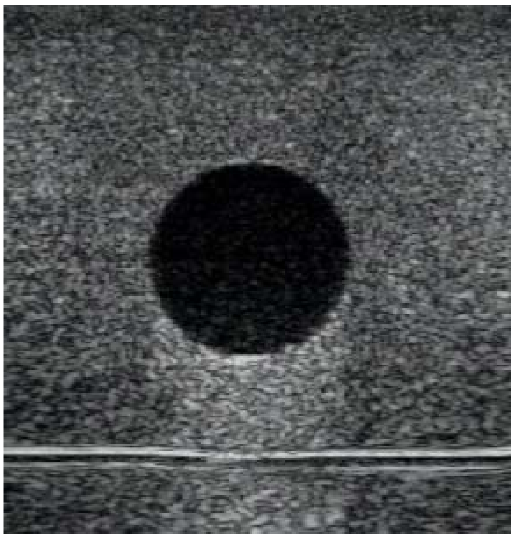


Fig. 58 - 2D scan plane from custom phantom

The way Sonocubic saves the scan planes to an AVI-file is very simple. The conversion is done by capturing the 2D scan plane window as it is displayed on the computer screen (moving the cursor on top of the 2D scan plane window will therefore capture the cursor in the AVI-file). The size of the 2D scan plane window is by default defined by the size of the ultrasound image that is transferred from Terason, and in our case this is 388x388 pixels. The size of the ultrasound image is decided by the Terason application. The AVI-file will therefore also end up having this dimension although the data being displayed actually comes from a dataset with 512x512 pixels. But as this happens the distance between the scan planes is not changed accordingly and we therefore end up with a dataset that has one resolution in two in-plane dimensions and another resolution in the third dimension, that is, the dimension out of the scan plane. Or said in another way the voxels are not cubic anymore but instead as a square cuboid (square box).

The resolution inside Sonocubic can be found by opening the Info->Distance menu item. Here a value called mmxpix can be read to be 0.093964 mm/pixel with the settings that I used. This value depends on the resolution that is transfered in the BMP-header from Terason. So, as mentioned before, Terason defines what the resolution of the 3D image in Sonocubic ends up being. The number of voxels used to describe 1 m^3 can be found to be :

$$res_z = \frac{1}{0.093964\,\frac{mm}{pixel}} = 10642.37\,\frac{pixels}{m}$$

$$res_x = res_y = \frac{388}{512} \cdot 10642.37 = 8064.924\,\frac{pixels}{m} \qquad \text{Eq. 15}$$

$$res = res_z \cdot res_x{}^2 = 10642.37 \cdot 8064.924^2 = 6.92 \cdot 10^{11}\,\frac{voxels}{m^3}$$

This is equivalent to a voxel size of 0.00144 mm^3. Knowing this number makes it possible to relate it to a given voxel count in a volume and thereby to make a translation from voxels to cubic metres. One might ask why the resolution to describe the resolution is in voxels per m^3 and not in m^3 per voxels, since it seems more straight forward to multiply than divide. The reason lies in the definition of the resolution in the BMP-header, where the vertical and horizontal resolution is given in pixels per meter. And to keep things consistent this format for the resolution is kept.

5.3 Matlab Algorithm for Estimating Volume

In Fig. 59 a graphical overview of the steps that has to be done in order to count the number of voxels in a volume is shown. After the repositioning and interpolation of the scan planes is done, an image loop of the 2D scan planes is stored into an AVI-file (video file). The area of interest in the AVI-file is cut out and saved in a file. In Matlab this file is opened and each frame (scan

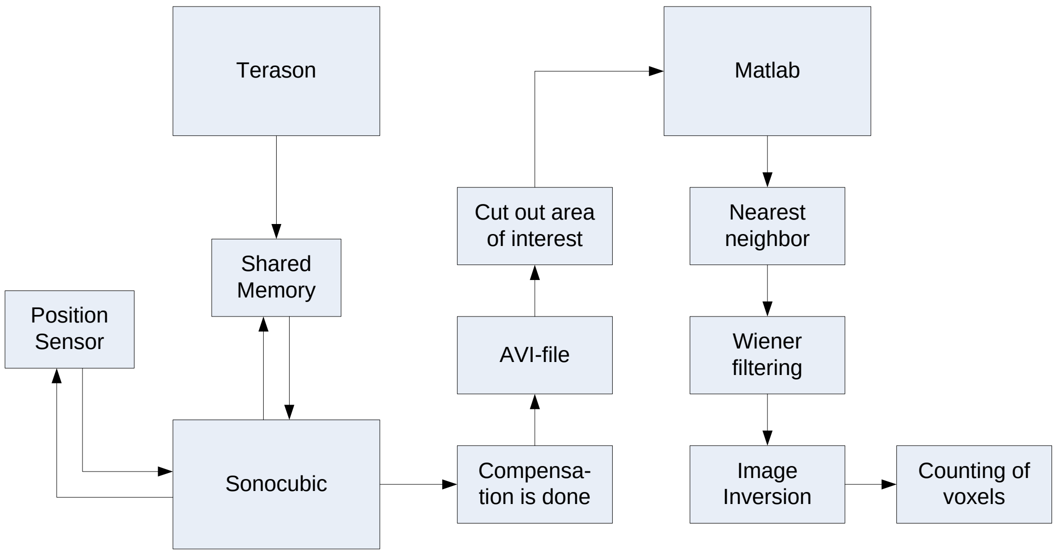


Fig. 59 - Graphical overview of image enhancement algorithm

plane) is processed one by one. First a nearest neighbor algorithm with a thresholding is applied to the scan plane, then a Wiener filtering is done and finally the image is inverted, since this makes it easier to visualize the result. The number of white pixels in the processed scan plane can now be counted. The number of pixels will equal the number of voxels and by counting the pixels in all scan planes the total number of voxels in the volume can be found.

The Matlab code for the algorithm described in short above can be found in appendix C. The nearest neighbor algorithm found here is partly based on Vikramjit Mitras work, a former graduate student in the Ultrasound Lab. In the Matlab code a Wiener filtering with a 5 x 5 neighborhood is used to remove noise. The Wiener filtering is an adaptive filtering, and by using the statistics from the 5 x 5 pixel neighborhood an appropriate low-pass filtering can be done. This filtering removes noise and makes the segmentation much more accurate. The analytical reasons for the choice of neighborhood was not considered but taken directly from an example in Matlab. Since this code provided good results it was decided to keep it as it was. The Matlab code for reading in a frame from the AVI-file, doing a Wiener filtering and a nearest neighbor filtering is shown on the next page.

```
    [RGB,Map] = frame2im(mov(i));
    A = rgb2gray(RGB);
    B = wiener2(A,[5 5]);
    C = ~nearestn(B,1,35);
```

The Wiener filtering that is done first estimates the local mean and variance around each pixel, as shown in Eq. 16 :

$$\mu = \frac{1}{NM-1}\sum_{n_1,n_2\in\eta} A(n_1,n_2)$$
$$\sigma^2 = \frac{1}{NM-1}\sum_{n_1,n_2\in\eta} A^2(n_1,n_2) - \mu^2 \qquad \text{Eq. 16}$$

'A' is the matrix containing the image and η is the N by M neighborhood around the center pixel. In this case both N and M is 5. A pixelwise Wiener filtering then takes place as shown in Eq. 17 :

$$B(n_1,n_2) = \mu + \frac{\sigma^2 - v^2}{\sigma^2}(A(n_1,n_2) - \mu) \qquad \text{Eq. 17}$$

v^2 is a parameter for the noise variance that can be used as an option when the wiener2 filtering function is called. If this parameter is not given, as in this case, the average of the local estimated variances is used. A simpler low-pass filtering and segmentation is subsequently done by calling the function called nearestn. In this function a nearest neighbor algorithm (closest nearest neighbor - 9 pixels) is done. As shown in Fig. 60 the algorithm sums up the surrounding 8 pixels (no. 1-8) and the pixel itself (in the center - no. 9).

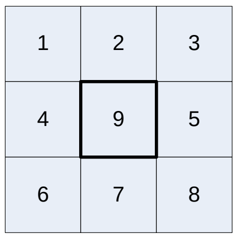


Fig. 60 - Closest nearest neighbor algorithm

If the total value is above a certain threshold the pixel is set to the max value and if it is below it is set to zero. After this a binary conversion is done, giving either a one or a zero for a pixel value. The big advantage of using a binary number is speed.

Following the segmentation all the pixel data is inverted. This is done by inverting all the returned data from nearestn. This makes the pure agar to appear white, as it in reality is visually, but acoustically it is black. It also prevents the surrounding agar/graphite from obstructing the view to the pure agar piece. All the pixels can then be summed up, using the sum command, in order to find the area and ultimately the volume of the pure agar piece.

```
map = [[0,0,0];[1,1,1]];
movproc(i) = im2frame(C, map);
count(scanno) = count(scanno) + sum(sum((movproc(i).cdata))');
```

Count is an array where the index is the scan number being processed. The processed data is stored in a new AVI-file, so the results can be seen by just playing an AVI-movie.

```
outputfilename = sprintf('proc_tube_phantom_w_methylpar_%i_cut.avi',scanno);
movie2avi(movproc,outputfilename, 'colormap', map);
```

Finally the volume is calculated and displayed. As can be seen, the $(388/512)^2$ conversion factor is taken into account when calculating the resolution factor.

```
mperpix = 0.093964e-3;    %Value that can be read in Sonocubic
pixperm = mperpix^-1;
resolution = (388/512*pixperm)^2*pixperm;
volume = count/resolution*100^3;           %in cm3
```

The results for the voxel count are shown in Table 4 below.

Table 4 - Estimation of complete volume based on five separate scans

Scan no.	1	2	3	4	5
Voxels	2143671	2016803	2227925	2156085	2086507
Volume [cm^3]	3.10	2.91	3.22	3.11	3.01
Normalized volume	1.02	0.96	1.06	1.03	0.99

Another similar test was done, but here only used 100 scan planes was used. Doing this eliminates problems that might occur at the end of the phantom, where there is a short transition when the scanner is moved over the pure agar volume. 100 scan planes has a volume of :

$$h_{100ScanPlanes} = 100 \cdot 0.093964 \tfrac{mm}{pixel} = 0.94 \text{ cm}$$
$$Vol_{100ScanPlanes} = \frac{\pi}{4} \cdot d \cdot h = \frac{\pi}{4} \cdot (1.35 \text{ cm})^2 \cdot 0.94 \text{ cm} = 1.35 \text{ cm}^3. \qquad \text{Eq. 18}$$

I got the following results from my measurements :

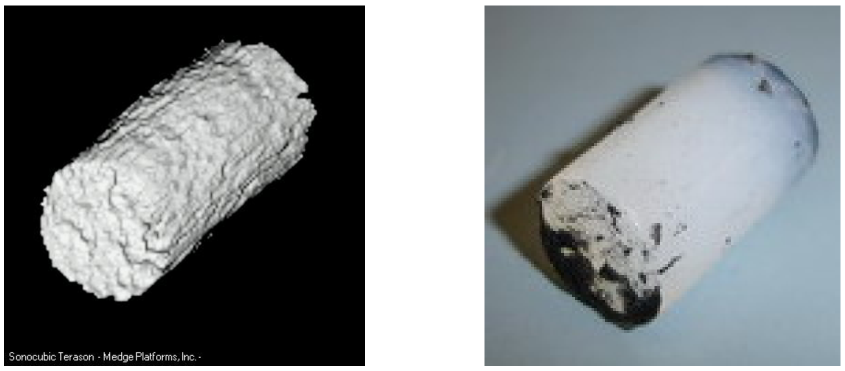
Table 5 - Estimation of volume in 100 scan planes based on five separate scans

Scan no.	1	2	3	4	5
Voxels	934220	927794	932341	939921	949513
Volume [cm^3]	1.35	1.34	1.35	1.36	1.37
Normalized volume	1.00	1.00	1.00	1.01	1.02

As it can be seen the estimated volumes are very close to the actual volume. Measuring the dimensions of the pure agar volume is a potential error source since it is soft. Also we do not know for sure if the pure agar body changes shape slightly when the agar with the graphite powder is poured on top of it. But the results lie within a very small error margin, and it can be seen that the measurements are consistent and can be repeated from scan to scan.

5.4 Phantom Visualization

In addition to the ability to quantify the size of a volume, another important aspect of 3D ultrasound is to display the processed data in a way that makes visualization easier. In Fig. 61 (a) is shown a reconstruction in Sonocubic of the data that was captured using our positioning system and processed to remove noise in Matlab. In Fig. 61 (b) is a photograph of the pure agar rod after it was taken out of the phantom (the black parts are from the agar and graphite solution). As it can be seen, there is a good agreement between the 3D ultrasound image and the actual photograph.



(a) (b)

Fig. 61 - Reconstructed 3D ultrasound image (a) and actual photograph (b) of the pure agar rod

5.5 Human Body Visualization

To test the system's ability to scan the human body it was decided to scan the common carotid artery. The carotid arteries carry blod to the brain and resides on each side of your neck. Because of their accessibility and proximity to the skin surface they are ideal for scanning with an ultrasound scanner. A medical illustration of the right carotid artery is shown in Fig. 62.

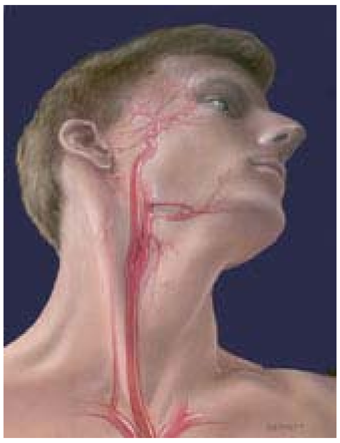


Fig. 62 - Medical illustration of common carotid artery

Scanning with the 3D positioning system described in this thesis makes it possible to visualize the common carotid artery. Since the volume of this artery is unknown the quantitative abilities of the system can not be tested. A scan was made of my right common carotid artery and the compensated scan planes were stored in an AVI-file. As it can be seen in Fig. 63 the ultrasound image is far from as homogeneous as the one of the phantom.

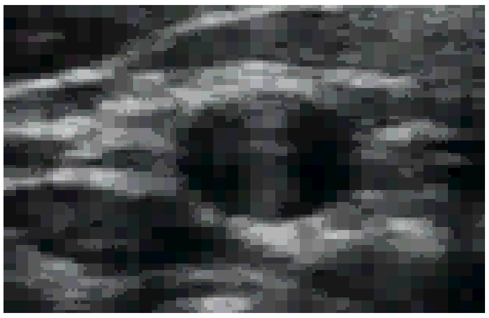


Fig. 63 - Cut out from ultrasound scan plane of common carotid artery

Because of the poor image quality it was decided not to use the Matlab algorithm, but instead a segmentation and boundary detection was done by manually cutting out areas of interest in each scanplane (a total of 226 images). This is a really time consuming process but provides very nice results. In Fig. 64 the reconstructed 3D image can be seen from 4 different views.

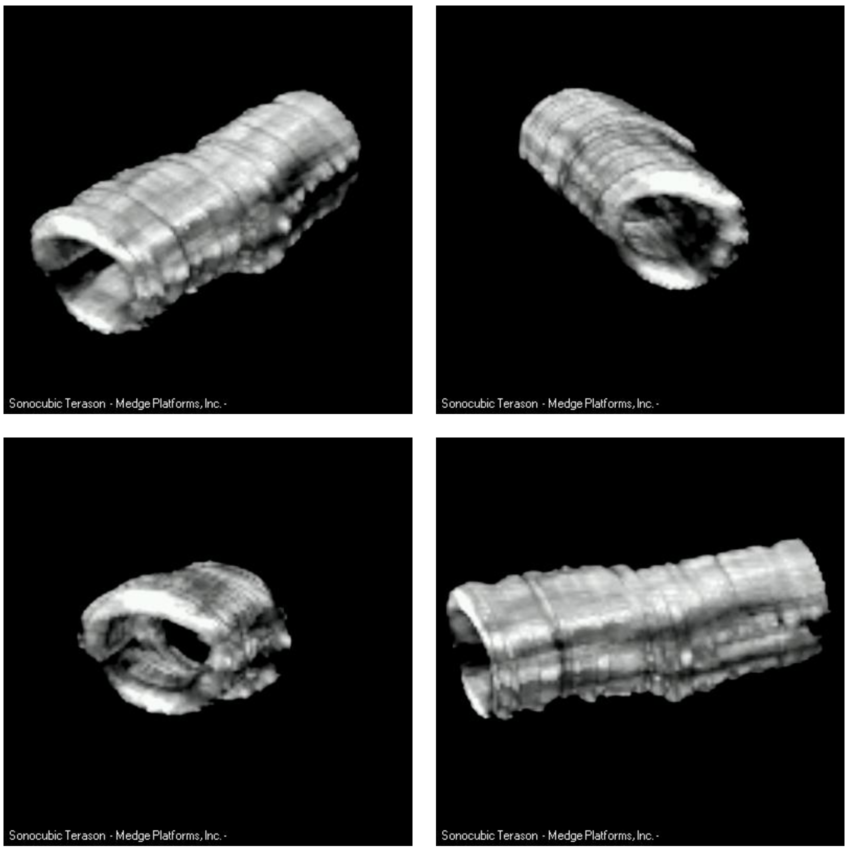

Sonocubic Terason - Medge Platforms, Inc. -
Sonocubic Terason - Medge Platforms, Inc. -
Sonocubic Terason - Medge Platforms, Inc. -
Sonocubic Terason - Medge Platforms, Inc. -

Fig. 64 - 3D reconstructed ultrasound images of carotid artery

As it can be seen the system presented in this thesis makes it possible to get 3D ultrasound images not only from an ultrasound phantom but also from an artery in the human body. The process of reconstruction is because of the manual boundary detection far from automatic, but with better boundary enhancement algorithms integrated into the 3D software it should be possible to make scans as described here in the near future.

6 Conclusions and Future Work

6.1 Conclusions

This thesis has shown the successful development of a new simple and portable optical tracking system that can be used with ultrasound transducers, to create quantitative 3D images.

The mounting of an optical fiber in front of the optical sensor showed us that the image is somewhat blurred and degraded compared to an implementation where just the optical sensor and a lens is used. This made it problematic to track on surfaces with a small variation in the optical pattern like the human skin. The mounting method it was decided to use, which consists of just a lens and the optical sensor mounted in a custom housing, works much better on all surfaces and it is possible to make a compact implementation without using the optical fiber.

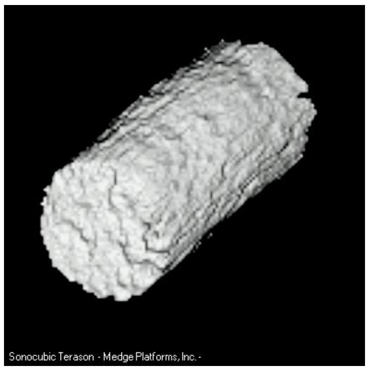
The interfacing of the optical sensor to the software was a success. Reusing the existing hardware and the use of a mouse filter driver eliminated a lot of development work. The use of a USB interface also makes it easy to use the position system in a portable system. Encapsulating the functions for obtaining the position into a DLL provides a good interface. This makes it easy to interface a program that needs the position information. The interface could have been made so it instead used an existing interface, like for example "Flock of Birds". In this way it would be easier for existing 3D programs to interface to our system. The changes can, however, easily be made.

The testing of the system with a computer phantom was a very useful idea. This test made it possible to make sure that all the conversions factors used were in place before a real world test on a 3D phantom was made.

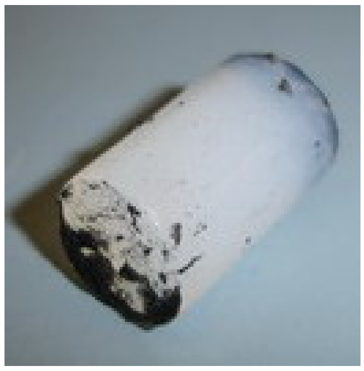
The phantom development proved to be essential for testing the performance of the positioning system. To have this expertise in house was really nice, since it made it possible to customize the phantoms. The ultrasound images are really nice and clear.

A simple Matlab algorithm consisting of a series of low-pass filters was used for boundary enhancement and volume estimation. The algorithm was able to determine the volume of the phantom within 6% by counting the number of voxels. If only the center part of the rod is used (100 scan planes) the accuracy gets as low as 2%. This shows that the system cannot only be used for visualization purposes, but also as a quantative tool. The good results were possible because of the high accuracy of the position sensor, the clear ultrasound images it was possible to obtain from the phantoms and the Matlab filtering algorithm. One problem with the Matlab algorithm is that it is difficult to implement in for example C-code, since it uses predefined Matlab functions. An implementation in C-code could be necessary if an "on the fly" algorithm is needed, for example when transfering the images from Terason to Sonocubic.

Using Sonocubic the processed data can be visualized. The processed data resembles the actual phantom quite accurately as shown in Fig. 65 (Note that (a) is inverted).



(a)



(b)

Fig. 65 - Reconstructed 3D ultrasound image (a) and actual photograph (b) of the pure agar rod

The system has some limitations since it is limited to two degrees of freedom. As we have demonstrated in this thesis it works fine on a flat surface. But scanning for example over the curved skin surface of a pregnant woman remains a problem.

6.2 Future work

As future work we suggest that the system is enhanced with capabilities to take angle information into account. This can be done using micro gyros [14]. The microgyros will make it possible to add angle information to each ultrasound scan plane giving up to 5 degrees of freedom, only missing the depth.

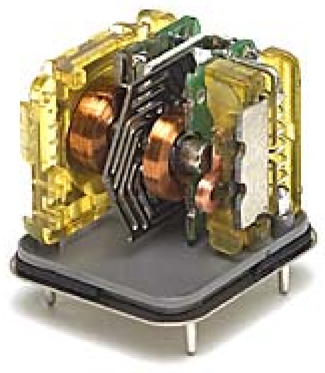


Fig. 66 - Cut through of a micro gyro from Gyration [14]

A preliminary study using a Gyro mouse (a GP210-001) from Gyration showed that the angle information can be obtained and displayed from the gyro mouse using the CPN mouse driver. Sonocubic is, however, not able to take the angle information into account.

A software package that can use the angle information will therefore be needed. It might be possible for Sonocubic to be customized so the angle information can be taken into account. Sonocubic is, however, an old program and for future purposes newer technologies should be used. This could for example be in the form of a program that could use 3D graphics hardware. This would make it possible to get high 3D performance by using commercial hardware that is available at a low cost.

Another thing that could be incorporated into a new program would be image enhancement algorithms. A thing similar to this has already been done in Volview, that is a 3D program by Kitware [15]. Volview has a series of standard medical filters and customized filters can be interfaced to the program through an API interface. This provides a good platform for development purposes.

References

[1] Development of a Field-Deployable Voice-Controlled Ultrasound Scanner System
Master thesis by Dalys Sebastian, Worcester Polytechnic Institute, June 2004

[2] Ascension Technology - Flock of Birds
http://www.ascension-tech.com/products/flockofbirds.php

[3] Polhemus - Patriot
http://www.polhemus.com/PATRIOT.htm

[4] 3D Imaging in Medicine by Jayaram K. Udupa, Gabor T. Herman, CRC Press, Sep 28, 1999

[5] Edge detection
http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MORSE/edges.pdf
Last date visited : 7/14/04

[6] Pointer Ballistics for Windows XP
http://www.microsoft.com/whdc/device/input/pointer-bal.mspx
Last date visited : 11/23/04

[7] CPN mouse filter driver
http://cpnmouse.sourceforge.net/
Last date visited : 11/23/04

[8] Agilent 20x Surface Tracking Improvement for Laser-Based Mice
Technical Overview
http://cp.literature.agilent.com/litweb/pdf/5989-1535EN.pdf
Last date visited : 1/3/05

[9] Datasheet for Agilent ADNS-2610 Optical Mouse Sensor
http://wpi.dk/thesis/datasheets/ADNS-2610.pdf

[10] Datasheet for Agilent HDNS-2100 Optical Mouse Lens
http://wpi.dk/thesis/datasheets/HDNS-2100_Optical_Mouse_Lens.pdf

[11] Pointer Ballistics for Windows XP
http://www.microsoft.com/whdc/device/input/pointer-bal.mspx

[12] Webpage about image conduits and camera
http://bruce.cs.cf.ac.uk/bruce/LVM/Methods%20030-039/LA%20Method%20037/LA%20037-%20text.htm

[13] USAF 1951 Test chart description
http://www.sinepatterns.com/USAF_labels.htm

[14] Gyromouse and microgyros from Gyration
http://www.gyration.com
Last date visited : 11/29/05

[15] Volview by Kitware
http://www.kitware.com/products/volview.html
Last date visited : 11/29/05

[16] Digital Design - Principles and Practices by John F. Wakerly,
Prentice Hall, 2001

[17] Introductory VHDL - From Simulation to Synthesis by Sudhakar Yalamanchili,
Prentice Hall, 2000

[18] VHDL - Programming by Example by Douglas L. Perry, McGraw Hill,
4th edition, 2002

[19] Spartan-3 Starter Kit Board User Guide UG130 (v1.0) April 26, 2004

Appendices

Appendix A - CCD - array image viewer

The CCD-array image viewer was made as a part of my VHDL class EE574 taught by Prof. R.J. Duckworth. The documentation presented here is a part of the report submitted for the final project in this class.

A.1.1 - Block Diagram of Hardware

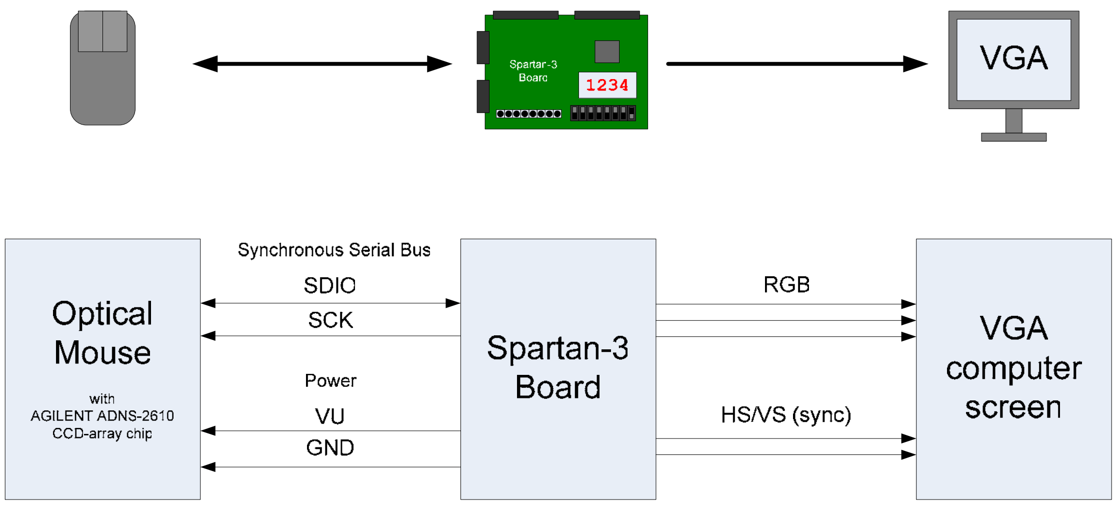


Fig. 67 - Block diagram of hardware used

In Fig. 67 a block diagram of the hardware can be seen. I use an optical mouse from Targus that contains the Agilent ADNS-2610 CCD-array chip (referred to as optical sensor). Through a synchronous serial bus the Spartan-3 developers board can communicate with it. This is done through a master/slave relationship where the Spartan-3 as the master sends out commands to the CCD-array that as a slave responds to the master's orders. Through the VGA display port the Spartan-3 can then display the image it acquires from the CCD-array. Other useful information like for example the value of the maximum pixel or the deviation in pixel values in an image can be displayed on the seven segment displays or on the LEDs.

A.1.2 - Design Methodology

I started this design by taking an old design and stripped of all the things I did not need in it. I thereby ended up with a module containing my state machine, clock conversions and the ability to display numbers in different states.

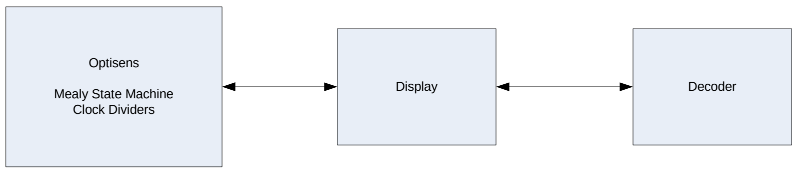


Fig. 68 - Simplified block diagram of modules I started with

This was a good starting point. I like this approach since it is easier to strip down a project than it is to start a new from scratch.

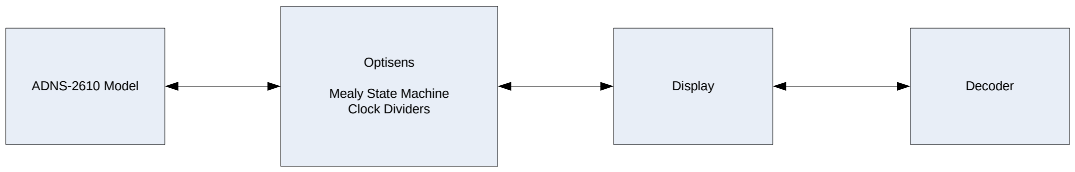


Fig. 69 - Simplified block diagram of model attached to the design

I then decided to make a model for my optical sensor. Even though it might seem a little cumbersome to make a model from scratch I think that I have saved a lot of time in the end, because it made me able to write a test bench and do a simulation of the commands that I sent to the optical sensor. The model is very simple and only answers back with a certain sequence of numbers if I send a specific command, but it is enough to prove that my communication is working. And since I am acquiring image information the data is constantly varying. The model is therefore a good representation of the actual optical sensor.

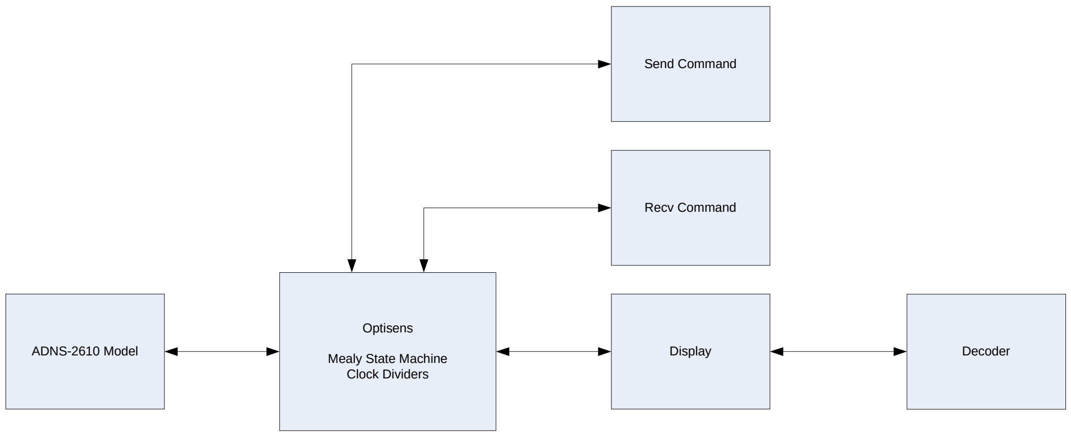


Fig. 70 - Simplified block diagram of design + send and recv command modules

After doing the basics in the optisens module I started working on the send command and receive command modules.

Send command is a module that takes care of sending a command to the optical sensor. After it has done so it sets a flag to indicate to the other module, recv command that it can begin to read a command from the serial synchronous line. Since this is an in/out line from the Optisens module and to the optical sensor it is split up into being an out line from the send command module and to be an in line going to the recv command module.

After doing the basics in the send_command module I then made a test bench for the optisens module. By simulating some commands on the sliders on the Spartan-3 board and simulating button presses to execute them I was able to see in my simulation what the waveform on the synchronous serial port looked like. This was very useful. By sending a specific command from the send_module I could then get a response back and try to interprete it in my recv_command module. Since I knew the response from my model I could then make sure that I got the same response in my recv_command module. I did this by looking at the output to the LEDs.

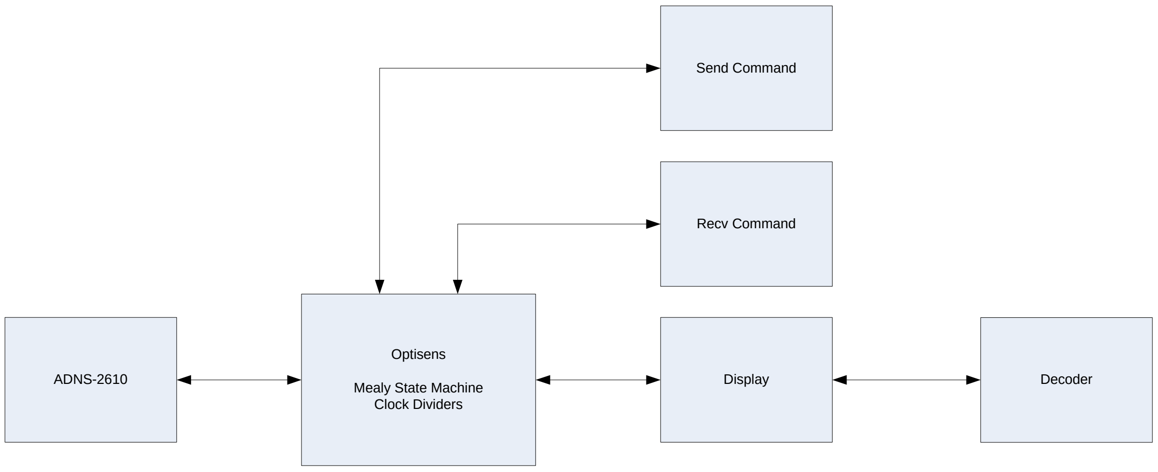


Fig. 71 - Simplified block diagram of ADNS-2610 connected (not the model)

It was now the time to look at my hardware. I connected the wires to the mouse through the expansion connector A2. In Fig. 72 and Fig. 73 the color codes for the actual wires are shown. In Fig. 74 an actual picture of the connections is shown.

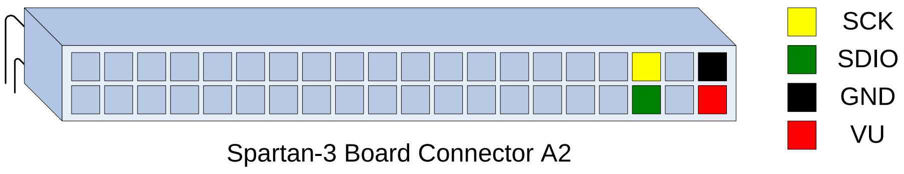


Fig. 72 - Color codes for wires connected to A2 expansion connector

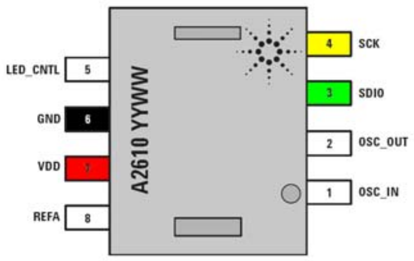


Fig. 73 - Color codes for wires connected to the ADNS-2610 optical sensor

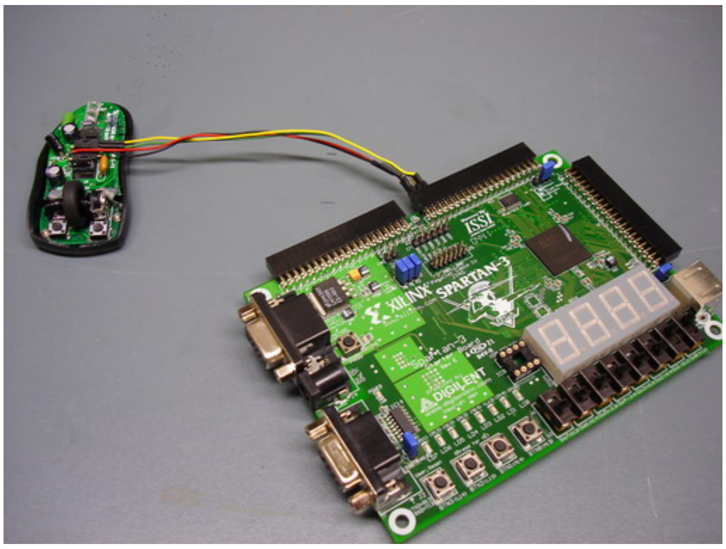


Fig. 74 - Picture of hardware setup

I was now able to get actual data from the mouse and it was working as the simulation said it would. I made it possible to display digits on the display in certain states using my display module.

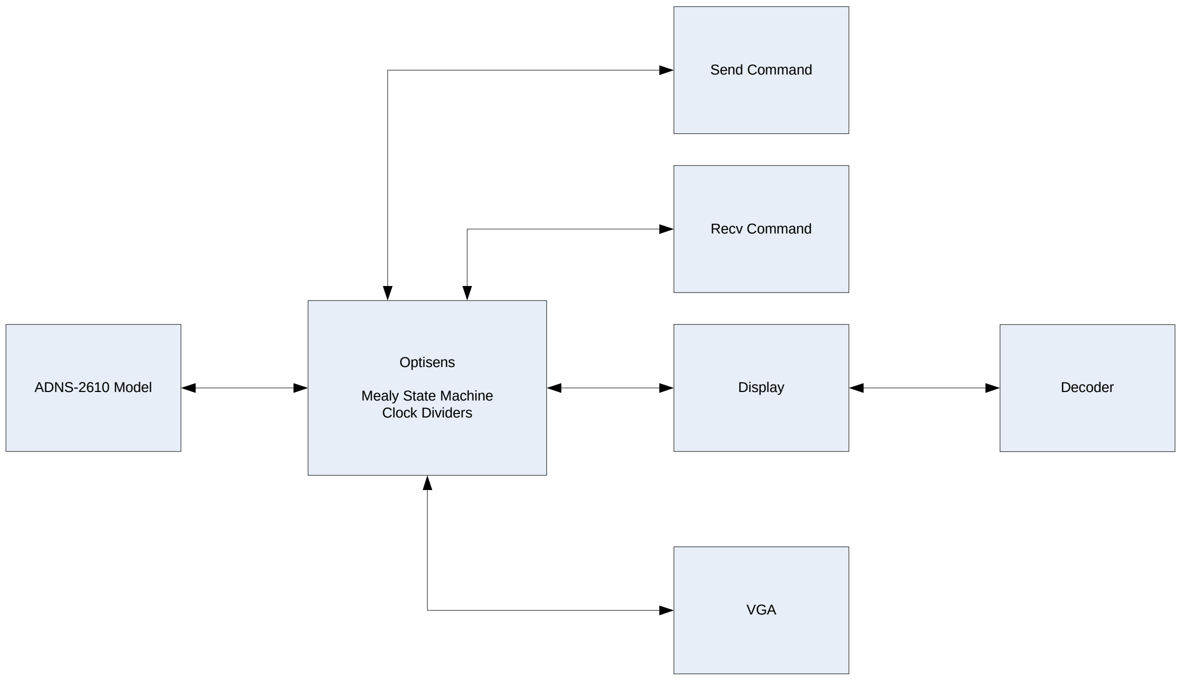


Fig. 75 - Simplified block diagram of final design

Finally I concentrated on the VGA module. The VGA module is quite simple as it basically consists of some clock converters and counters. The key thing to begin with is to get the horizontal and vertical sync signals to work.

Displaying the actual image is done in certain periods of the cycle and I started out by being able to display a constant color like red or green. I had followed the documentation for the Spartan-3 board, but to my surprise something was wrong. The problem is that the front and back porch are wrongly depicted in [19]. The two figures I am mentioning can be seen in Fig. 76 and Fig. 77.

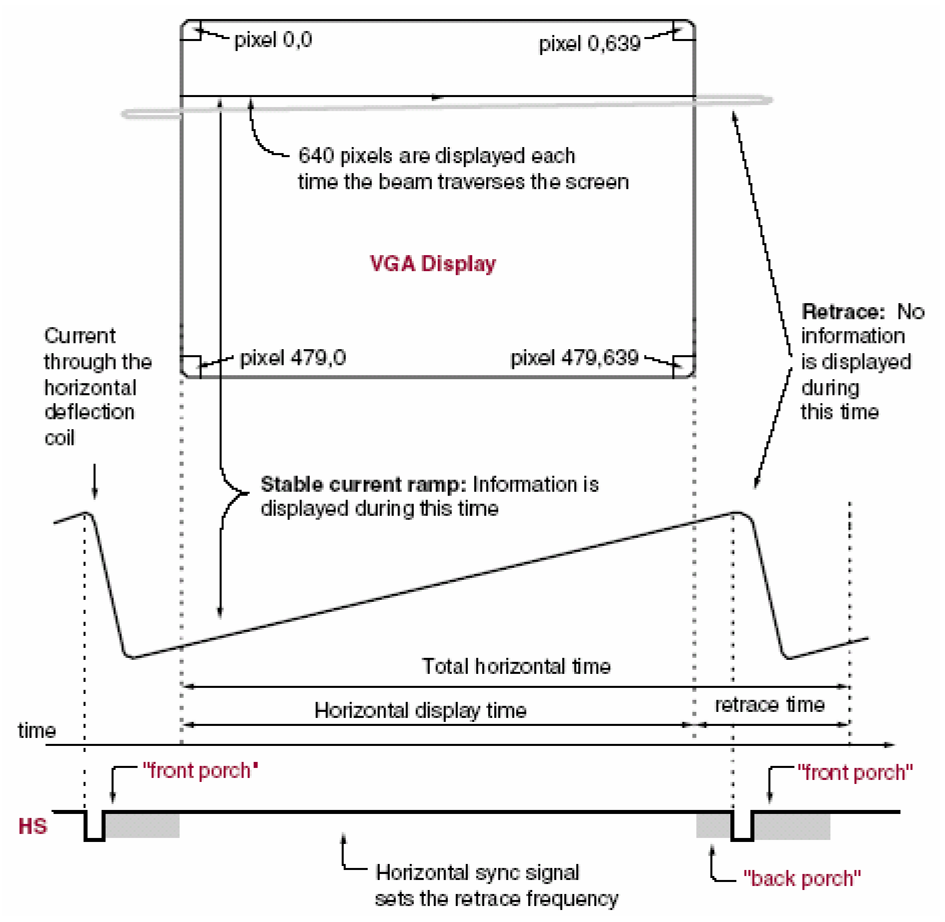

pixel 0,0
pixel 0,639
640 pixels are displayed each time the beam traverses the screen
VGA Display
Retrace: No information is displayed during this time
Current through the horizontal deflection coil
pixel 479,0
pixel 479,639
Stable current ramp: Information is displayed during this time
Total horizontal time
Horizontal display time
retrace time
time
"front porch"
"front porch"
HS
Horizontal sync signal sets the retrace frequency
"back porch"

Fig. 76 - Wrong VGA timing diagram

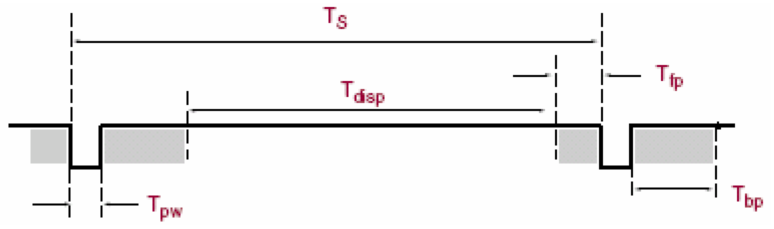


Fig. 77 - Correct VGA timing diagram

The result from switching the back and front porch can be seen as a black vertical stripe on the monitor as shown in Fig. 78 (left) where the result of a correct back and front porch is also shown (right).

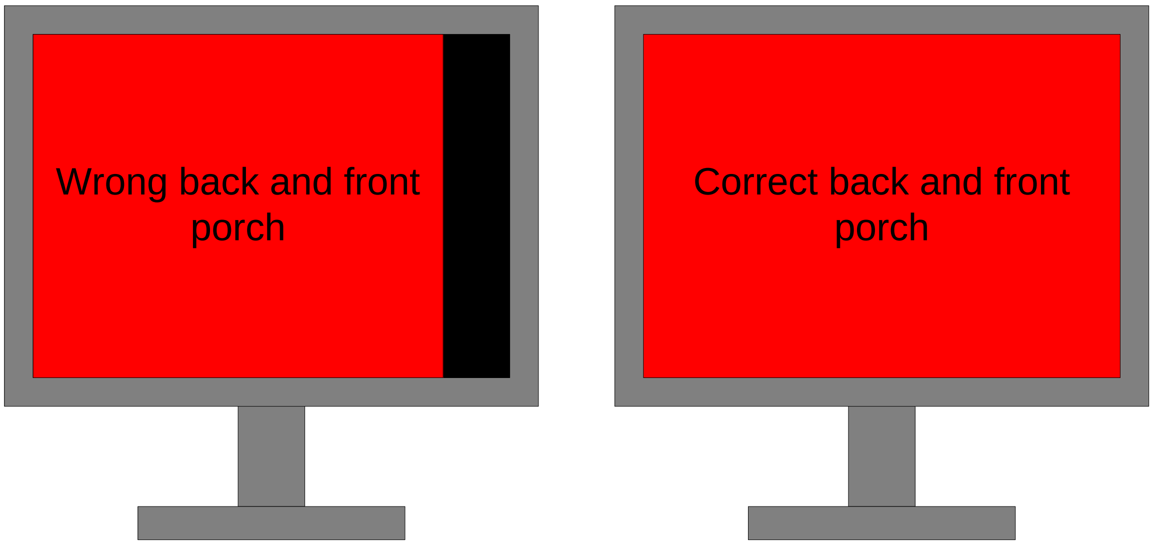


Fig. 78 - Wrong back and front porch (left) and correct back and front porch (right)

After resolving this problem I continued to work on the VGA module, so it would be able to constantly read the results from the recv_command module (given as an output on the LEDs). This data is stored in an array of 18 x 18 = 324 corresponding to the amount of pixels in the CCD-array. Each element in the array contains 6 bits corresponding to the 64 grayscales it is possible to obtain from the optical sensor. This uses a total of 324 x 6 = 1944 flip-flops.

I am then constantly indexing into this array to display the actual image on the screen.

Below in Fig. 79 I have shown a detailed block diagram with all the connections between the modules I am using.

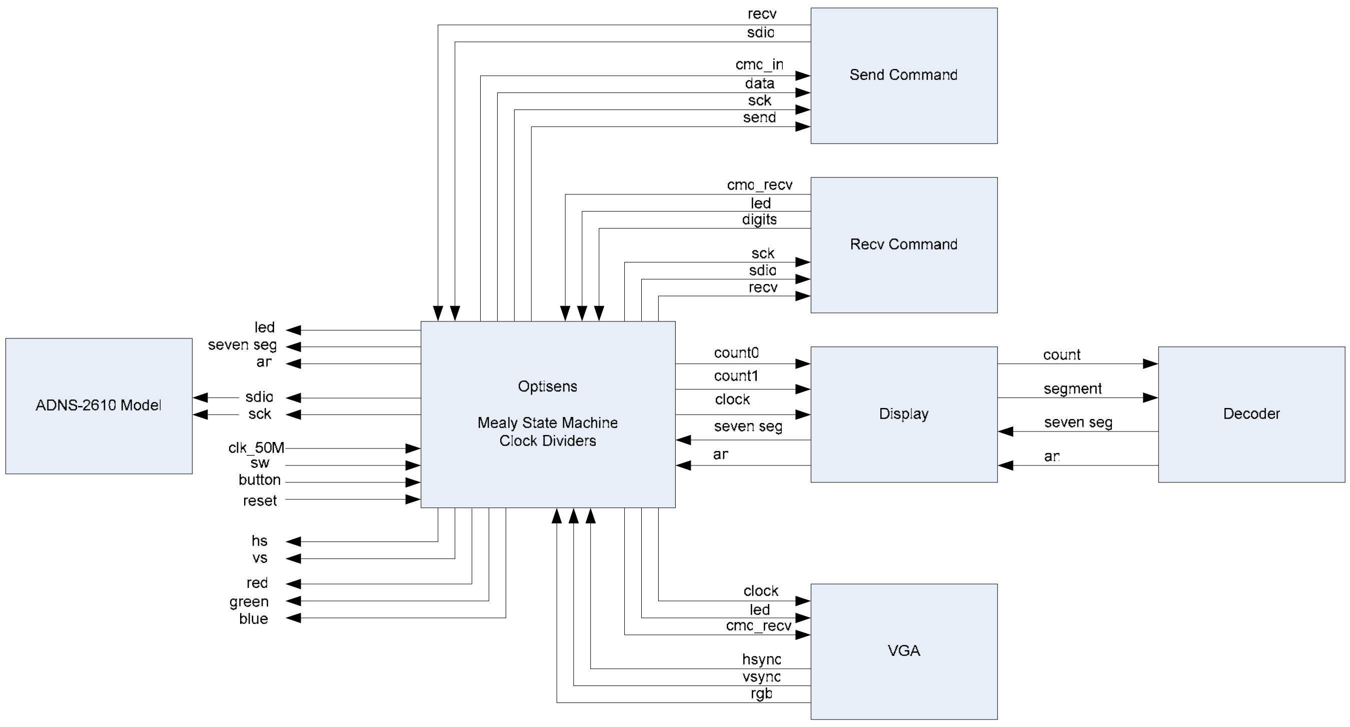


Fig. 79 - Block Diagram of the System

On the next page is a functional description of my state machine that I am using in the optisens module.

A.2 - State Diagram

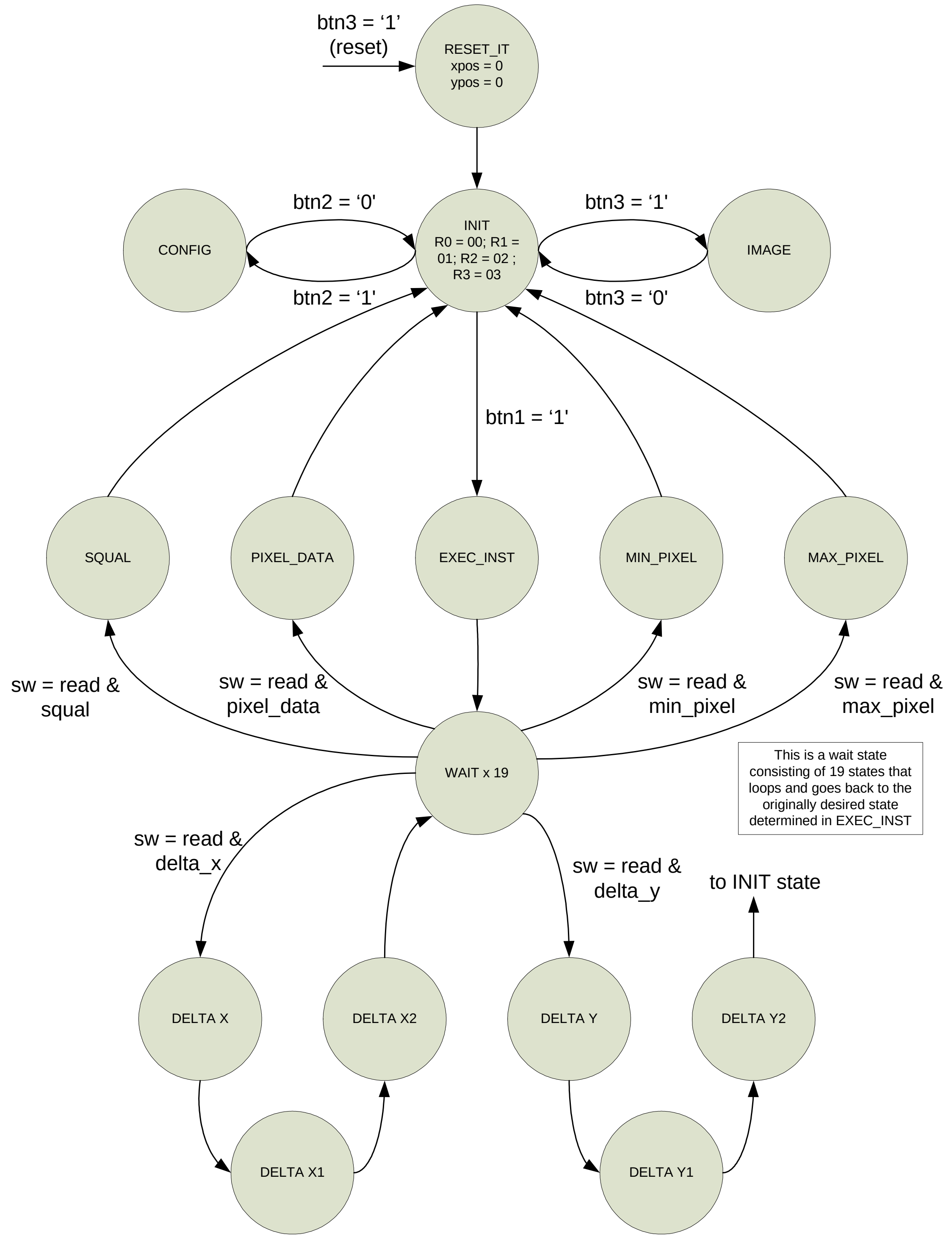


Fig. 80 - State diagram of optisens machine

I have designed this project with state machines. I like state machines very much, since I think that they give a much better overview of what is going on.

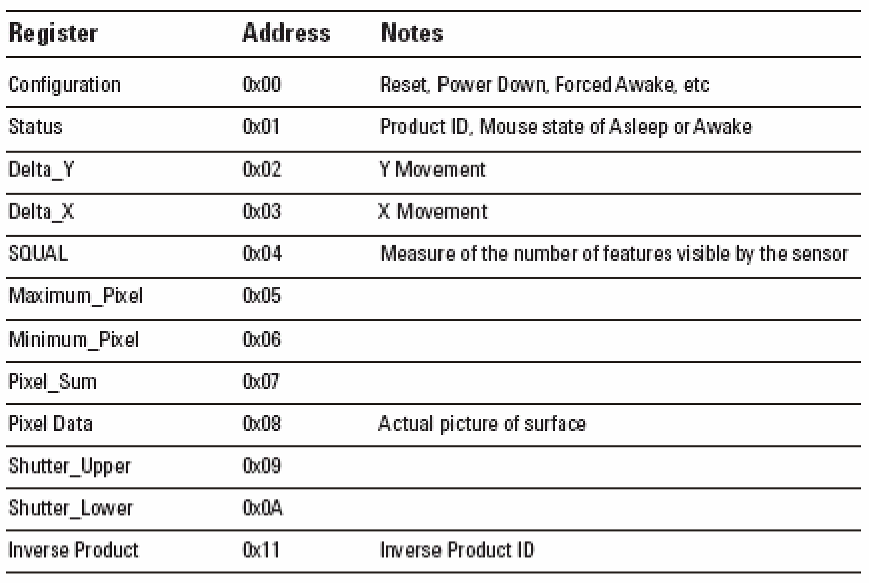
The state machine works in the following way. After reset a RESET_IT state is entered. The idea with this state is to reset the counters for the X and Y position. Then INIT is entered, which is the default state to be in. If button 2 or 3 is pressed either the config or image state is entered. In these two states data is written to either the config or image register in the optical sensor. The config register has to be written to in order to reset the sensor and to turn on the LED attached to the sensor to illuminate the surface with full brightness constantly. It is necessary to write to the image register in order to capture an image with the sensor at a given moment. Otherwise for other commands (that is read commands) the EXEC_INST is entered. The command is read from the dip switches and send to the optical sensor. We then enter a wait state that consists of a number of wait states that we cycle through before returning to the command that we want to read the response to. This is done in the states like SQUAL, PIXEL_DATA, MIN_PIXEL etc.. Finally we end up in INIT again. The states for DELTA_X and DELTA_Y are a little special since they are read consecutively when DELTA_X is executed. This is done in order to be able to show both the X and Y position simultaneously on the seven-segment display.

A.3 - Special Features

I have added most of the commands for the CCD-array since I felt that it was nice to have more commands implemented. Furthermore it was not a lot of effort to make this addition since the procedure for sending commands and receiving the response is very similar. The commands supported are :

- Configuration
- X-position
- Y-position
- SQUAL, a measure of the features visible by the sensor
- Maximum pixel value
- Minimum pixel value
- Pixel data (outputs the values on the VGA display)

The values for the commands follows the commands found in the ADNS-2610 datasheet (optical sensor datasheet). These are shown in Fig. 81



Register	Address	Notes
Configuration	0x00	Reset, Power Down, Forced Awake, etc
Status	0x01	Product ID, Mouse state of Asleep or Awake
Delta_Y	0x02	Y Movement
Delta_X	0x03	X Movement
SQUAL	0x04	Measure of the number of features visible by the sensor
Maximum_Pixel	0x05	
Minimum_Pixel	0x06	
Pixel_Sum	0x07	
Pixel Data	0x08	Actual picture of surface
Shutter_Upper	0x09	
Shutter_Lower	0x0A	
Inverse Product	0x11	Inverse Product ID

Fig. 81 - Full command set for the optical sensor

A.4 - Testing in simulator (Modelsim)

A.4.1 - Testbench for VGA module

In appendix page 43 the testbench for the VGA module can be seen. On page 61 and 62 a wave form for the simulation can be seen. I have divided the waveforms up into two different pages since the vertical sync has a much lower frequency than the horizontal sync, so having a whole period of the vertical sync would therefore make it impossible to see all the details on the waveform. On page 61 a waveform of the beginning of the vertical sync can be seen. Note that the data for RGB is undefined to begin with. This is because the array that contains all the pixel values has to filled up before it can be output correctly. This is done using cmd_recv that clocks the data displayed on the LEDs in, in this case 8Fh and 4Fh. The first pixel has to have an '1' as MSB since this indicates that the pixel data from the optical sensor is a start of the frame, therefore the change from 8Fh to 4Fh. I have also demonstrated what the timing diagrams can be used to by calculating the periods for the hs and vs to see if it corresponded with expected values, which it did. Another and better approach would have been to output the timings to the simulator which I have not done here.

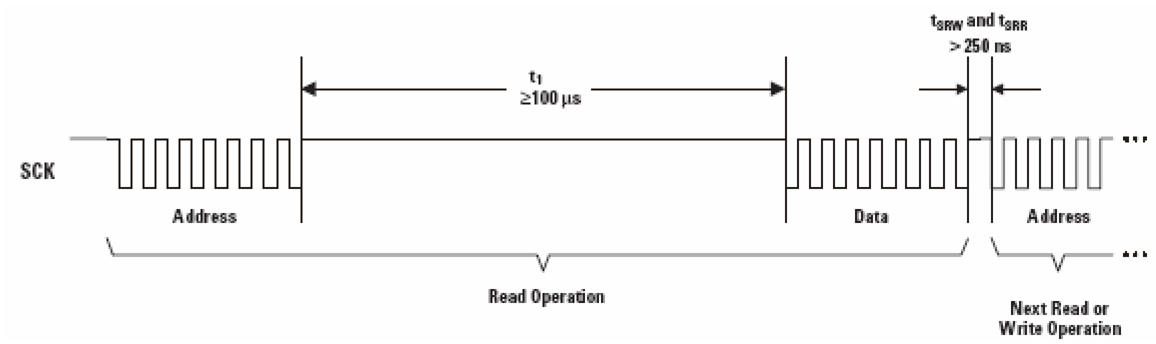
A.4.2 - Agilent ADNS-2610 model

Before I started to work with the serial commands I decided to make a simple model of the ADNS-2610 chip. The reason for this is that the timing when communicating with the CCD-array is very critical and by using my model it was possible for me to see if the bit patterns that I was sending and receiving looked right. The model is shown in appendix page 40.

A.4.3 - Testbench for the whole project

In this appendix on page 122 a waveform for the whole optisens project can be seen. Here I have simulated the acquisition of the change in x position and y position. I have designed my system so that each time an xpos is acquired the ypos will be acquired later and output on the seven segment display (xpos at pos.3-2 and ypos at pos. 0-1).

In the waveform it can be seen how a command is set by the switches. In this case the command is delta_x, corresponding to 03h. After that is done button 1 is pressed = '1' and the state machine therefore goes into the exec_inst state. Send is set to high, to signal that we want the send module to send the command to the optical sensor. This makes the send module start sending the command and at the same time the clock is changed from being constantly '1' to oscillate. After this the output on SDIO goes into high-Z and the state machine then goes into a wait state. According to the datasheet for the ADNS-2610 see Fig. 82, we have to wait at least 100 us before reading the result of our command.



t_{SRW} and t_{SRR} > 250 ns
t_1 ≥100 µs
SCK
Address
Data
Address
Read Operation
Next Read or Write Operation

Fig. 82 - Timing between write and read to the ADNS-2610 and subsequent commands

On the waveform in the appendix it can be seen that this is the case since we are waiting more than 500 us. In order to speed up the acquisition this could be changed (but it is fast enough as it is). I have then defined my model so it responds to the delta_x command with AAh. This response is received when the recv flag is set to high in the DELTA_X state, the clock starts oscillating and the read module reads in the response from my model. As it can be seen this is output correctly to the LEDs as AAh.

A.5 - Testing in real life

In order to test if the system was working I decided to view a piece of white PVC tape with some black lines as shown in Fig. 83

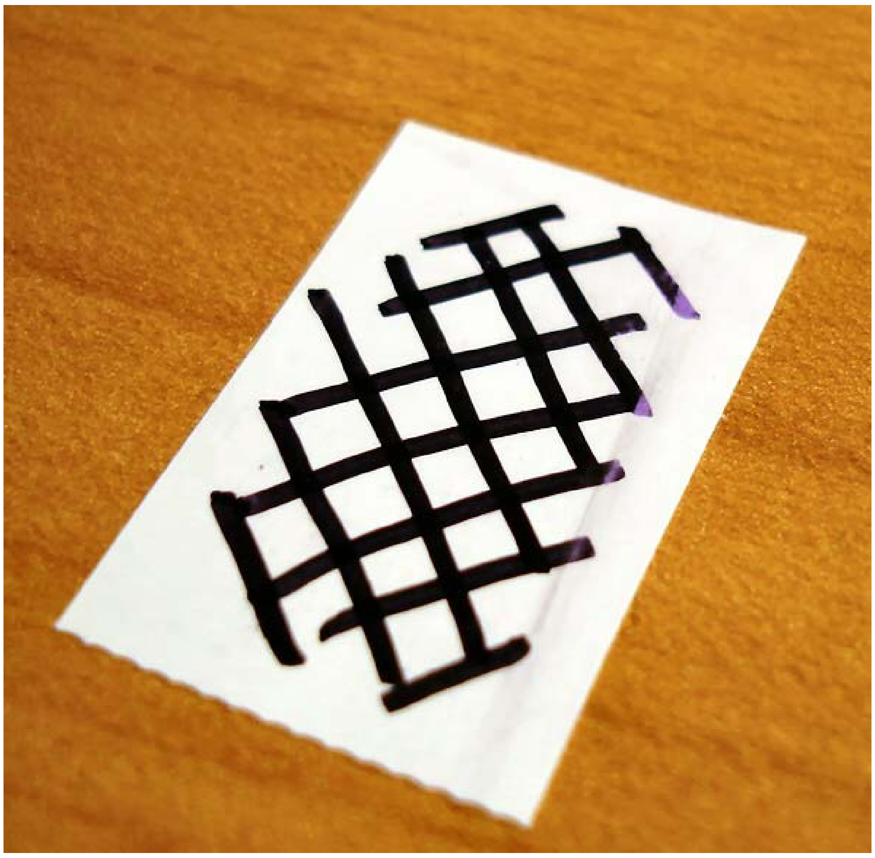


Fig. 83 - PVC tape with black lines used for testing

To begin with I choose to use an 3-bit color scale giving a total combination of 8 colors. This color scale comes from the three bits, Red, Green and Blue that you can either turn on or off on the Spartan board.

Using these 3-bits I would get the color scale shown in Fig. 84

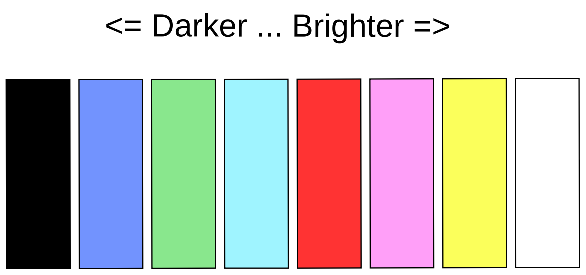


Fig. 84 - 3-bit Color scale

On the VGA-screen this would look as shown in Fig. 85. What you can see here is the border between a black line and a white area. The line in the left side of the screen is just a repetition of the right border because the 18 x 18 pixels do not have the same dimension as the VGA screen.

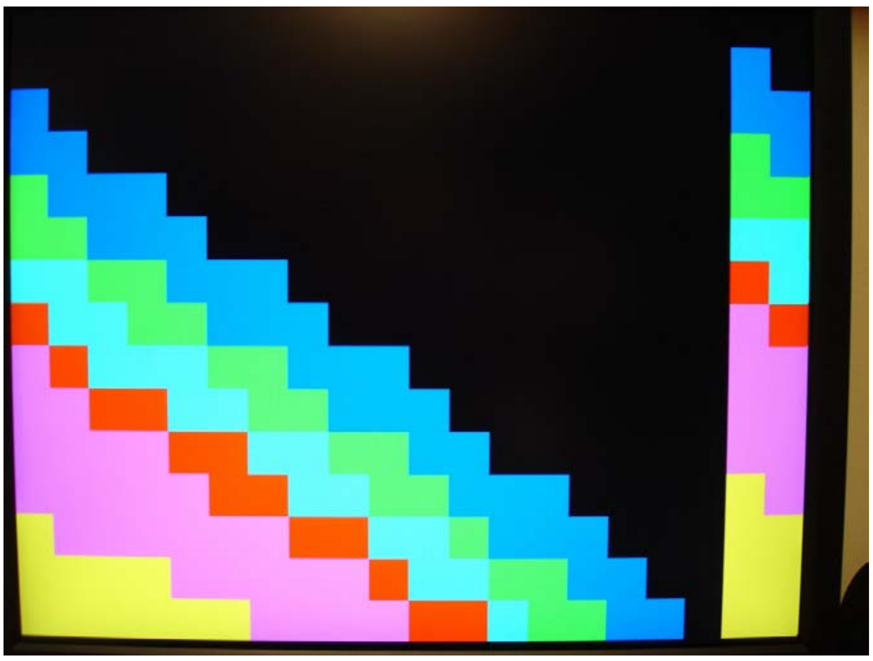


Fig. 85 - VGA image using an 3-bit color scale giving a total of 8 different colors

I find this color scale useful, but it is not very easy to get an understanding of what parts of the image that are darker than the others unless you look at the color scale shown in Fig. 84.

Now that I knew that it was possible to extract the image from the CCD-array and display it on the VGA screen I therefore decided to try and see if I could get grayscales from the Spartan board. One way to get grayscales is by using 6 bits with a resistor ladder. Turning on or off certain bits makes it possible to vary an analog voltage and thereby change the greyscale. This turned out to be successful and the result can be seen in Fig. 86.

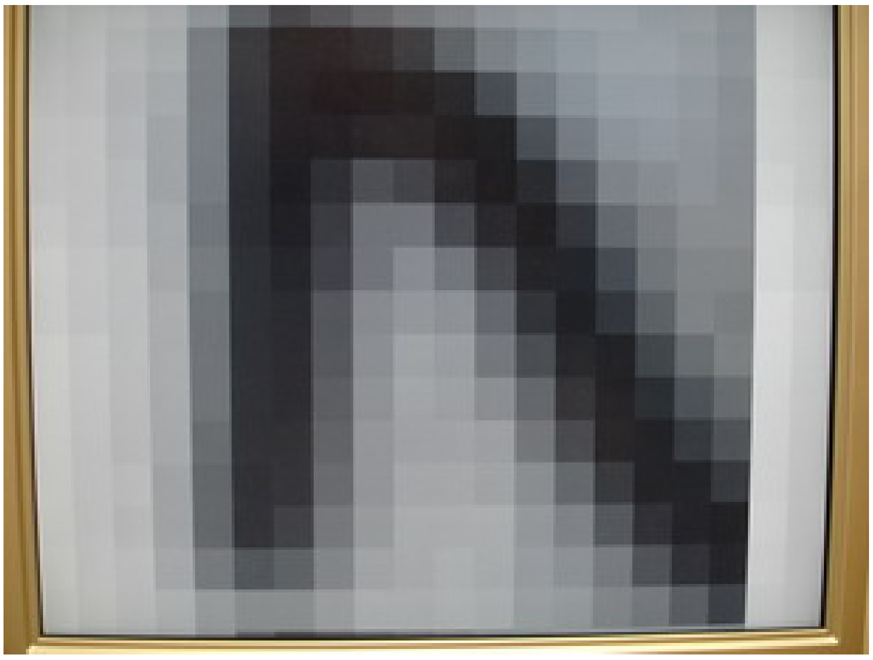


Fig. 86 - Greyscale image from CCD-array viewer

What can be seen in Fig. 86 is a part of the number 2 in the USAF 1951 chart [13] .

A.8 - Conclusion for the CCD-array image viewer

I think that this project is a good example of what a microprocessor is good for and what an FPGA is useful for. The communication with the optical sensor is very critical in terms of timing and involves sequential operations. This is implemented in my FPGA using my state machine and signaling with some flags when to do certain things. I think this makes my code a little clumsy compared to how I could have done the same thing in a microcontroller.
On the other hand I find that the VGA module is a good example of something the FPGA is much better at that than what could be done in a microcontroller. I like the fact that I could just build this module and I did not have to care about anything like processor usage, interrupts and so on, since everything would happen in parallel with the rest of my code. So as a conclusion I think that the best way to solve this task is by using an FPGA and then implement a microcontroller into it. In this way one would get the best of both worlds. This would give you the best cost/price and development time.

Besides that I think that it was a very interesting project to work with. I think I learned a lot from this too. My goal was to get a tool that made it possible to see whether the optical sensor was in focus or not and I have that now.

As further improvements I would like to improve the optisens project by dividing the VGA screen into smaller tiles, so consecutive images could be shown in for example a 13 x 13 pixel grid giving a total of 169 images. This would be a very nice feature to have too since a comparison between different images would be easy. A serial interface would also be nice so data could be stored and used in for example Matlab.

But there are many things that can be made as improvements and only your imagination sets the limit. I am stopping here since my primary goals have been achieved.

A.10 - VGA timing information

```
"VGA industry standard" 640x480 pixel mode

General characteristics

Clock frequency 25.175 MHz
Line  frequency 31469 Hz
Field frequency 59.94 Hz

One line

  8 pixels front porch
 96 pixels horizontal sync
 40 pixels back porch
  8 pixels left border
640 pixels video
  8 pixels right border
---
800 pixels total per line

One field

  2 lines front porch
  2 lines vertical sync
 25 lines back porch
  8 lines top border
480 lines video
  8 lines bottom border
---
525 lines total per field

Other details

Sync polarity: H negative, V negative
Scan type: non interlaced.
Information source
Article "Re: VGA specifications ,where ?" posted 19 Nov 1997 to sci.electronics.design
newsgroup by Jeroen Stessen
```

A.11 - Code

A.11.1 - optisens.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.optisens_package.ALL;


entity optisens is

  Port (
      -- Clock
      clk_50M       : in    std_logic;                              -- clock from board

      -- User inputs
      sw            : in    std_logic_vector(7 downto 0);           -- switches
      btn           : in    std_logic_vector(2 downto 0);           -- buttons
      reset         : in    std_logic;                              -- reset

      -- For the seven segment display and LEDs
      seven_seg     : out   std_logic_vector(7 downto 0) := x"00";  -- pattern for seven
                                                                       segment
      an            : out   std_logic_vector(3 downto 0) := x"1";   -- anodes for LEDs
      led           : out   std_logic_vector(7 downto 0);           -- LEDs

      -- For the optical mouse
      sdio          : inout std_logic :='Z';                        -- serial line
      sck           : out   std_logic := '0';                       -- serial clock

      -- For the VGA connection
      hs            : out   std_logic;                            -- horizontal sync
      vs            : out   std_logic;                            -- vertical sync
      red           : out   std_logic;                            -- RGB value for red
      green         : out   std_logic;                            -- RGB value for green
      blue          : out   std_logic);                           -- RGB value for blue

end optisens;

architecture Behavioral of optisens is

  component display is
    Port ( count0      : in std_logic_vector(7 downto 0);
           count1      : in std_logic_vector(7 downto 0);
           clock       : in std_logic;
           seven_seg   : out std_logic_vector(7 downto 0);
           an          : out std_logic_vector(3 downto 0));
  end component;

  component send_command is
    Port ( sck         : in    std_logic;       -- clock
           send        : in    std_logic;       -- flag to start transmission
           cmd_in      : in    std_logic_vector(7 downto 0);  -- command that we want to
                                                                 send
           data        : in    std_logic_vector(7 downto 0);  -- data that we want to
                                                                 send
           sdio        : out   std_logic;                     -- data line to the optical
                                                                 sensor
           send_en_clk : out   std_logic);

  end component;

  component recv_command is
    Port ( sck   : in    std_logic;             -- clock
           sdio  : in    std_logic;             -- data line that goes to optical sensor
           recv  : in    std_logic;             -- flag to indicate that receiver should
                                                     start to receive
           led   : out   std_logic_vector(7 downto 0);-- LEDs to show result of a cmd
           digits: out   std_logic_vector(7 downto 0);-- digits to show result
                                                                 of a cmd
           cmd_recv    : out   std_logic;
           recv_en_clk : out   std_logic);
  end component;

  component vga is
    Port ( clock       : in    std_logic;                     -- clock
           led         : in    std_logic_vector(7 downto 0);  -- reads command input from
                                                                 LEDs
--         data_rgb    : out   std_logic_vector(5 downto 0);  -- data for future 6-bit
                                                                 grey-scale
           cmd_recv    : in    std_logic;                     -- indicator when a command
                                                                 is received
           hsync       : out   std_logic;                     -- horizontal sync
           vsync       : out   std_logic;                     -- vertical sync
           rgb         : out   std_logic_vector(2 downto 0)); -- RGB values
  end component;

  --Define all the states we have in our design so the state_type type can be defined
  type state_type is (RESET_IT, INIT, CONFIG, IMAGE, EXEC_INST, SQUAL, MAX_PIXEL,
MIN_PIXEL, DELTA_X, DELTA_X1, DELTA_X2, DELTA_Y, DELTA_Y1, DELTA_Y2, DELTA_Y3, WAIT_0,
WAIT_1, WAIT_2, WAIT_3, WAIT_4, WAIT_5, WAIT_6, WAIT_7, WAIT_8, WAIT_9, WAIT_10, WAIT_11,
WAIT_12, WAIT_13, WAIT_14, WAIT_15, WAIT_16, WAIT_17, PIXEL_DATA, PIXEL_DATA_1,
PIXEL_DATA_2, PIXEL_DATA_3);
  type reg_type is array(0 to 3) of std_logic_vector(7 downto 0);

  --Define registers
  signal digits        : std_logic_vector(7 downto 0);
  signal digits_0      : std_logic_vector(7 downto 0);
  signal digits_1      : std_logic_vector(7 downto 0);

  signal data          : std_logic_vector(7 downto 0);

  signal cmd_in        : std_logic_vector(7 downto 0) := x"00";
  signal led_sig       : std_logic_vector(7 downto 0) := x"00";
  signal rgb_sig       : std_logic_vector(2 downto 0) := "000";

  --Define state signals
  signal state         : state_type;
  signal next_state    : state_type;
  signal next_state_2  : state_type;

  --Define clock signals
  signal clk_state     : std_logic := '0';
  signal clk_disp      : std_logic := '0';
  signal clk_optchip   : std_logic := '0';
  signal clk_vga       : std_logic := '0';
  signal send          : std_logic := '0';
  signal recv          : std_logic := '0';
  signal cmd_recv      : std_logic := '0';
  signal send_en_clk   : std_logic := '0';
  signal recv_en_clk   : std_logic := '0';
--  signal data_rgb      : std_logic_vector(5 downto 0) := "000000";

  begin

    --Make a copy of all components used in the design
    display_copy      : display port map (count0 => digits_0, count1 => digits_1, clock
=> clk_disp, seven_seg => seven_seg, an => an);
    send_command_copy : send_command port map (cmd_in => cmd_in, sck => clk_optchip, send
=> send, data => data, sdio => sdio, send_en_clk => send_en_clk);
    recv_command_copy : recv_command port map (sck => clk_optchip, sdio => sdio, recv =>
recv, led => led_sig, digits => digits, cmd_recv => cmd_recv, recv_en_clk =>
recv_en_clk);
    vga_copy          : vga port map (clock => clk_vga, cmd_recv => cmd_recv, hsync =>
hs, led => led_sig, vsync => vs, rgb => rgb_sig);
--    vga_copy          : vga port map (clock => clk_vga, cmd_recv => cmd_recv, hsync =>
hs, led => led_sig, data_rgb => data_rgb, vsync => vs, rgb => rgb_sig);

    sck <= clk_optchip OR (NOT (send_en_clk OR recv_en_clk));
    led <= led_sig;
--    sw_sig <= sw;

    red   <= rgb_sig(2);
    green <= rgb_sig(1);
    blue  <= rgb_sig(0);

    --generate other clocks from the 50 MHz oscillator
```

```
clock_process: process (clk_50M, reset)

  variable counter_50M     : integer range 0 to 12_500_000;
  variable counter_disp    : integer range 0 to 1_000;
  variable counter_optchip : integer range 0 to 250 := 0;
  variable counter_vga     : integer range 0 to 2 := 0;
  variable temp            : std_logic_vector(5 downto 0) := "000000";

  begin

    if reset = '1' then
      counter_50M     := 0;
      counter_disp    := 0;
      counter_optchip := 0;
      counter_vga     := 0;
    elsif clk_50M'event and clk_50M = '1' then
      counter_50M     := counter_50M + 1;
      counter_disp    := counter_disp + 1;
      counter_optchip := counter_optchip + 1;
      counter_vga     := counter_vga + 1;

      --clock used for the state machine
      if counter_50M >= 400 then           --for the simulation
        clk_state <= NOT clk_state;
        counter_50M := 0;
      end if;

      if counter_optchip >= 25 then           --for the simulation
        clk_optchip <= NOT clk_optchip;
        counter_optchip := 0;
      end if;

      --clock used for the display module
      if counter_disp = 1_000 then
        clk_disp <= NOT clk_disp;
        counter_disp := 0;
      end if;

      --clock used for the vga module
      if counter_vga = 1 then
        clk_vga <= NOT clk_vga;
        counter_vga := 0;
      end if;

    end if;

  end process;

--process that syncronizes the state machine with the clock
sync_process: process (clk_state, reset)
  begin
    if (clk_state'event and clk_state = '1') then
      if reset = '1' then
        state <= RESET_IT;
      else
        state <= next_state;
      end if;
    end if;
end process;


--MEALY State Machine - Outputs based on state and inputs
--state_process : process (state, digits_count)
state_process : process (clk_state)

variable xpos : std_logic_vector(7 downto 0) := x"00";
variable ypos : std_logic_vector(7 downto 0) := x"00";

begin
  --When there is a clk event the state machine changes since
  --we have memory ~ flip-flops
  if clk_state'event and clk_state = '1' then

    case (state) is
      --Initial state that we enter. Initialize registers used.
```

```
when RESET_IT =>
  xpos := x"00";          -- reset X position
  ypos := x"00";          -- reset Y position

when INIT =>
  recv <= '0';
  send <= '0';

when CONFIG =>
  cmd_in <= x"80";  --(write & config);
  data <= sw;
  recv <= '0';
  send <= '1';

when IMAGE =>
  cmd_in   <= x"88";--(write & pixel_data);
  data     <= sw;
  recv     <= '0';
  send     <= '1';

when EXEC_INST =>
  cmd_in <= sw;
  recv <= '0';
  send <= '1';

when PIXEL_DATA =>  -- to read the data from the CCD-array
  recv <= '1';
  send <= '0';

when DELTA_X =>     -- read the change in delta x position and add it to xpos
  recv <= '1';
  send <= '0';

when DELTA_X1 =>
  recv <= '0';
  send <= '0';

when DELTA_X2 =>
  xpos := xpos + digits;
  digits_1 <= xpos;

  cmd_in <= x"02";
  recv <= '0';
  send <= '1';

when DELTA_Y =>     -- read the change in delta y position and add it to ypos
  recv <= '1';
  send <= '0';

when DELTA_Y1 =>
  recv <= '0';
  send <= '0';

when DELTA_Y2 =>
  ypos := ypos + digits;
  digits_0 <= ypos;

  recv <= '0';
  send <= '0';

when SQUAL =>       -- read the number of features visible by the sensor
  digits_0 <= digits;
  digits_1 <= x"00";
  recv <= '1';
  send <= '0';

when MAX_PIXEL =>   -- read the value of the maximum pixel in the image
  digits_1 <= digits;
  recv <= '1';
  send <= '0';

when MIN_PIXEL =>   -- read the value of the minimum pixel in the image
  digits_0 <= digits;
  recv <= '1';
  send <= '0';
```

```
      when others =>

    end case;
  end if;
end process;


next_state_process : process (state, btn, clk_state)
--clock needed since memory is in next_state_2
begin

  if clk_state'EVENT and clk_state = '1' then

  next_state <= state;  --default is to stay in current state
  next_state_2 <= next_state_2;

    case (state) is

      -- Initial state
      when RESET_IT    => next_state <= INIT;

      when INIT        =>
      -- Depending on what button is pressed move into next state
        if btn = "001" then
          next_state <= EXEC_INST;
        elsif btn = "010" then
          next_state <= CONFIG;
        elsif btn = "100" then
          next_state <= IMAGE;
        else
          next_state <= INIT;
        end if;

      when CONFIG      =>
        next_state <= INIT;

      when EXEC_INST =>  -- go into a wait state after a read command
                          is sent and come back into next_state_2
        next_state <= WAIT_0;

        case sw is
          when x"02"  => next_state_2 <= DELTA_Y;
          when x"03"  => next_state_2 <= DELTA_X;
          when x"04"  => next_state_2 <= SQUAL;
          when x"05"  => next_state_2 <= MAX_PIXEL;
          when x"06"  => next_state_2 <= MIN_PIXEL;
          when x"08"  => next_state_2 <= PIXEL_DATA;
          when others => next_state_2 <= INIT;
        end case;

      when PIXEL_DATA => next_state    <= WAIT_0;
                         next_state_2  <= INIT;

      when DELTA_X  => next_state   <= DELTA_X1;
      when DELTA_X1 => next_state   <= DELTA_X2;
      when DELTA_X2 => next_state   <= WAIT_0;
                       next_state_2 <= DELTA_Y;

      when DELTA_Y  => next_state   <= DELTA_Y1;
      when DELTA_Y1 => next_state   <= DELTA_Y2;
      when DELTA_Y2 => next_state   <= INIT;

      when IMAGE   => next_state    <= WAIT_14;
                      next_state_2  <= INIT;

      -- wait statements to give the required delay between command and response
      when WAIT_0   => next_state   <= WAIT_1;
      when WAIT_1   => next_state   <= WAIT_2;
      when WAIT_2   => next_state   <= WAIT_3;
      when WAIT_3   => next_state   <= WAIT_4;
      when WAIT_4   => next_state   <= WAIT_5;
      when WAIT_5   => next_state   <= WAIT_6;
      when WAIT_6   => next_state   <= WAIT_7;
      when WAIT_7   => next_state   <= WAIT_8;
```

```
          when WAIT_8   => next_state   <= WAIT_9;
          when WAIT_9   => next_state   <= WAIT_10;
          when WAIT_10  => next_state   <= WAIT_11;
          when WAIT_11  => next_state   <= WAIT_12;
          when WAIT_12  => next_state   <= WAIT_13;
          when WAIT_13  => next_state   <= WAIT_14;
          when WAIT_14  => next_state   <= WAIT_15;
          when WAIT_15  => next_state   <= next_state_2;

          when others => next_state <= INIT;
        end case;
      end if;

    end process;


end Behavioral;
```

A.11.2 - VGA.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.optisens_package.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity vga is

    Port ( clock       : in    std_logic;
           led         : in    std_logic_vector(7 downto 0);
           cmd_recv    : in    std_logic;
--           data_rgb    : out   std_logic_vector(5 downto 0);
           hsync       : out   std_logic;
           vsync       : out   std_logic;
           rgb         : out   std_logic_vector(2 downto 0));

end vga;

architecture Behavioral of vga is

begin

process(clock)

variable hs_val : std_logic := '0';
variable vs_val : std_logic := '0';
variable index  : std_logic := '0';
variable i      : integer range 0 to 800 := 0;
variable j      : integer range 0 to 415_200 := 0;

variable k      : integer range 0 to 323 := 0;
variable m      : integer range 0 to 323 := 0;
--variable r      : integer range 0 to 948 := 0;

variable x      : integer range 0 to 639 := 0;
variable y      : integer range 0 to 479 := 0;
variable x_s    : integer range 0 to 20 := 0;
variable y_s    : integer range 0 to 20 := 0;

--type data_type is array (323 downto 0) of std_logic_vector(5 downto 0);
type data_type is array (323 downto 0) of std_logic_vector(2 downto 0);

variable data         : data_type;
variable old_cmd_recv : std_logic := '0';
variable value        : integer range 0 to 63 := 0;

begin
```

```
if clock'EVENT and clock = '1' then

  --For storing the data in memory
  --if a new response to a command has been received
  if cmd_recv = '1' and old_cmd_recv = '0' then
    old_cmd_recv := '1';

    --reset if start of a frame from pixel data
    if led(7) = '1' then
      k := 0;
    end if;

    --load data into buffer
    if led(6) = '1' then   -- if data is valid
      data(k) := led(5 downto 3);
      --data(k) := led(5 downto 0);
    end if;

    --taking care of the indexing variable
    if k = 323 then
      k := 0;
    else
      k := k + 1;
    end if;
  else
    old_cmd_recv := cmd_recv;
  end if;

  i := i + 1;

  --For the horizontal sync
  if hs_val = '1' then
    if i = 48 then          -- end of back porch, start of Tdisp
      index := '1';
      m := 0;
      x := 0;
    elsif i = 688 then      -- end of Tdisp, start of front porch
      index := '0';
    elsif i = 704 then      -- end of Ts, sync pulse time
      hs_val := '0';        -- hs changes value from 1 to 0
      i := 0;               -- and counter i is reset
    end if;
  else
    if i = 96 then          -- end of Tpw, pulse width time
      hs_val := '1';        -- hs changes value from 0 to 1
      i := 0;               -- and counter i is reset
    end if;
  end if;

  --For the vertical sync
  j := j + 1;

  if vs_val = '1' then
    if j = 23_200 then      -- end of back porch, start of Tdisp
      y := 0;
      index := '1';

      if value = 63 then
        value := 0;
      else
        value := value + 1;
      end if;

    elsif j = 407_200 then  -- end of Tdisp, start of front porch
      index := '0';
    elsif j = 415_200 then  -- end of Ts, sync pulse time
      vs_val := '0';        -- hs changes value from 1 to 0
      j := 0;               -- and counter j is reset
    end if;
  else
    if j = 1600 then        -- end of Tpw, pulse width time
      vs_val := '1';        -- hs changes value from 0 to 1
      j := 0;               -- and counter j is reset
    end if;
  end if;
```

```
    --For the stuff being displayed
    if index = '1' then

      --location in XY coordinates of position on the screen
      if x = 639 then
        x := 0;
        if y = 479 then
          y := 0;
        else
          y := y + 1;
        end if;
      else
        x := x + 1;
      end if;

      x_s := x/32;
      y_s := y/32;

      m := x_s + y_s*18;

      rgb <= data(m);
      --data_rgb <= data(m);
    else
      rgb <= "000";
    end if;


  end if; --clock'EVENT

hsync <= hs_val;
vsync <= vs_val;

end process;

end Behavioral;
```

A.11.3 - Send_command.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity send_command is

  Port ( cmd_in      : in    std_logic_vector(7 downto 0); -- Command that we want to
send
         data        : in    std_logic_vector(7 downto 0); -- Data that we want to send
         send        : in    std_logic;                    -- Flag to indicate that we
want to send

         sck         : in    std_logic;                    -- Serial clock that is used
with the optical sensor
         sdio        : out   std_logic := 'Z';             -- The serial port to the
optical sensor
         send_en_clk : out   std_logic := '0');

end send_command;

architecture Behavioral of send_command is


begin

process(sck)

variable i : integer range 0 to 16 := 0;
variable send2 : std_logic := '0';
variable oldsend : std_logic := '0';

begin
```

```
if sck'EVENT and sck = '0' then --write on falling edge

  -- Check if the state of send has changed since last time
  if oldsend = '0' and send = '1' then
    if send2 = '0' then
      i := 0;
      send2  := '1';
      oldsend := '1';
    end if;
  else oldsend := send;
  end if;

--  recv <= '0';

  if send2 = '1' then

    -- if this is a read command
    if cmd_in(7) = '0' then
      if i < 8 then
        sdio <= cmd_in(7-i);
        i := i + 1;
        send_en_clk <= '1';
      else
        sdio <= 'Z';
--        recv <= '1';
        send_en_clk <= '0';
        send2 := '0';
      end if;

    -- else if this is a write command
    elsif cmd_in(7) = '1' then

      -- write the address
      if i < 8 then
        sdio <= cmd_in(7-i);
        i := i + 1;
        send_en_clk <= '1';
      -- and write the data
      elsif i < 16 then
        sdio <= data(15-i);
        i := i + 1;
--        send_en_clk <= '1';
      else
      -- when done set the IO to high-Z, indicate that we are ready to receive a command
      -- reset send2 and disable the clock to the optical sensor
        sdio <= 'Z';
        send_en_clk <= '0';
--        recv <= '1';
        send2 := '0';
      end if;
    end if;
  end if;

end if;

end process;

end Behavioral;
```

A.11.4 - Recv_command.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity recv_command is

  Port ( sck         : in    std_logic;
         sdio        : in    std_logic;
         recv        : in    std_logic;
         led         : out   std_logic_vector(7 downto 0);
         digits      : out   std_logic_vector(7 downto 0);
         cmd_recv    : out   std_logic;
         recv_en_clk : out   std_logic := '0');

end recv_command;

architecture Behavioral of recv_command is

begin

process(sck)

variable i : integer range 0 to 8 := 0;
variable response : std_logic_vector(7 downto 0) := x"00";
variable recv2 : std_logic := '0';
variable oldrecv : std_logic := '0';

begin

if sck'EVENT and sck = '1' then --read on rising edge

  if oldrecv = '0' and recv = '1' then
    if recv2 = '0' then
      i           := 0;
      recv2       := '1';
      oldrecv     := '1';
      recv_en_clk <= '1';
    end if;
  else oldrecv := recv;
  end if;

  response(8-i) := sdio;

  if recv2 = '1' then
    if i >= 8 then
      i := 0;
      recv_en_clk <= '0';
      led <= response(7 downto 0);
      digits <= response(7 downto 0);
      cmd_recv <= '1';
      recv2 := '0';
    else
      i := i + 1;
      cmd_recv <= '0';
    end if;
  end if;

end if;

end process;


end Behavioral;
```

A.11.5 - Display.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


--Display decoder used to display digits on the four seven-segment
--displays.
entity display is

  Port ( count0    : in std_logic_vector(7 downto 0);
         count1    : in std_logic_vector(7 downto 0);
         clock     : in std_logic;
         seven_seg : out std_logic_vector(7 downto 0);
         an        : out std_logic_vector(3 downto 0));

end display;

architecture Behavioral of display is
```

```
  --Decode a number into the value displayed on the port to the seven segment display
      component decoder is
        Port ( count     : in std_logic_vector(3 downto 0);
           segment   : in integer range 0 to 3;
           seven_seg : out std_logic_vector(7 downto 0);
           an        : out std_logic_vector(3 downto 0));
  end component;

      signal count   : std_logic_vector(3 downto 0);
      signal segment : integer range 0 to 3;

  begin

    --Decode a number into the value displayed on the port to the seven segment display
    decoder1: decoder port map (count => count, segment => segment, seven_seg =>
seven_seg, an => an);

    count_process: process(count0, count1, clock)

    variable count_0 : std_logic_vector(3 downto 0);
    variable count_1 : std_logic_vector(3 downto 0);
    variable count_2 : std_logic_vector(3 downto 0);
    variable count_3 : std_logic_vector(3 downto 0);
    variable seg     : integer range 0 to 3 := 0;

    begin

      --As default, zero on all digits
      count_0 := count0(3 downto 0);
      count_1 := count0(7 downto 4);
      count_2 := count1(3 downto 0);
      count_3 := count1(7 downto 4);


      --Multiplex the display
      if clock'EVENT and clock = '1' then
        if seg = 3 then seg := 0;
        else seg := seg + 1;
        end if;
      end if;

      case seg is
        when 0 => count <= count_0;
        when 1 => count <= count_1;
        when 2 => count <= count_2;
        when 3 => count <= count_3;
      end case;

      segment <= seg;

   end process;


end Behavioral;
```

A.11.6 - Decoder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.optisens_package.ALL;

--Decoder than decodes a number into the value needed on the port to the
--seven segment display
entity decoder is
  Port ( count     : in std_logic_vector(3 downto 0);
         segment   : in integer range 0 to 3;
         seven_seg : out std_logic_vector(7 downto 0);
                 an        : out std_logic_vector(3 downto 0));
end decoder;

architecture Behavioral of decoder is
```

```
begin
      -- display the count value on the seven segments
  seven_segment_decoder_process: process(count, segment)

  begin
            case count is
                  when "0000" => seven_seg <= zero;
                  when "0001" => seven_seg <= one;
                  when "0010" => seven_seg <= two;
                  when "0011" => seven_seg <= three;
                  when "0100" => seven_seg <= four;
                  when "0101" => seven_seg <= five;
                  when "0110" => seven_seg <= six;
                  when "0111" => seven_seg <= seven;
                  when "1000" => seven_seg <= eight;
                  when "1001" => seven_seg <= nine;
                  when "1010" => seven_seg <= ten;
                  when "1011" => seven_seg <= eleven;
                  when "1100" => seven_seg <= twelve;
                  when "1101" => seven_seg <= thirteen;
                  when "1110" => seven_seg <= fourteen;
                  when "1111" => seven_seg <= fifteen;
      when others => seven_seg <= zero;
            end case;

    case segment is
      when 0 => an <= digit0;
      when 1 => an <= digit1;
      when 2 => an <= digit2;
      when 3 => an <= digit3;
    end case;

  end process seven_segment_decoder_process;

end Behavioral;
```

A.11.7 - ADNS_2610_model.vhd

```
entity agilent_ADNS_2610_model is
      PORT(
            sdio : INOUT std_logic := 'Z';
            sck  : IN    std_logic);
end agilent_ADNS_2610_model;

architecture Behavioral of agilent_ADNS_2610_model is

signal command : std_logic_vector(7 downto 0);

begin


process(sck)

variable i : integer range 0 to 9 := 0;
variable j : integer range 0 to 8;
variable cmd_rcv : std_logic := '0';
variable command2 : std_logic_vector(7 downto 0);

variable reg_delta_x : std_logic_vector(7 downto 0) := x"AA";
variable reg_delta_y : std_logic_vector(7 downto 0) := x"AA";

variable send : std_logic := '0';

begin

  if sck'EVENT and sck = '1' then      --read on rising edge

    if send = '0' then
      command2 := command2(6 downto 0) & command2(7);    -- shift the command in
      command2(0) := sdio;
    end if;

  elsif sck'EVENT and sck = '0' then   --write on falling edge
```

```
    if command2 = read & delta_x then  -- check to see which command it was
      command <= command2;
      command2 := x"00";
      j := 0;
      send := '1';
    end if;

    if command = read & delta_x then

      sdio <= reg_delta_y(7-j);
      j := j + 1;

      if j = 8 then
        command <= x"00";
        send := '0';
      end if;

    else
      sdio <= 'Z';
    end if;

  end if;

end process;

end Behavioral;
```

A.11.8 - optisens_tb.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
use work.optisens_package.ALL;

ENTITY optisens_optisens_tb_vhd_tb IS
END optisens_optisens_tb_vhd_tb;

ARCHITECTURE behavior OF optisens_optisens_tb_vhd_tb IS

      COMPONENT optisens
      PORT(
            clk_50M : IN std_logic;
            sw : IN std_logic_vector(7 downto 0);
            btn : IN std_logic_vector(2 downto 0);
            reset : IN std_logic;
            sdio : INOUT std_logic;
            seven_seg : OUT std_logic_vector(7 downto 0);
            an : OUT std_logic_vector(3 downto 0);
            sck : OUT std_logic;
            led : OUT std_logic_vector(7 downto 0)
            );
      END COMPONENT;

      COMPONENT agilent_adns_2610_model
      PORT(
            sdio : INOUT std_logic;
            sck  : IN    std_logic);
      END COMPONENT;


      SIGNAL clk_50M :  std_logic := '0';
      SIGNAL sw :  std_logic_vector(7 downto 0);
      SIGNAL btn :  std_logic_vector(2 downto 0);
      SIGNAL reset :  std_logic;
      SIGNAL seven_seg :  std_logic_vector(7 downto 0);
      SIGNAL an :  std_logic_vector(3 downto 0);
      SIGNAL sdio :  std_logic := 'Z';
      SIGNAL sck :  std_logic;
      SIGNAL led :  std_logic_vector(7 downto 0);

BEGIN

      uut: optisens PORT MAP(
            clk_50M => clk_50M,
            sw => sw,
            btn => btn,
            reset => reset,
            seven_seg => seven_seg,
            an => an,
            sdio => sdio,
            sck => sck,
            led => led
      );

       model1: agilent_adns_2610_model port map (sdio=>sdio,sck=>sck);

-- *** Test Bench - User Defined Section ***
   sw       <= read & delta_x,
               write & pixel_data after 155 us,
               x"08" after 242 us;


   reset    <= '1', '0' after 1 us;

   clk_50M  <= not clk_50M after 10 ns; -- 50 MHz clock

   btn      <= "001" after 2 us,
               "000" after 68 us,
               "001" after 269 us,
               "000" after 282 us,
               "010" after 342 us,
               "000" after 348 us;


   tb : PROCESS
   BEGIN
      wait; -- will wait forever
   END PROCESS;
-- *** End Test Bench - User Defined Section ***

END;
```

A.11.9 - vga_tb.vhd

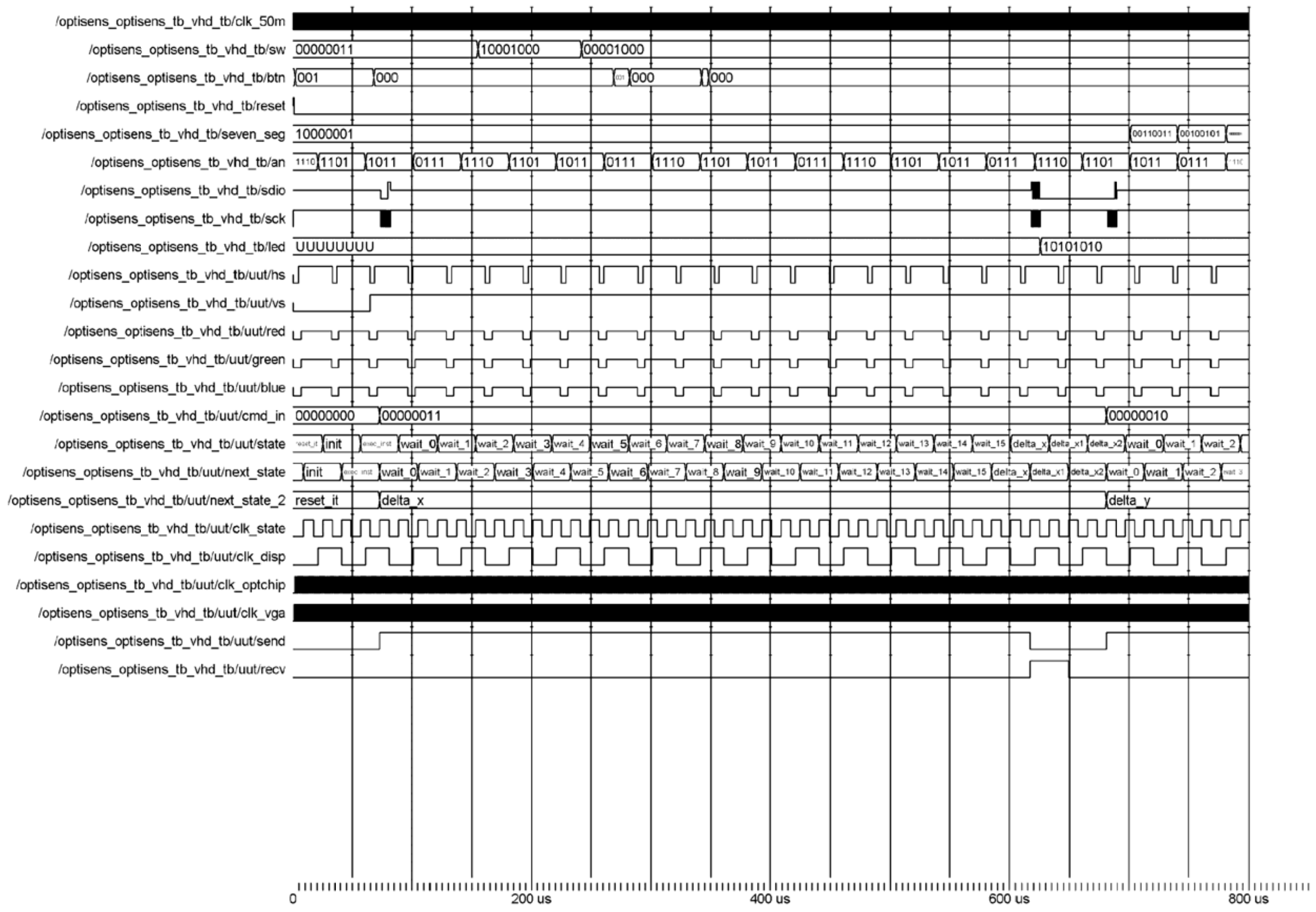
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY vga_vga_tb_vhd_tb IS
END vga_vga_tb_vhd_tb;

ARCHITECTURE behavior OF vga_vga_tb_vhd_tb IS

      COMPONENT vga
    Port ( clock       : in    std_logic;
           led         : in    std_logic_vector(7 downto 0);
           cmd_recv    : in    std_logic;
           data_rgb    : out   std_logic_vector(5 downto 0);
           hsync       : out   std_logic;
           vsync       : out   std_logic;
           rgb         : out   std_logic_vector(2 downto 0));
      END COMPONENT;

   SIGNAL clock     : std_logic := '0';
   SIGNAL led       : std_logic_vector(7 downto 0);
   SIGNAL cmd_recv  : std_logic := '0';
   SIGNAL data_rgb  : std_logic_vector(5 downto 0);
   SIGNAL hsync     : std_logic;
   SIGNAL vsync     : std_logic;
   SIGNAL rgb       : std_logic_vector(2 downto 0);

BEGIN

  uut: vga PORT MAP(
    clock    => clock,
    led      => led,
    cmd_recv => cmd_recv,
    data_rgb => data_rgb,
    hsync    => hsync,
    vsync    => vsync,
    rgb      => rgb
      );

   clock    <= not clock after 20 ns; -- 25 MHz clock ~ 50/2 MHz
   led      <= x"8F", x"4F" after 20 us;
   cmd_recv <= not cmd_recv after 40 ns;


   tb : PROCESS
   BEGIN
      wait; -- will wait forever
   END PROCESS;

END;
```

A.11.10 - optisens.ucf

```
NET "an<0>"  LOC = "d14"  ;
NET "an<1>"  LOC = "g14"  ;
NET "an<2>"  LOC = "f14"  ;
NET "an<3>"  LOC = "e13"  ;
NET "clk_50M"  LOC = "t9"  ;

NET "sck"    LOC = "d5"  ;
NET "sdio"   LOC = "c5"  ;

NET "seven_seg<0>"  LOC = "p16"  ;
NET "seven_seg<1>"  LOC = "e14"  ;
NET "seven_seg<2>"  LOC = "g13"  ;
NET "seven_seg<3>"  LOC = "n15"  ;
NET "seven_seg<4>"  LOC = "p15"  ;
NET "seven_seg<5>"  LOC = "r16"  ;
NET "seven_seg<6>"  LOC = "f13"  ;
NET "seven_seg<7>"  LOC = "n16"  ;

NET "led<0>"  LOC = "k12"  ;
NET "led<1>"  LOC = "p14"  ;
NET "led<2>"  LOC = "l12"  ;
NET "led<3>"  LOC = "n14"  ;
NET "led<4>"  LOC = "p13"  ;
NET "led<5>"  LOC = "n12"  ;
NET "led<6>"  LOC = "p12"  ;
NET "led<7>"  LOC = "p11"  ;

NET "red"     LOC = "r12";
NET "green"   LOC = "t12";
NET "blue"    LOC = "r11";
NET "hs"      LOC = "r9";
NET "vs"      LOC = "t10";

NET "btn<0>"  LOC = "m13"  ;
NET "btn<1>"  LOC = "m14"  ;
NET "btn<2>"  LOC = "l13"  ;
NET "reset"  LOC = "l14"  ;

NET "sw<0>"  LOC = "f12"  ;
NET "sw<1>"  LOC = "g12"  ;
NET "sw<2>"  LOC = "h14"  ;
NET "sw<3>"  LOC = "h13"  ;
NET "sw<4>"  LOC = "j14"  ;
NET "sw<5>"  LOC = "j13"  ;
NET "sw<6>"  LOC = "k14"  ;
NET "sw<7>"  LOC = "k13"  ;
```

A.11.11 - optisens_package.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package optisens_package is
  --Define constants for the various numbers that can be outputted
  constant zero     : std_logic_vector(7 downto 0) := "10000001";
  constant one      : std_logic_vector(7 downto 0) := "11110011";
  constant two      : std_logic_vector(7 downto 0) := "01001001";
  constant three    : std_logic_vector(7 downto 0) := "01100001";
  constant four     : std_logic_vector(7 downto 0) := "00110011";
  constant five     : std_logic_vector(7 downto 0) := "00100101";
  constant six      : std_logic_vector(7 downto 0) := "00000101";
  constant seven    : std_logic_vector(7 downto 0) := "11110001";
  constant eight    : std_logic_vector(7 downto 0) := "00000001";
  constant nine     : std_logic_vector(7 downto 0) := "00100001";
  constant ten      : std_logic_vector(7 downto 0) := "00010001";
  constant eleven   : std_logic_vector(7 downto 0) := "00000111";
  constant twelve   : std_logic_vector(7 downto 0) := "10001101";
  constant thirteen : std_logic_vector(7 downto 0) := "01000011";
  constant fourteen : std_logic_vector(7 downto 0) := "00001101";
  constant fifteen  : std_logic_vector(7 downto 0) := "00011101";
  constant black    : std_logic_vector(2 downto 0) := "000";
  constant blue     : std_logic_vector(2 downto 0) := "001";
  constant green    : std_logic_vector(2 downto 0) := "010";
  constant cyan     : std_logic_vector(2 downto 0) := "011";
  constant red      : std_logic_vector(2 downto 0) := "100";
  constant magenta  : std_logic_vector(2 downto 0) := "101";
  constant yellow   : std_logic_vector(2 downto 0) := "110";
  constant white    : std_logic_vector(2 downto 0) := "111";

  --Define addresses in the ADNS-2610
  constant read           : std_logic :=   '0'; --To add in front of address when reading
  constant write          : std_logic :=   '1'; --To add in front of address when writing

  constant config         : std_logic_vector(6 downto 0) := "0000000"; --x00 Reset, power
down, forced awake, etc.
  constant status         : std_logic_vector(6 downto 0) := "0000001"; --x01 Product ID,
mouse state of asleep or awake
  constant delta_y        : std_logic_vector(6 downto 0) := "0000010"; --x02 Y movement
  constant delta_x        : std_logic_vector(6 downto 0) := "0000011"; --x03 X movement
  constant squal          : std_logic_vector(6 downto 0) := "0000100"; --x04 Measure no.
of features visible by sensor
  constant max_pixel      : std_logic_vector(6 downto 0) := "0000101"; --x05
  constant min_pixel      : std_logic_vector(6 downto 0) := "0000110"; --x06
  constant pixel_sum      : std_logic_vector(6 downto 0) := "0000111"; --x07
  constant pixel_data     : std_logic_vector(6 downto 0) := "0001000"; --x08 Actual
picture of surface
  constant shutter_upper  : std_logic_vector(6 downto 0) := "0001001"; --x09
  constant shutter_lower  : std_logic_vector(6 downto 0) := "0001010"; --x0A
  constant inv_product_id : std_logic_vector(6 downto 0) := "0010001"; --x11 Inverse
product ID

  --Define the selection of digits
      constant digit0   : std_logic_vector(3 downto 0) := "1110";
      constant digit1   : std_logic_vector(3 downto 0) := "1101";
      constant digit2   : std_logic_vector(3 downto 0) := "1011";
      constant digit3   : std_logic_vector(3 downto 0) := "0111";
end optisens_package;

package body optisens_package is

end optisens_package;
```



Entity:optisens_optisens_tb_vhd_tb Architecture:behavior Date: Fri Apr 29 11:51:05 Eastern Daylight Time 2005 Row: 1 Page: 1

Appendix B - Pos3D.dll code

B.1 - Header file : Pos3D.h

```
#ifdef    __cplusplus      /* If C++, disable "mangling" */
extern "C" {
#endif /* __cplusplus */

/* ---------------------------------------------------------------------------------- */

#if defined(_POS3DLIB_)
#define POS3DAPI     __declspec(dllexport)
#else
#define POS3DAPI     __declspec(dllimport)
#endif

/* ---------------------------------------------------------------------------------- */

typedef struct PositionDesc {
              int    x;       // horizontal position perpendicular to the path
              int    y;       // vertical position perpendicular to the path
              int    z;       // distance (dz = distance between scan planes)
              int    tilt;    // tilt angle (atan(Y/X))
              int    elev;    // elevation angle (atan(Z/X))
} PositionDesc;

typedef struct MouseResolution {
              int    Res1;    // Resolution of the first mouse
              int    Res2;    // Resolution of the second mouse
} MouseResolution;

/* ---------------------------------------------------------------------------------- */

//__declspec(dllexport) int __stdcall Init();
POS3DAPI int __stdcall Init(void);
POS3DAPI int __stdcall UnInit(void);
POS3DAPI int __stdcall GetPosition(PositionDesc *pos);
POS3DAPI int __stdcall SetResolution(MouseResolution *res);
POS3DAPI int __stdcall GetResolution(MouseResolution *res);
POS3DAPI int __stdcall GetDLLVersion(int *DLLversion);

/* ---------------------------------------------------------------------------------- */

#ifdef __cplusplus
}
#endif /* __cplusplus */
```

B.2 - C-file : Pos3D.c

```
#include "stdafx.h"
#include <stdio.h>
#include "hapi.h"
#include "pos3D.h"

//Current DLL version
const char version=3;

//Shared memory segment for interprocess communication
//#pragma comment(linker, "/SECTION:.shared,RWS")
#pragma data_seg(".shared")
//Relation between pixels and real world distance (mm)
MouseResolution scale = {0};

//Last position read from the mouse
PositionDesc savedpos = {0};
#pragma data_seg()

//Function that initializes the mouse driver stack
POS3DAPI int __stdcall Init(void)
{

int count=0;
```

```
//count = hInitialise(0, NULL, GetDC(NULL), BUTTON | MOVEMENT | CURSOR | CLIP |
ACCELERATE);

count = hInitialise(0, NULL, GetDC(NULL), BUTTON | MOVEMENT | CURSOR);
printf("Initializing - I found %i mice using CPNMouse driver\n", count);

if(count) return 1;
else return 0;

}

//Function that uninitializes the mouse driver stack
POS3DAPI int __stdcall UnInit(void)
{
hCleanup();

return 1;
}


/*
typedef struct PositionDesc {
              int    x;       // horizontal position perpendicular to the path
              int    y;       // vertical position perpendicular to the path
              int    z;       // distance (dz = distance between scan planes)
              int    tilt;    // tilt angle (atan(Y/X))
              int    elev;    // elevation angle (atan(Z/X))
} PositionDesc;
*/

POS3DAPI int __stdcall GetPosition(PositionDesc *pos)
{
static int x=0,y=0,z=0;

hPOINT p1, p2;

hGetAbsolutePosition(1, &p1);
hGetAbsolutePosition(2, &p2);

//Print out the current position for debugging purposes
if( x != p1.x || z != p1.y)
{
    printf("Position   : %i,%i\n",p1.x,p1.y);
}

//Depends on the mounting of the optical sensor

x = -p1.y;
z = p1.x;

//Store the positions in the PositionDesc pointed to by pos
pos->x = (__int64) x*scale.Res1/400;
pos->y = 0;
pos->z = (__int64) z*scale.Res1/400;
pos->tilt = 0;
pos->elev = 0;

//Store the current position read in shared memory so another process can read it as well
savedpos.x = x;
savedpos.z = z;

return 1;
}

POS3DAPI int __stdcall LastPos(PositionDesc *pos)
{

pos->x = savedpos.x;
pos->y = 0;
pos->z = savedpos.z;
pos->tilt = 0;
pos->elev = 0;

return 1;
```

```
}

POS3DAPI int __stdcall SetResolution(MouseResolution *res)
{
printf("Setresolution : %i\n",*res);
scale = *res;

return 1;
}

POS3DAPI int __stdcall GetResolution(MouseResolution *res)
{
*res = scale;
printf("Getresolution : %i\n",*res);

return 1;
}

POS3DAPI int __stdcall GetDLLVersion(int *DLLversion)
{

*DLLversion = version;

return 1;
}
```

Appendix C - Matlab filtering code

testcp.m

```
no_scans = 9;

count = zeros(1,no_scans);

for scanno = 5:no_scans,

  scanno

  clear RGB J K y map mov movproc;

  filename = sprintf('tube_phantom_w_methylpar_%ia_cut.avi',scanno);

  fileinfo = aviinfo(filename);
  mov = aviread(filename);

  for i=1:fileinfo.NumFrames,

    [RGB,Map] = frame2im(mov(i));
    A = rgb2gray(RGB);
    B = wiener2(A,[5 5]);
    C = ~nearestn(B,1,35);
    C = uint8(C);
    map = [[0,0,0];[1,1,1]];

    %  map = colormap(gray);
    %  D = edge(C,'sobel', 0.001);
    %  D = uint8(D);

    movproc(i) = im2frame(C, map);
    count(scanno) = count(scanno) + sum(sum((movproc(i).cdata))');
  end

  outputfilename = sprintf('proc_tube_phantom_w_methylpar_%ia_cut.avi',scanno);
  movie2avi(movproc,outputfilename, 'colormap', map);

end

mperpix = 0.093964e-3; %Value from Sonocubic
pixperm = mperpix^-1;
resolution = (388/512*pixperm)^2*pixperm;
```

```
volume = count/resolution*100^3; %in cm3
rel = (pi/4*1.35^2*2.12)./volume; %in cm3
%rel = (pi/4*1.35^2*0.94)./volume; %in cm3
```

nearestn.m

```
function y = nearestn(image, N, thresh)

n = N;
th = thresh;
count = (2*n+1)^2;
a = double(image);
[row,col] = size(image);
for i = (1+n):(row-n)
    for j = (1+n):(col-n)
        x = 0;
        rowdn = i - n;
        rowup = i + n;
        coldn = j - n;
        colup = j + n;
        % Min row check
        if rowdn < 1
            rowup = n;
            rowdn = 1;
        end
        % Max row check
        if rowup > row
            rowup = row;
            rowdn = rowdn - n;
        end
        % Min column check
        if coldn < 1
            colup = n;
            coldn = 1;
        end
        % Max column check
        if colup > col
            colup = col;
            coldn = coldn - n;
         end
         for m = rowdn:rowup
            for k = coldn:colup
               x = x + a(m,k);
            end
         end
           sat = x/count;
        if sat > th
           res(i,j) = 100*sat;
        else
           res(i,j) = 0;
        end
     end
end
for j = 1:col
    for i = 1:(1+n)
       res(i,j) = 100*sat;
    end

    for i = (row-n):row
       res(i,j) = 100*sat;
    end
end
for i = 1:row
    for j = 1:(1+n)
       res(i,j) = 100*sat;
    end

    for j = (col-n):col
       res(i,j) = 100*sat;
    end
end

  y = res;
return;
```

Appendix D - Customized Teratest code for testing

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <windows.h>
#include "..\..\Pos3D\Pos3D\Pos3D.h"
#include "uslink.h"
#include <math.h>
/* ------------------------------------------------------------------------------ */

int offset( BITMAPINFO *dib, char *bits, int new_offset )
{

int i, j;
int width=0, height=0, offset=0;
static int old_offset;

offset = new_offset - old_offset;
old_offset = new_offset;

height = dib->bmiHeader.biHeight;
width  = dib->bmiHeader.biWidth;
//res    = dib->bmiHeader.biXPelsPerMeter;

if(offset > 0)
for(i=0;i<height;i++) {
    for(j=0;j<(width-offset);j++) {

    *(bits+i*width+j) = *(bits+i*width+j+offset);

    }

    for(j=(width-offset);j<width;j++) {

    *(bits+i*width+j) = 0;

    }
  }

else if(offset < 0)
for(i=0;i<height;i++) {
    for(j=width;j>-offset;j--) {

    *(bits+i*width+j) = *(bits+i*width+j+offset);

    }

    for(j=0;j<-offset;j++) {

    *(bits+i*width+j) = 0;

    }
  }


  return 1;
}

/* ------------------------------------------------------------------------------- */
static void put( char *bits, PositionDesc pos, int buf_width, int buf_height )
{
        int  a,i,j;
        char *p;

        a = buf_width/2 + buf_width*buf_height/2 - pos.x;
        p = bits + a;

        for ( i = 0; i < buf_width; i++ ) {
                *(bits+i) = 0x00;
            for ( j = 0; j < buf_height; j++ ) {
                *(bits+j+i*buf_width) = 0x00;
                }
        }

        a = pos.z;

        if(a < 20) a = 0;
        else a = a-20;
        if(a > 88) a = 0;//88;
        //a = 40;

        //Make sure that we stay inside the frame
        if((a+pos.x) >= buf_width/2) a = buf_width/2-pos.x;
        else if((-a+pos.x) <= -buf_width/2) a = buf_width/2+pos.x;

        //Place box

        //Horizontal placement
        for (i=-a; i<=a; i++ ) {
        //if(i <= buf_height && i >= -buf_height)
        //Line number
          for (j=-a; j<a; j++ ) {
        //if(j <= buf_width/2 && j >= -buf_width/2)
                *(p + i + j*buf_width) = 0xFF;
              }
        }
}
/* ---------------------------------------------------------------------------------- */
static BITMAPINFO *create_dib( int buf_width, int buf_height, char **pbits )
/*
        Alloc DIB and fill it with a test pattern. Free it with free(lpbi).
*/
{
        BITMAPINFO      *lpbi;
        RGBQUAD         *tbl;
        char            *bits, *pos, *p;
        int             n, i, j;

        // Alloc bitmap
        if ( (lpbi = (LPBITMAPINFO)calloc( sizeof(BITMAPINFO) + 256 * sizeof(RGBQUAD) +
buf_width * buf_height, 1 )) == NULL )  {
                return( NULL );
        }
        // Fill header
        lpbi->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
        lpbi->bmiHeader.biWidth = buf_width;
        lpbi->bmiHeader.biHeight = -buf_height;
        lpbi->bmiHeader.biCompression = BI_RGB;
        lpbi->bmiHeader.biBitCount = 8;
        lpbi->bmiHeader.biXPelsPerMeter = 1000;
        lpbi->bmiHeader.biYPelsPerMeter = 1000;

        tbl = (RGBQUAD*)((char*)lpbi + sizeof(BITMAPINFOHEADER));

        bits = (char*)lpbi + sizeof(BITMAPINFO) + 256 * sizeof(RGBQUAD);

        // Fill color table
        for ( i = 0; i < 255; i++ ) {
                tbl[i].rgbBlue = tbl[i].rgbGreen = tbl[i].rgbRed = (BYTE)i;
        }
        tbl[255].rgbRed   = 0;
        tbl[255].rgbGreen = 0xFF;
        tbl[255].rgbBlue  = 0xFF;

        *pbits = bits;

        return( lpbi );
}


/* --------------------------------------------------------------------------------- */
int     main( int argc, char *argv[] )
/*
        teratest <delay> <width> <height>
        <delay>: Time in ms to wait between calls to SendDIBToSonocubic, default 30.
                 This depends on OS time granularity. A real ultrasound application
                 should continue its work instead of calling Sleep(), most important,
                 it must continue processing Windows messages.
        <width>: DIB width, default 256
        <height>: DIB height, default 256
*/
{
        BITMAPINFO          *lpbi;
        char                *bits;
        int                 n, num = 0;
        HMODULE             h;
        pSendDIBToSonocubic psnd;
        int                 buf_width = 256, buf_height = 256, delay = 30;

        char                bfHeader[14];
        char                bmiHeader[40];
        const char          no_images=9;

        BITMAPFILEHEADER    *lpbfHeader;
        BITMAPINFOHEADER    *lpbmiHeader;
        BITMAPINFO          *lpbiHeader;

        RGBQUAD             *lpbmiColors, *tbl;

        FARPROC             Init,CurPos,SetRes;
            LONG                *xres,*yres;
        int                 g,i, j=0, k, r, s, numread, numwritten, width, height, sum,
a,b,c, maxi=0;
        PositionDesc        p, oldpos = {0}, pos = {0};
        HINSTANCE           hLib;

        MouseResolution res;

        FILE *stream;
        char list[30], buffer[100];

        double    oldtime=0, time=0;
        int       difftime=0;


        if ( argc >= 2 ) {
                delay = atoi( argv[1] );
        }
        if ( argc >= 4 ) {
                buf_width = atoi( argv[2] );
                buf_height = atoi( argv[3] );
        }
        // Create DIB and fill it with a test pattern
        if ( (lpbi = create_dib(buf_width, buf_height, &bits)) == NULL ) {
                printf( "DIB couldn't be created\n" );
                return(1);
        }
        // Frame counter position in DIB
        //pos =  bits;// + buf_width / 2 - CH_WIDTH * 2 + buf_width * buf_height / 2 -
CH_HEIGHT / 2;


        //Load the positioning library (Dynamic Link Library)
        hLib = LoadLibrary("c:\\adq\\Pos3D.dll");

        if(hLib==NULL)
        {
              printf("Unable to load Pos3D.dll library!\n");
              getch();
              return 0;
        }

        Init    = GetProcAddress((HMODULE)hLib, "_Init@0");
        CurPos  = GetProcAddress((HMODULE)hLib, "_CurPos@8");
        SetRes  = GetProcAddress((HMODULE)hLib, "_SetResolution@4");

        if(Init == NULL || CurPos == NULL || SetRes == NULL )
        {
              printf("Unable to load function(s).%i,%i,%i\n",Init,CurPos,SetRes);
              FreeLibrary((HMODULE)hLib);
              return 0;
        }
        // Load interface DLL
        if (  ((h = LoadLibraryEx("c:\\adq\\uslink.dll", NULL,
LOAD_WITH_ALTERED_SEARCH_PATH)) == NULL)
           || ((psnd = (pSendDIBToSonocubic)GetProcAddress( h, "_SendDIBToSonocubic@8" ))
== NULL)
           ) {
                printf( "c:\\adq\\uslink.dll couldn't be loaded\n" );
                if ( h != NULL ) {
                      printf( "freeing library\n" );
                        FreeLibrary( h );
                }
                free( lpbi );
                return(1);
        }
        i=0;

        res.Res1 = 1600000;
        res.Res2 = 1600000;

      /* Open for write */
      if( (stream = fopen( "c:\\Matlab\\work\\data.txt", "w+t" )) != NULL )
            {
      printf( "The file 'data.txt' was not opened\n" );
            /* Write 25 characters to stream */
            //numwritten = fwrite( list, sizeof( char ), 25, stream );
            //printf( "Wrote %d items\n", numwritten );
            }
            else
            printf( "The file 'data.txt' was opened\n" );

            //SetThreadPriority(GetCurrentThread,THREAD_PRIORITY_HIGHEST);
            //Set the old time where we begin our sweep
        oldtime = GetTickCount();
        // Begin sending frames. Press a key to stop
        while( !kbhit() ) {
                                            //Get the current time in form of ticks
                time = GetTickCount();
                //Find the difference in time since we began our sweep
                difftime = time - oldtime;
                //Scale with a factor 10
                difftime = difftime / 10;
                //Get the current position that depends on the time
                CurPos(&p,difftime);
                //Write our position to the file together with the current time
                sprintf( buffer, "%d,%d,%i\n", p.z, p.x, difftime );
                fwrite( buffer, sizeof( char ), strlen(buffer), stream );

                if(oldpos.x != p.x || oldpos.z != p.z) {
                    //printf("Position : %i,%i\n", p.x, p.z);
                    oldpos = p;
                }
                //generate scanplane offset
                put( bits, p, buf_width, buf_height );  // Draw frame counter

                //Send scanplane to Sonocubic
                n = psnd( lpbi, bits );            // Send DIB to Sonocubic

                Sleep( 300 );                    // Delay

        }
                        if(stream) fclose( stream );
        printf( "Waiting for link to be closed...\n" );

        while( psnd( lpbi, NULL ) != 0 )  {
                Sleep( 30 );
        }

        FreeLibrary( h );       // Free interface DLL
        free( lpbi );
        return(0);
}
```

Appendix E - Customized Pos3D.dll code for testing

```
// Pos3D.cpp : Defines the entry point for the DLL application.

#define _USE_MATH_DEFINES

#include "stdafx.h"
#include <stdio.h>
#include "hapi.h"
#include "pos3D.h"
#include <math.h>

const char version=3;
//Relation between pixels and real world distance (mm)
//#pragma comment(linker, "/SECTION:.shared,RWS")
#pragma data_seg(".shared")
MouseResolution scale = {0};
PositionDesc savedpos = {0};
#pragma data_seg()

/* Function that initializes the mouse driver stack */
POS3DAPI int __stdcall Init(void)
{
int count=0;

//count = hInitialise(0, NULL, GetDC(NULL), BUTTON | MOVEMENT | CURSOR | CLIP |
ACCELERATE);
//count = hInitialise(0, NULL, GetDC(NULL), BUTTON | MOVEMENT | CURSOR);
count = 1;
printf("Initializing - I found %i mice using CPNMouse driver\n", count);

//Initialize the mouse resolution defaulting to 400 cpi
//scale.Res1 = 400;
//scale.Res2 = 500;

printf("scale.Res1 = %i\n", scale.Res1);

if(count) return 1;
else return 0;

}

/* Function that initializes the mouse driver stack */
POS3DAPI int __stdcall UnInit(void)
{
hCleanup();

return 1;
}


/*
typedef struct PositionDesc {
              int    x;      // horizontal position perpendicular to the path
              int    y;      // vertical position perpendicular to the path
              int    z;      // distance (dz = distance between scanplanes)
              int    tilt;   // tilt angle (atan(Y/X))
              int    elev;   // elevation angle (atan(Z/X))
} PositionDesc;
*/


POS3DAPI int __stdcall CurPos(PositionDesc *pos, int time)
{
double x=0,y=0,z=0, a=0, b=0;

//40 mm corresponds to 40/25.4 inches.
//40/25.4 inches corresponds to 1575 counts
//40 mm offset varying over a period of 1 sec
x = 1575*sin((time*2*M_PI)/1000);
//x = 0;

//For a constant speed of 40 mm/sec
//z = (1575*time)/1000;

//For a variable speed of 40 mm/sec +/- 20 mm/sec
z = (1575*time)/1000 - (887*cos((2*M_PI*time)/1000))/(2*M_PI) + 887/(2*M_PI);

//Picture resolution is 1000 pixels per meter
pos->x = x*254/10000; //The same as x*25.4/1000
pos->y = 0;
pos->z = z*254/10000;
pos->tilt = 0;
pos->elev = 0;

//For variable speed
savedpos.x = x;
savedpos.z = z;

//For constant speed or offset
//savedpos.x = 0;
//savedpos.z = (1575*time)/1000;

return 1;
}

POS3DAPI int __stdcall GetPosition(PositionDesc *pos)
{
static int x=0,y=0,z=0;

x = savedpos.x;
z = savedpos.z;

pos->x = (__int64) x*scale.Res1/1000;;
pos->y = 0;
pos->z = (__int64) z*scale.Res1/1000;
pos->tilt = 0;
pos->elev = 0;

return 1;
}

POS3DAPI int __stdcall SetResolution(MouseResolution *res)
{
printf("Setresolution : %i\n",*res);
scale = *res;

return 1;
}

POS3DAPI int __stdcall GetResolution(MouseResolution *res)
{
*res = scale;
printf("Getresolution : %i\n",*res);

return 1;
}

POS3DAPI int __stdcall GetDLLVersion(int *DLLversion)
{

*DLLversion = version;

return 1;
}
```