GIJS BOER

AN INCREMENTAL APPROACH TO REAL-TIME HAND POSE ESTIMATION USING THE GPU

[May 25, 2010 at 1:29]

AN INCREMENTAL APPROACH TO REAL-TIME HAND POSE ESTIMATION USING THE GPU

GIJS BOER

A thesis submitted for the degree of Master of Science in Computing Science

May 2010

[May 25, 2010 at 1:29]

Gijs Boer: *An incremental approach to real-time hand pose estimation using the GPU,* A thesis submitted for the degree of Master of Science in Computing Science, © May 2010

ABSTRACT

The research presented is part of a project called "Augmented Reality for 3D Multi-user Interaction," or ARMI for short. The goal of project ARMI is to develop a system that allows multiple users to interact with an augmented reality using their hands as input. Interaction is performed without making any use of a mouse or keyboard. Also, no markers or gloves will be attached to the hands. The augmented reality is shared across the Internet so that multiple users can interact with the same environment. This allows both users to discuss and change a design of a building, for instance. The hands of the users are replicated and displayed as virtual models so that each user knows what the other one is pointing at. The augmented reality is displayed by making use of a head mounted display.

A total of four different areas are researched for project ARMI. These are: the 3D interface to display the interactions and the augmented reality, the replication algorithm to communicate the changes made to the environment, a hand tracking algorithm that tracks the user's hands in the video feed, a hand pose estimation (HPE) algorithm to determine the correct pose and position of the hand. The HPE algorithm is described in this thesis. To make sure there is enough processing power available, the HPE algorithm is run on the GPU. To make optimal use, the best way to perform calculations on the GPU is researched. Afterwards, the 3D hand model is made which will be used to match the model onto the real hand in the video feed. The total degrees of freedom (DOF) of a hand can be minimized to nine DOFs and five weak constraints. Also, the movement of the fingers is constrained so the hand model can also incorporate these constraints to decrease the total search space which in turn improves performance.

The HPE algorithm receives the input from the hand tracker which marks each pixel that is part of the hand. The image is fed through a Sobel operator to retrieve all relevant edge information. Now, a search algorithm adjusts the hand model so that it matches the real hand in the video feed. This is done by subtracting the edges of the 3D model from the edges of the video feed. To determine whether certain settings result in a good fit, all the pixel information that is left after subtraction is summed together. This results in a value which describes the error of a particular setting. The search space, which contains all the settings, is searched through by an optimization algorithm to find the best fit as fast as possible. Three different optimization algorithms are evaluated: Secant method, Nelder-Mead, and Simulated Annealing. Each algorithm is tested to see if they are able to track a ball, an oblong, and a hand. The Simulated Annealing method gave the best results when compared to the other two methods.

The final implementation of the system is able to successfully track the hand in the video feed. However, it is not able to accurately determine

the complete pose of the hand. Also, it is not able to perform the estimation process in real-time which makes it hard to use for augmented reality. Many improvements can be made however. The input, speed, and estimation process can all be optimized. All in all, the research shows promise and has many possible applications.

ACKNOWLEDGMENTS

First of all I'd like to thank my supervisor, Michael Wilkinson, for the support, guidance, and advice he gave me for my research and this thesis. Also, Prof. Dr. Marco Aiello and Dr. Tobias Isenberg for their help during the startup phase of the project.

Furthermore, I'd like to thank my friends Pieter Bruining, Maarten Fremouw, and Heino Lenting for accepting and creating my idea for the ARMI project. Also, my little brother Steven Boer for helping me with the math problems that I encountered and helping me to solve problems and give suggestions for the algorithm. Equally important, my other little brother Robin Boer for helping me to create most of the illustrations in this thesis.

Finally I'd like to thank the following people for proof reading my thesis: Heino Lenting, Mark van Halsema, and Steven Boer.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API Application Programming Interface

AR Augmented Reality

COTS Commercial, Off-The-Shelf

DMA Direct Memory Access

DOF Degrees Of Freedom

FBO Frame Buffer Object

FLOPS FLoating point Operations Per Second

FPS Frames Per Second

GLSL OpenGL Shading Language

GPS Global Positioning System

GUI Graphical User Interface

HMD Head-Mounted Display

HPE Hand Pose Estimation

LIDAR LIght Detection And Ranging

OPENGL Open Graphics Library

1 INTRODUCTION

This introduction chapter will explain the goal of this master thesis. It will discuss the current state of research surrounding this master thesis. Afterwards, a problem statement will be given, to see what kind of problems have to be solved. Finally, an overview of the entire thesis will be given.

Before explaining what the goal of this master thesis is, a short explanation of Augmented Reality (AR) will be given first.

1.1 AUGMENTED REALITY

AR describes a technique that involves placing virtual objects on top of the real physical world. In other words, reality is augmented with virtual objects, they become part of reality. These virtual objects can for instance be used to display information about real physical objects. AR can be used in many different ways. One of the most familiar is AR on TV. For instance, the yellow "first down" line in American football is shown using AR, as can be seen in Figure 1a. Another example of AR is a special version of the first person shooter Quake called ARQuake. A team at the university of South Australia, initially lead by professor Bruce Thomas, adapted Quake to work with the latest mobile AR technology. A screenshot of what ARQuake looks like is shown in Figure 1b.

(a) The yellow "first down" line (source: HowStuffWorks [19]).

(b) ARQuake (source: ARQuake Project [12]).

Figure 1: Several AR examples, showing the "first down" line in American football and ARQuake.

Since AR applications can be found in many different forms, it is helpful to have a clear definition of AR. It is defined by Azuma to have three different characteristics [6]. Augmented reality:

1. combines real and virtual;
2. is interactive in real time;
3. is registered in 3-D.

Depending on the application, AR might need different hardware, but two things remain the same for all AR applications. First, the application needs accurate localization. This can, for instance, be provided by a GPS combined with an electronic compass [12]. Another method is to visually inspect the target at which AR needs to be shown with the use of a webcam. The location and orientation is used to correctly place the virtual object in the physical world. And secondly, the application needs some kind of way to show the virtual objects. This can be done through the use of a head-mounted display (HMD) or a regular screen from a TV or mobile phone. For more details about the problems and applications of AR, see the extensive survey of Azuma [6].

1.2 GOAL MASTER THESIS

Recent developments concerning hardware and research in the field of Augmented Reality have made it possible to build usable AR applications. One thing that remains a problem is the interaction with an AR environment. To interact with a 3D virtual world is completely different from what we are used to when compared to normal 2D computer interaction. However, in the normal world people perform 3D interactions on a daily basis, using our hands instead of devices like a keyboard or a mouse. In order to provide the most intuitive 3D interaction for AR, our hands would be the best option available without making use of extra gear. However, in order to make complete use of the human hand, tracking and complete pose estimation in an unrestricted environment would be required, which still remains a problem to date. Data gloves offer a way of tracking a hand, but they are costly and difficult to configure [14]. They also do not offer an unrestricted way to interact with a virtual environment since the user would need to wear the gloves.

This master thesis focuses on an incremental approach to hand pose estimation (HPE) using the GPU to provide an AR-environment with an input "device". HPE is a name for algorithms that can estimate where the human hand is positioned and how it is oriented. HPE algorithms also provide information on the angles of all or some fingers. Without reduction, a complete human hand has 23 degrees of freedom (DOFs): four for every finger, five for the thumb, and two more for the bones that connect the little and ring finger with the wrist (see Figure 7 for details) [47]. Another six DOFs would be needed to describe the pitch, yaw, and roll parameters as well as the x, y, and z coordinates of the hand. This makes a total of 29 DOFs that need to be determined real-time if

used for AR applications. The final goal of this thesis is to develop and test a 3D interaction "device" using HPE.

1.3 STATE OF THE ART

In current research, hand pose estimation can be found using several different algorithms and hardware [14]. The most reliable hardware and also the most reliable method at the moment is by making use of a data glove [14]. A data glove is a glove with sensors on it to measure the angles of the joints of the fingers, sometimes accompanied by magnetic sensors to provide tactile response in virtual environments. These data gloves are quite expensive, at the time of writing, ranging from $3600 (X-IST Data Glove HR1, 15 DOF glove) up to $5495 (5DT Glove 14 ultra, 14 DOF), and they are not easy to set up [14]. A different strategy is by using vision-based techniques which have the possibility to be very cheap. Vision-based techniques make use of infrared or normal cameras to register the hands of the user [33, 39]. The vision-based HPE approach can be subdivided into two areas [27]. The first area requires the user to wear a glove with distinct markers or colors. These distinct features provide the algorithms with easier detection and estimation. One particular research performs updates of the hand pose at 10 Hz with a color glove [53]. The second area is where the user does not have to wear anything special and the user can just use his or her hands. The second area of research can again be subdivided into two areas: model-based and appearance-based approaches [31].

Model-based approaches use a 3D model to compare the image features of the 3D hand model and the hand images retrieved from the camera(s). The state of the 3D model that best fits onto the image is assumed to be the correct state of the hand [29]. Several techniques are available to solve the problem using a model-based approach. For instance, a database-approach has been proposed by Zhou and Huang [55], and Athitsos and Sclaroff [5]. The database is used to search through the possible states and calculate the error between the observed image and the possible states. Since not every pose can be stored, the usual result of this technique is that there will always be a relatively large error. More samples would be required to decrease this error, however this would mean that it would take even longer to search through the entire database. To solve this problem, Lin, Wu, and Huang proposed to use a database with training examples to provide a rough estimate of the hand pose [29]. After this a particle filter uses this rough estimate to further increase the accuracy of the pose estimation. Particle filters use the current state and a probability distribution to predict what the next state will be. During the next state, the predicted state and the estimated state are compared and the error between them is reduced in order to provide a better prediction next time. They have been used extensively in many different forms [7, 8, 20, 24, 44, 45].

The second and last area, appearance-based approaches, attempt to provide pose estimation directly from image features. Nonlinear mappings are learned from a large number of training images [29]. Lin, Wu,

and Huang determined that it is possible to provide quick estimates of the hand pose once the mapping is learned. However, it is difficult to determine the optimal structure of the mapping function [29]. Various types of data gloves are used to gather the training data, like the CyberGlove. Rosales et al. use this data to render 3D models of the hand [41]. From these rendered models, the image features are learned. The image features are extracted from the video feed by performing hand segmentation using the color of the skin.

The performance of each proposed HPE algorithm differs quite substantially. Early algorithms needed up to 80 minutes to process each frame [25]. Other algorithms are able to perform at a rate of 30 frames per second [4, 22, 48]. Each of them also have varying abilities. Some are able to determine all DOFs while others are only able to point out which finger the index finger is [32]. An interesting note is that almost all of them are implemented on the CPU. Only on a rare occasion the GPU is used [42].

One of the problems regarding model-based approaches is that the 3D model should closely fit the user's hand in order for the algorithm to provide a good estimate of the hand pose. Given a well-initialized 3D model, the technique can provide accurate results [29]. A drawback from this technique is that the search is done in a very high-dimensional space (29 DOFs), which results in high computational complexity. However, previous work by Chua et al. has shown that the hand motion is highly constrained [11]. The research, using a 27 DOF hand model instead of 29, was able to bring the 27 DOFs down to 12 DOFs without any significant loss of accuracy.

Appearance-based approaches have the disadvantage that they require an initial calibration phase. This calibration phase requires expert knowledge in some cases, as in the research of Heap and Hogg [18]. This is not something that a user would be able to do, nor should he or she need to. Other research requires a data glove for calibration [41]. This would make such a system unnecessarily expensive. Closely related to the calibration problem, is that most systems assume that only one user will use the system. This means that it cannot be used as a generic input device. Another problem is that some techniques do not offer real-time performance. For a technique to be remotely usable, it would have to at least be able to run in real-time.

In short, a perfect HPE technique that provides a generic input device is able to:

- provide real-time estimation;
- be usable by different users;
- provide brief automatic calibration, if calibration is required;
- provide information about all 29 DOFs of the hand;
- be constructible with cheap COTS hardware;
- provide estimation without making use of extra hardware.

None of the research reviewed is able to fulfill all of these requirements. Therefore research has to be performed to improve on existing techniques or create new techniques to provide a generic input device for an AR user.

1.4 PROJECT ARMI

The research performed in this master thesis is part of a project called ARMI. ARMI, which stands for "Augmented Reality for Multiuser 3D Interaction," is a project that has the final goal to develop an affordable AR application. In this application it is possible for multiple users to interact with virtual objects in a shared virtual environment. The users each have their own table with the necessary equipment. Users can connect to other users through a network like the Internet so that they can share their virtual environment. Apart from being able to see the shared environment, users can also simultaneously interact with this shared environment. Each user can see the other users' hands so that other people in the same shared environment know where he or she is pointing at or what he or she is doing. The virtual environment is augmented onto a table and can contain 3D models of any shape, as long as they do not exceed the physical size of the table. An example of a possible use for this environment might be an architect, virtually meeting with a customer and discuss a design of a house or a building. Both can see each others hands in the virtual world so both can see what the other is talking about. The architect or customer can move, scale, and rotate virtual objects using their hands. No extra mouse or keyboard is needed to interact with the system.

As explained in Section 1.1, AR needs at least two things: localization information and a way of showing the virtual objects. To show the virtual objects, the user wears an HMD. A webcam is attached to the HMD to supply the HMD-display with a video-stream of the real world. The webcam records at 30 frames per second with a resolution of 640 by 480 pixels. Both the camera and the HMD can be seen in Figures 2a and 2b. The video-stream can now be augmented with virtual objects.

(a) HMD - side view.

(b) HMD - front view.

Figure 2: The SPC1000NC webcam from Philips mounted on top of the iWear VR920 HMD from Vuzix (source: master thesis Lenting [26]).

To correctly place the virtual objects, localization information is needed. The localization information is provided by a software system called ARToolkit, which makes use of markers or tags. This tag recognition software is used to detect the tag in the video-feed, sent from the webcam mounted on the HMD. An example of such a tag is shown in Figure 3a, in the middle of the table. With ARToolKit it becomes possible to detect such a tag in the video-feed and estimate its position and orientation. This can then be used to augment the video-feed with virtual objects, as can be seen in Figure 3b where a teapot is drawn on top of the tag. The video-feed received from the webcam is augmented with the virtual objects and sent to the HMD to complete the AR environment.

(a) The tag used for tag-recognition.

(b) A virtual teapot is displayed on top of the tag.

Figure 3: An AR example, using ARToolKit to detect the tag and display virtual objects (source: master thesis Lenting [26]).

Multiple fixed tags will be used to position the AR environment on top of a table. The reason that multiple tags are used, is because that when a tag is occluded by something, the system is unable to retrieve localization information from that particular tag. If there are multiple tags and one is occluded by an object, the system can still use the localization information retrieved from the other tags.

Project ARMI is divided up into four different parts:

- A 3D interface to supply the user with an understandable environment in which interaction is obvious and easy.
- A replication-algorithm to communicate actions and transfer objects between each system that is connected to the virtual environment.
- A hand tracking algorithm to track the hands of the user in the video feed of the camera.
- An HPE algorithm to determine the exact angles and position of the hands of the user.

The development, implementation, and testing of the HPE algorithm is described in this master thesis. The other project members are:

- Pieter Bruining - 3D interface development [10];
- Heino Lenting - Replication-algorithm [26];
- Maarten Fremouw - Hand tracking algorithm [15].

The result of all of these sub-projects will come together in one final application.

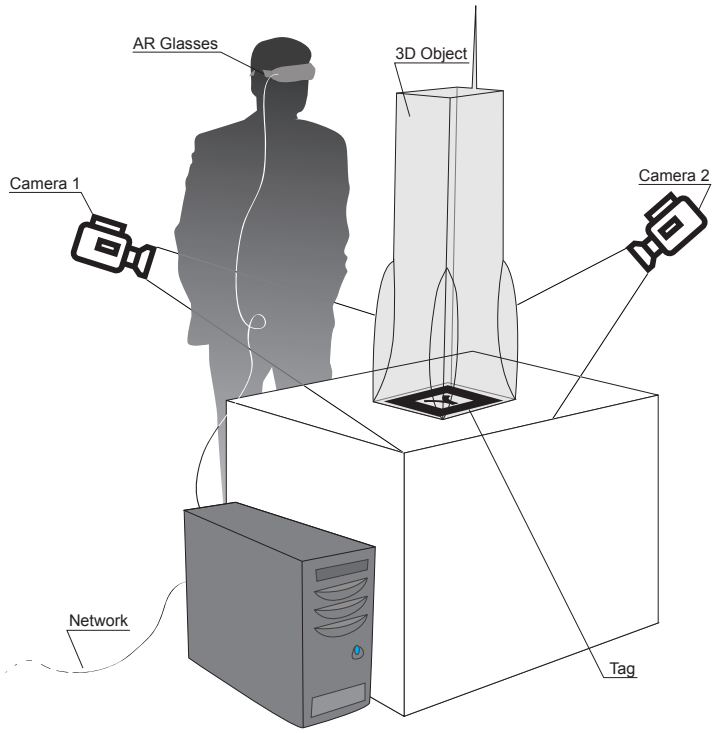


Figure 4: The setup for ARMI, shown with one tag here. (author: R. Boer).

The final setup of project ARMI would look like the illustration shown in Figure 4.

1.5 PROBLEM STATEMENT

One of the first problems that need to be solved, is the development of a real-time pose estimation algorithm. The HPE algorithm should be able to deliver the system with a reasonable accurate pose estimation that can be used for interaction in AR. The real-time aspect of the system is quite important, since a system that lags or stutters is not usable. Second, automatic calibration should be possible, in one form or another. All this should be possible, ideally, without having to resort to markers on the hand or other hardware. This is to be as unrestricted and user-friendly as possible.

1.6 OVERVIEW THESIS

The thesis is divided into different chapters that each describe a particular area. Each area describes problems and details which finally leads to a complete implementation of the hand pose estimation algorithm.

Chapter 2 starts out discussing the requirements of the system. It continues with a general description how computations are done on the GPU. The 3D hand model and the hand tracker are also introduced, as well as an edge enhancement filter to filter the relevant areas for the search algorithms. Finally, in Chapter 2, the theoretical details of the search algorithms are explained.

Chapter 3 gives a general system overview of how the hand pose estimation algorithm works. Each step of the system is then explained in detail. The hand tracker implementation is discussed as well as a description on how the search algorithms are implemented, along with their specific implementation problems. Afterwards, the GPGPU techniques are introduced to make efficient use of the GPU. Finally, the error functions are described and a final implementation overview is given which shows each component and how it interacts with the rest of the system.

The tests that are done to see whether the algorithm works as it should is described in Chapter 4, along with their results. The conclusion of the thesis is described in Chapter 5 and future improvements are discussed in Chapter 6.

2

ESTIMATING THE HAND POSE

The goal of this chapter is to show how the complete HPE algorithm works. It discusses what kind of choices have been made, as well as the motivations behind those choices. It also shows several problems that have occurred during the process of creating a workable solution.

2.1 REQUIREMENTS

When the project started, several requirements were set as to how the 3D interaction should be performed:

- It should not be needed to wear anything except an HMD. So no gloves or markers should be used.
- The system should be usable within minutes for any user. No gathering of training data with extra equipment or hours of training and testing should be done.
- It should be real-time. Real-time in this case means that the algorithm should be able to perform updates on the hand at a rate of 15 frames per second.

Another implicit requirement was that the system should not be expensive. Equipment like a LIDAR (the laser equivalent of the RADAR) or structured light scanners would be too expensive. So different cheap COTS hardware should be used.

2.2 ANALYZING REQUIREMENTS

During the research of current HPE algorithms, it became clear that there were only two categories that would match the requirements set beforehand. Non-computer vision solutions, for instance data-gloves, would be too expensive. Model-based or appearance-based algorithms were the only remaining choices. Since appearance-based algorithms need a lot of time training and testing its data and offer no general solution for every user, this was also quickly ruled out. A model-based algorithm looked like a possible solution to perform the estimation. Also because the 3D hand model needs to be created anyway, since the video feed is augmented with the 3D hand model for visual feedback.

A model-based solution tries to match a 3D model of the hand of the user to the actual hand of the user in the video-feed. For this to work, it requires a number of things. First, it needs to know where the hand is located inside the video-feed. This is done using a hand tracking algorithm. It also requires a model of the hand that is a close or perfect match of the hand of the user. Obviously, a better model will provide more accurate results. And finally, it requires a method of matching the 3D hand model against the hand seen in the video-feed. The beauty of this method is that it does not need to know which finger is where in the image. It just assumes that the pose that has the smallest error, represents the best possible fit, without having to know where each finger is. The focus of this master thesis is not the hand tracking algorithm, but it instead focuses on the hand model and the matching of this model against the hand in the video-feed. The hand tracking algorithm will be developed by Maarten Fremouw, as a different part of project ARMI [15].

Since a model-based algorithm requires a lot of image processing, it became clear that a normal CPU might not be enough to satisfy the real-time requirement. A GPU on the other hand is built for real-time image processing and should be able to handle the job. The difference between a GPU and a CPU is that a GPU performs calculations on pixels and vertexes in parallel, whereas a CPU performs calculations on floats and integers in sequence. CPUs nowadays do have some parallelism in the form of multiple cores and SSE instructions, but the GPU has much more raw processing power at the time of writing. Currently the CPU with the most processing power, the Intel Xeon W5590, has 53.28 gigaFLOPS of computing power [50]. One of the fastest GPUs at the moment, the AMD HD5870, can achieve up to 544 gigaFLOPS on double precision floats [51]. However, not all calculations can use the full potential of a GPU. As mentioned in the research performed by Trancoso and Charalambous, a GPU works at its best when several conditions are met [49]. These conditions are:

1. Format the input into two-dimensional arrays;
2. process large data arrays in every pass;
3. perform a considerable amount of simple operations per data element.

Since the GPU is specialized at performing calculations on images or textures, the input data should consist of two-dimensional arrays. This first condition is easily met since the HPE algorithm mostly processes rendered images which are in essence two-dimensional arrays of data. The last two conditions have to do with the overhead that is involved when performing calculations on the GPU. The HPE algorithm should perform as many calculations on as much data as possible for every pass.

Apart from the previously mentioned three conditions, another condition was set by Trancoso and Charalambous. Since the calculations of the GPU reside in the GPU memory, it has to be read back to the CPU

memory when the program wants to do something with it. Trancoso and Charalambous observed that sometimes this single action of reading the data back into CPU memory, would consume up to 50% of the entire processing time. Why this happens has two reasons. First being the bandwidth between the GPU and CPU. At the time of writing, one of the fastest graphics card, the AMD HD5870, has an internal memory bandwidth of 153.6 GB/s [51]. The PCIe 2.0 x16 interface, that is used to transfer data between the GPU and CPU, only has a maximum bandwidth of 8 GB/s [43]. To keep maximum performance a program would want to keep the data sent back and forth between the GPU and CPU at a minimum. The second reason why reading takes up so much time has to do with buffers inside the GPU. When the CPU asks the GPU for data, the GPU has to finish processing all its commands present in the buffers. During this time, the CPU waits for the GPU to finish. When the GPU is finished it stalls while waiting for new commands from the CPU. Obviously, during the time that either the CPU or GPU waits for the other, no calculations can be performed. So to make maximum use of the GPU, the algorithm would need to keep the GPU busy at all times, while reading back as little as possible.

The last condition is that a GPU cannot efficiently handle if-statements in its code, also known as branching. One of the earliest GPUs that supported the so called shader programs were very basic. Support for branching was added in a later stage and even then the execution was very crude. It simply evaluated all branches and then finally the correct branch was returned and the rest was thrown away. This technique is called "branch predication" and has the disadvantage that many execution cycles are lost because pieces of code are executed which are not necessary to determine the final result. GPUs now have better support for branching with the introduction of a technique called "dynamic branching." Dynamic branching resembles how a CPU handles branches in that it *tries* to only evaluate the necessary branches instead of all of them. It *tries*, since there are certain conditions where it is still needed to evaluate all branches. If possible, branching should be avoided since it usually comes with a performance penalty. For more details regarding GPU programming, see the survey of Owens et al. [37].

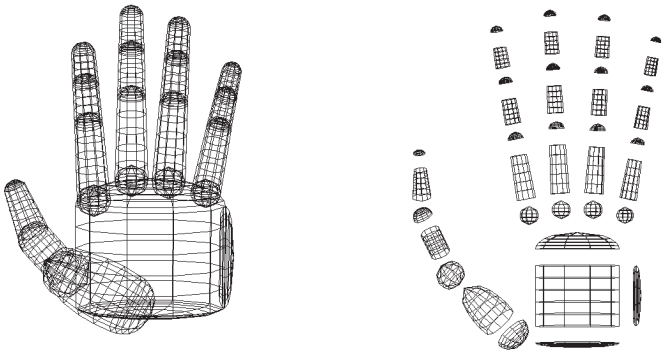
To keep the system as inexpensive as possible, regular webcams are used. However, these have a disadvantage that their reaction-time is slow compared to professional video-cameras. This makes the image very blurry when objects are moving rapidly. As explained in Section 1.4, the webcam is mounted on top of the user's head. This would make the image very blurry when the user moves his or her head and hands at the same time. This would greatly diminish the performance of algorithms that are used to retrieve the location of the hand. So instead, two additional webcams (namely the Logitech S7500) are used to supply visual information on the hand. They will be mounted onto a table and set at an angle from each other. Giving it a wide baseline to provide better stereo images.

Since a model-based algorithm tries to match the 3D model with the hand in the video-feed, it is necessary to know where the cameras are located. This location can then be used to set the virtual cameras

exactly the way the real cameras are set, so that a proper comparison can be made. The tag recognition software ARToolKit is used to supply the orientation and location of each of the cameras in the form of an OpenGL model-view matrix. This model-view matrix describes how the tag is positioned as seen from the camera. This model-view matrix can then be loaded before positioning the virtual hand. This will give it the same perspective as the real camera.

2.3 THE 3D HAND MODEL

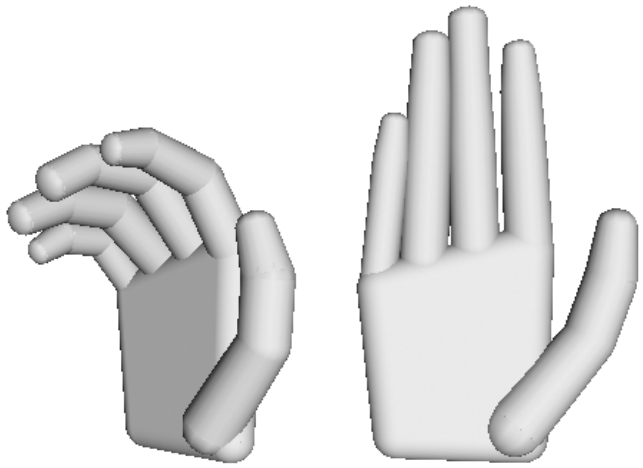
The basis of any model-based HPE algorithm is the 3D model of the hand. The 3D model used in this master thesis has been adapted from the research performed by Stenger, Mendonça, and Cipolla [44], shown in Figures 5a and 5b (further referred to as the "Stenger-model"). The reason why the Stenger-model was chosen is because it is common enough to fit most hands. Furthermore, it can be constructed in such a way that it allows for easy manual or automatic calibration. This can be done by making the size of the joints and length of the fingers variable. Apart from having variable sizes and lengths, the joints can also be rotated easily to fit different postures of the hand.



(a) Normal view of the Stenger-model. (b) Exploded view of the Stenger-model.

Figure 5: The 3D hand model of Stenger, Mendonça, and Cipolla [44].

When the Stenger-model was overlaid on a picture of a hand, it was first manually adjusted in such a way that it would fit the hand as good as possible. However, it was not possible to properly fit the thumb and the palm of the hand. The palm area of the Stenger-model has a rectangular shape when seen from the front. However, a normal hand does not have a rectangular shape, but a trapezoid shape, as can be seen in Figure 7. Also, the palm area, when seen from above, is much flatter than what is presented in the Stenger-model. Finally, the trapeziometacarpal joint of the thumb is not that thick on the outside of the hand (see 7 for joint definitions). It is shaped much straighter than the round elliptical form used in the Stenger-model. The Stenger-model was adjusted so that the model would better fit a normal hand. This resulted in the model seen in Figures 6a and 6b.



(a) Adjusted 3D hand model with its fingers bent.

(b) Front view of the adjusted 3D hand model.

Figure 6: The adjusted 3D hand model.

As previously explained in Section 1.2, a normal human hand has 29 DOFs, as can be seen in Figure 7. Instead of using all 29 DOFs, the metacarpocarpal joints are left out, bringing the total DOFs to 27 DOFs. This is something that most research regarding HPE algorithms does, since it is a relatively easy reduction step without sacrificing much accuracy [11, 13, 27, 45, 46]. Before describing the DOFs and the implemented constraints and limitations, a brief explanation will be given regarding anatomical definitions of muscle motion. All definitions were taken from [54].

- Adbuction: A motion that pulls a digit away from the midline of the hand (see Figure 8a).
- Adduction: Opposite of abduction, a motion that pulls a digit towards the midline of the hand (see Figure 8a).
- Flexion: A bending movement decreasing the angle between two parts (see Figure 8b).
- Extension: Opposite of flexion, a straightening movement increasing the angle between two parts (see Figure 8b).

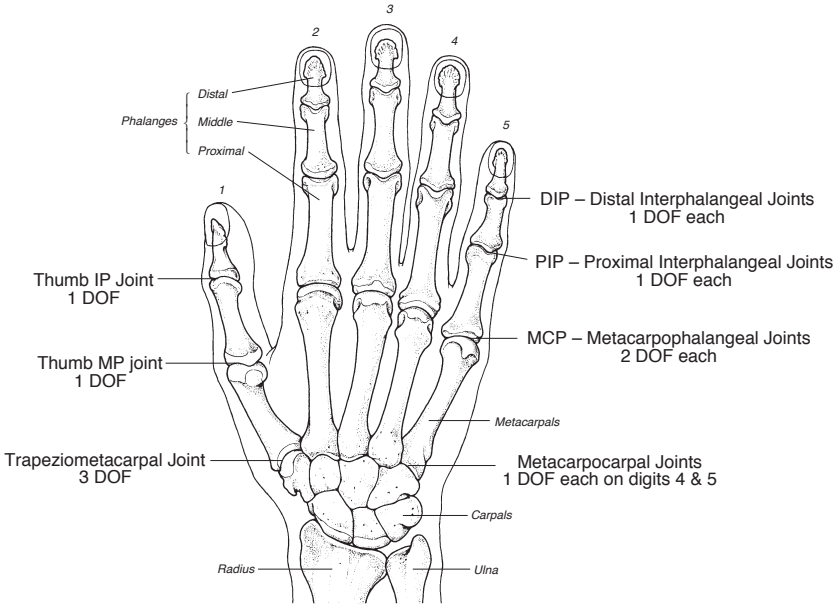


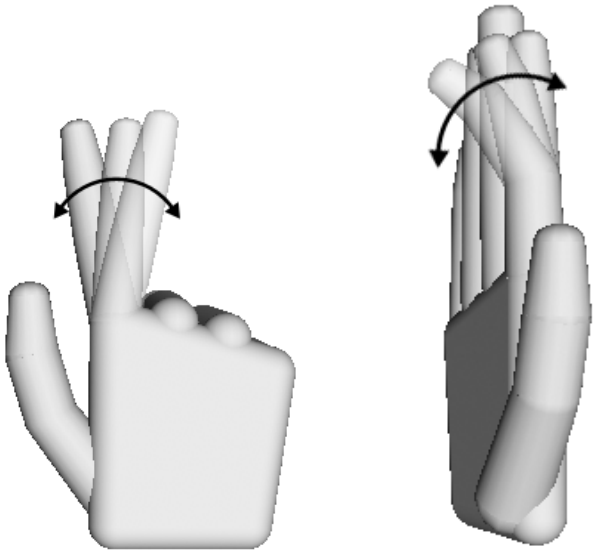
Figure 7: Bone structure of the human hand with its respective DOFs (source: Sturman [47]).

Each joint shown in Figure 7 and their associated muscle motion type is described below:

- Distal Interphalangeal joints (DIP): flexion/extension.
- Proximal Interphalangeal joints (PIP): flexion/extension.
- Metacarpophalangeal joints (MCP): flexion/extension, abduction/adduction.
- Thumb Interphalangeal joint (IP): flexion/extension.
- Thumb Metacarpophalangeal joint (MP): flexion/extension.
- Trapeziometacarpal joint (TMC): flexion/extension, abduction/adduction, twist.

Rijpkema and Girard observed that replacing the twist DOF of the TMC joint of the thumb by an abduction/adduction DOF at the MP joint resulted in a more workable model [40]. Therefore, the model in this thesis also adapts this convention. Apart from this difference and the removal of the DOFs located at the metacarpocarpal joints, the 3D model and the DOFs of the human hand are the same.

In an effort to decrease the dimensionality of the hand, research by Chua, Guan, and Ho has shown that the human hand is highly constrained [11]. They were able to bring down the number of DOFs to 12 without sacrificing too much accuracy. This is possible because the movement of the fingers in the human hand are inter-dependent. The constraints presented in the research is implemented in the adapted 3D hand model of this thesis, to decrease the search space of the HPE



(a) Abduction and adduction muscle motion.

(b) Flexion and extension muscle motion.

Figure 8: Anatomical definitions of muscle motion.

algorithm. Apart from implementing constraints, the fingers also have a specific range at which they can bend [27]. This also limits the search space by eliminating impossible movements. It should however be noted that these limits are based on natural finger motion. It is still possible to bend a finger in a particular way that is not possible to do using the muscles of the finger alone. Limiting the finger motions is valid since the application of this thesis expects the user not to perform such "artificial" movements.

Chua, Guan, and Ho group constraints together by weak and strong constraints. The difference between the two is that weak constraints assume a particular initial factor between two DOFs, but this factor might be different depending on the posture and person. Strong constraints should always hold for natural finger motion.

Constraint 1

The first strong constraint is proposed by Rijpkema and Girard [40]. The relationship between angles of the proximal interphalangeal (PIP) and distal interphalangeal (DIP) joints are as follows:

$$\mathrm{DIP}_{fe} = \tfrac{2}{3}\mathrm{PIP}_{fe} \tag{2.1}$$

Where fe refers to the flexion/extension DOF. With this constraint the number of DOFs decrease from four to three per finger.

Constraint 2

The next strong constraint is also proposed by Rijpkema and Girard [40]. From experimental observation it was possible to deduce the following dependency between the TMC and MP thumb joint:

$$TMC_{fe} = 2(MP_{fe} - \tfrac{1}{6}\pi) \tag{2.2}$$

Constraint 3

Experimental data obtained by Rijpkema and Girard [40] also showed that there was another dependency between the TMC and MP thumb joint:

$$TMC_{aa} = \tfrac{7}{5}MP_{aa} \tag{2.3}$$

Where aa refers to the abduction/adduction DOF. With this constraint it is now possible to describe all DOFs of the thumb using only three DOFs instead of five.

Constraint 4

Lee and Kunii observed that there was little abduction and adduction in the MCP joint of the middle finger [25]. Therefore, it is possible to define the following constraint for the MCP joint of the middle finger:

$$MCP_{aa} = 0 \tag{2.4}$$

As explained before, the constraints are based on natural finger motion. Even though this constraint restricts a movement that is possible to do using your normal finger muscles, it is normally not used. It is therefore valid to say this movement would not occur during the use of ARMI.

Constraint 5

The next weak constraint is proposed by Kuch and Huang [23]. The MCP and PIP joints have a dependency represented by the following equation:

$$MCP_{fe} = k \times PIP_{fe} \qquad 0 \leqslant k \leqslant \tfrac{1}{2} \tag{2.5}$$

The initial value that was used in the research of Kuch and Huang for k is $\frac{1}{2}$. If this happens to deliver high errors between the model and the image, it will be adjusted downwards until a satisfactory result is found.

Constraint 6

The following constraint is also a weak constraint, proposed by Chua, Guan, and Ho [11]. It describes the dependency between the DIP and MP joint of the thumb:

$$IP_{fe} = a \times MP_{aa} \qquad a \geqslant 0 \tag{2.6}$$

Limitations thumb

The following limitations for the thumb are described in the research by Lien [27]:

$$0^\circ \leqslant MP_{fe} \leqslant 45^\circ \tag{2.7}$$

$$0^\circ \leqslant IP_{fe} \leqslant 90^\circ \tag{2.8}$$

Limitation fingers

The next limitations for the four fingers are taken from the research by Lin, Wu, and Huang [28]:

$$0^\circ \leqslant MCP_{fe} \leqslant 90^\circ \tag{2.9}$$

$$0^\circ \leqslant PIP_{fe} \leqslant 110^\circ \tag{2.10}$$

$$0^\circ \leqslant DIP_{fe} \leqslant 90^\circ \tag{2.11}$$

$$-15^\circ \leqslant MCP_{aa} \leqslant 15^\circ \tag{2.12}$$

Limitation 2.12 does not apply to the middle finger, since the MCP_{aa} of the middle finger is set to zero in Constraint 2.4.

The hand model now consists of nine DOFs and five weak constraints in total. Two DOFs for each finger and the thumb, except the middle finger which has one DOF. Each finger and the thumb also have one weak constraint. All fingers have also been limited in their movement which should severely decrease the search space.

2.4 GENERAL SYSTEM OVERVIEW

In Section 2.2 it was explained that a model-based algorithm seems to be the best approach to take. This approach renders several different

configurations of the 3D hand model and determines which of these models is the best fit to the hand seen in the video feed. This clearly needs a way of comparing the different configurations of the hand model with the original hand. Edge information is usually used to compare the 3D model with the video feed [45]. The edges can be compared with each other to establish a form of error measurement. The advantage of such a technique is that the algorithm does not need to know which part of the hand it actually sees. A finger is the same to the algorithm as a piece of the palm. This avoids requiring the user to label their fingers with markers or require other knowledge of the users' hand. Apart from the previous advantage, edge enhancement can also be easily mapped to GPU hardware which makes it a perfect candidate to use in the HPE algorithm.

Since the system needs to be able to track the hand of the user in real-time, it cannot simply render millions of different configurations. Even though this would probably result in a near-perfect match, there is no ordinary computer at this time that can perform that many calculations and still achieve real-time performance. So some form of search algorithm is needed that can search through the twelve DOFs and find the optimal configuration with a minimal amount of renders.

With the basic idea in mind of how a model-based algorithm should work, a system was designed to take the following steps:

1. Receive data from the hand tracking algorithm.
2. Perform edge enhancement on the video frames.
3. Adjust the 3D hand model using a multidimensional search algorithm.
4. Perform edge enhancement on the 3D hand model.
5. Determine the error of the 3D hand model by subtracting the edges of the 3D model from the edges of the hand in the video-feed.
6. Repeat from step 3, if time allows, or stop if the error is acceptably low.
7. Return the 3D hand model with the smallest error.

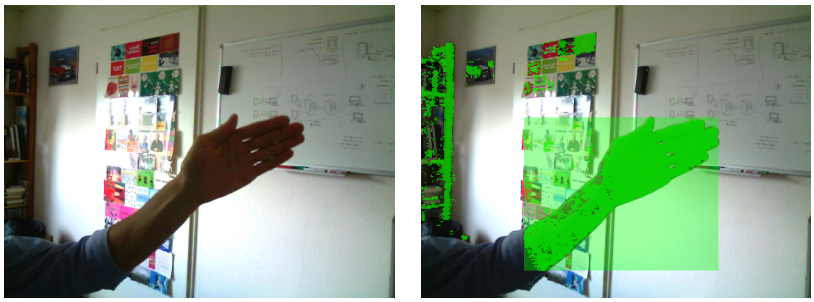
The system is designed as an iterative approach, constantly trying to find a better match. This happens between step 3 and 6. The system will enter the last step if it has found a sufficiently matching configuration, or when there is no more time left. After the last step the system returns the configuration of the model with the smallest error to the GUI part of the ARMI system. The GUI updates the position of the 3D hand and then renders the hand so that the user can see the pose and position of the hand. This will also give visual feedback whether the HPE algorithm performs as it should.

The next sections of this chapter will discuss all theoretical details of the system step by step.

2.5 HAND TRACKING ALGORITHM

To estimate the hand pose of the user, the system first needs to see the hand. This is done using two cameras mounted onto the table. The video feeds from these cameras are fed through a hand tracking algorithm which delivers the input for the HPE algorithm. The input received from the hand tracking algorithm consists of the following data:

- A frame from each of the two cameras. Each frame consists of a texture of four color channels. Three channels are used to hold the red, green, and blue color components and the fourth is used to indicate whether a pixel is classified as a hand pixel or not.
- Coordinates and size of one or more bounding boxes that indicate where the hand is in each frame.



(a) Original camera feed.

(b) Camera feed added with hand tracker information.

Figure 9: The hand tracker output (source: thesis Fremouw [15]).

An example of the input received from the hand tracking algorithm is shown in Figures 9a and 9b. Figure 9a shows the basic red, green, and blue color channels. Figure 9b shows all color channels including the fourth channel (shown in bright green), to indicate which pixels are classified as hand pixels. Also, the bounding box is drawn around the hand (shown in transparent green). The HPE algorithm uses this information to know which pixels it should process. For more details regarding the hand tracking algorithm, see the master thesis of Fremouw [15].

2.6 EDGE ENHANCEMENT

The second and fourth step of the system applies an edge enhancement algorithm on the video frames and the 3D model. The edges of the video frames and the 3D model are compared to each other to measure

the error between them. The edge enhancement algorithm that is used in this thesis is the Sobel operator [16] since it is fast and relatively easy to implement. The Sobel operator uses two 3x3 kernels (shown in Equations 2.13 and 2.14) which are convolved with the original image to calculate approximations of the horizontal and vertical derivatives. Combining both the horizontal and vertical derivatives results in an image where the edges are enhanced. In mathematical terms, the Sobel operator can be expressed using the following equations:

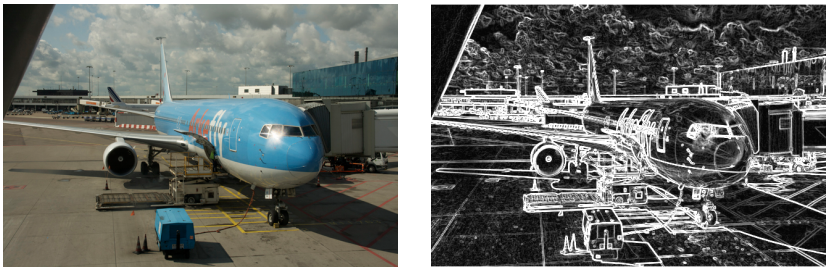
$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \tag{2.13}$$

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \tag{2.14}$$

$$G = \sqrt{{G_y}^2 + {G_x}^2} \tag{2.15}$$

Where $*$ denotes the 2-dimensional convolution operation. Also, A represents the original image and, G_y and G_x represent images that contain the horizontal and vertical derivative of A. The final edge enhanced image is denoted by G in the last Equation 2.15.

The result of the Sobel operator applied to an image can be seen in Figures 10a and 10b.



(a) Original picture of an airplane. (b) Sobel operator applied to the picture.

Figure 10: An example of the Sobel operator applied to an image.

Since edge enhancement is applied to every 3D model and each video frame, it should require as few calculations as possible. This is the main reason the Sobel operator is chosen as the edge enhancement algorithm. It is relatively inexpensive in terms of computations when compared to, for instance, the Canny edge detection operator.

2.7 ADJUST THE HAND MODEL TO FIND THE BEST FIT

At every iteration, the configuration of the model is adjusted to try and find a better fit. This search is guided by an optimization algorithm that can search the multidimensional space in which all possible configurations exist. Many optimization algorithms exist, but many also do not go along well with the specific circumstances that are present in the system. Most algorithms, like the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [36], require the calculation of derivatives. While it is possible to numerically estimate these derivatives, they are also very expensive to calculate time wise. Since for each calculation the entire process of rendering, edge enhancement, and error determination has to be done. Therefore, the optimization algorithm should not rely on derivatives since they would be unable to achieve real-time performance.

The optimization algorithms that are available are narrowed down to the following:

- genetic algorithms;
- neural networks;
- Nelder-Mead method;
- simulated annealing.

From these different types of algorithms, Nelder-Mead method (NM) and simulated annealing (SA) are chosen. Both have been used before in motion tracking research so it is assumed they provide a good starting point [1, 29]. Also, as a comparison with an algorithm that uses derivatives, the Secant method is chosen. It was chosen because it requires very little derivatives when compared to other derivative-based algorithms that were reviewed like BFGS. This way it might have the possibility to outperform NM and SA. The following sections will now discuss all the theoretical details of each algorithm.

2.7.1 *Secant method*

The Secant method is named after the way it operates. It uses secant lines (any line that intersects two points on a curve) to find better approximations of the root of the target function. This can be seen in Figure 11. The pure form of the Secant method equation is 2.16:

$$x_{n+1} = x_n - \frac{x_n - x_{n\text{-}1}}{f(x_n) - f(x_{n\text{-}1})} f(x_n) \tag{2.16}$$

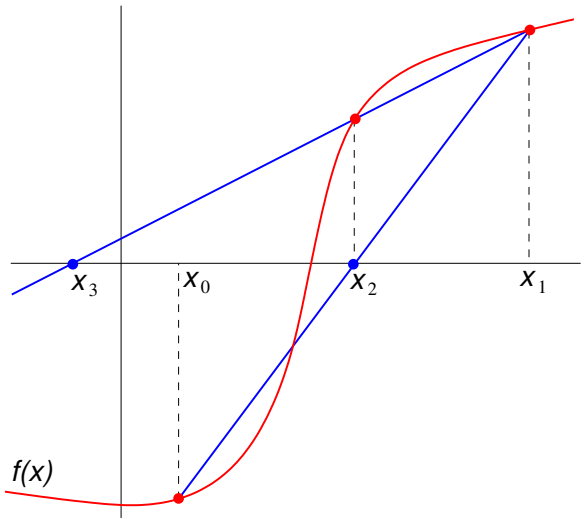


Figure 11: An example of the first two steps of the Secant method (source: Jitse Niesen [35]).

The method needs two starting positions, x_0 and x_1, as can be seen in equation 2.16. These starting points should ideally be chosen close to the root of the function.

The Secant method is designed to operate on one-dimensional data. However, since the problem at hand is a multidimensional problem, it requires adjustments to the original equation. Even though there are other algorithms, like Broyden's method [9] or BFGS [36], that extend the Secant method to support multiple dimensions, it is determined that this requires the calculation of too much derivatives. Instead, the method will be adjusted to do one dimensional steps in each dimension of the search space. This way the number of derivatives that have to be calculated will be kept at a minimum, while still maintaining the original operation of the algorithm.

One of the biggest problems of the Secant method is that, when the data lies on a flat plane, it will overshoot or jump to infinity very quickly. To solve this, the maximum step size the method is allowed to do should be restricted to a certain value. What this value should be in the case of this project is determined in Chapter 4.

2.7.2 *Nelder-Mead method*

The Nelder-Mead method or downhill simplex method is a greedy method that was originally proposed by Nelder and Mead [34]. It is able to minimize an objective function in a multidimensional search space without the need for calculating derivatives. To do this, the method uses a multidimensional shape called a simplex. A simplex is a multidimensional generalization of a triangle (2D) or a tetrahedron (3D). A simplex is chosen as a starting point and with each iteration it moves through the search space in a predefined manner. At the end of each iteration it replaces its worst vertex with a vertex that is better than any

of its other vertices. This new vertex is found using a set of predefined steps. A visual representation of these steps, in a 3 dimensional search space, is presented in Figure 12.

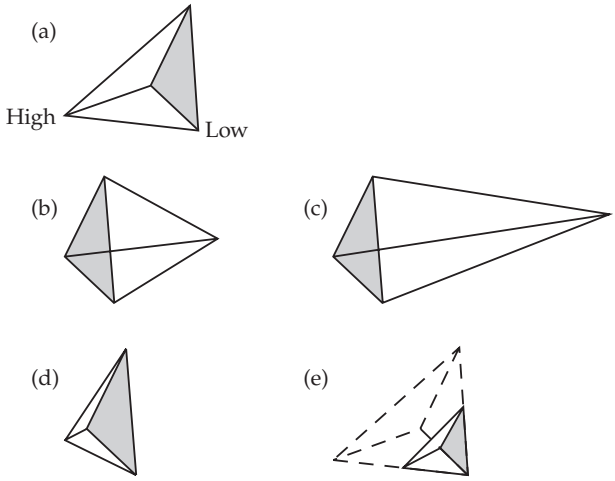


Figure 12: A visual representation of all possible steps of the Nelder-Mead method. In each iteration of the method the simplex (a), displayed as a tetrahedron here, can either be reflected (b), reflected and expanded (c), contracted in one dimension (d), or contracted in all dimensions towards the best or "low" vertex (source: Numerical recipes in C [38]).

Mathematically the steps are defined as follows [34]:

- Order all vertices according to the values at each vertex (see Figure 12-a):

$$f(\mathbf{x}_1) \leqslant f(\mathbf{x}_2) \leqslant \cdots \leqslant f(\mathbf{x}_{n+1}) \tag{2.17}$$

- Calculate the center of gravity ($\mathbf{x}_0$) of the simplex without using the worst vertex ($\mathbf{x}_{n+1}$):

$$\mathbf{x}_0 = 0.5 * \sum_{i=0}^{i=n} \mathbf{x}_i \tag{2.18}$$

- Calculate the reflected vertex (see Figure 12-b):

$$\mathbf{x}_r = \mathbf{x}_0 + \alpha(\mathbf{x}_0 - \mathbf{x}_{n+1}) \tag{2.19}$$

  Where α represents the reflection coefficient, with a default and minimum value of 1.

  Now the next step of the iteration will be determined by evaluating the value of the reflected vertex compared to the other vertices:

  - if $f(\mathbf{x}_1) \leqslant f(\mathbf{x}_r) < f(\mathbf{x}_n)$ then replace $\mathbf{x}_{n+1}$ with $\mathbf{x}_r$ and go to the next iteration.

- if $f(\mathbf{x}_r) < f(\mathbf{x}_1)$ then calculate the expanded vertex
- else calculate the contracted vertex

If the first condition holds, then the simplex most probably hit the other side of a valley or it might go down a slope. But since the reflected vertex is not better than the best vertex, it is safe to assume that the slope or valley does not proceed to its minimum in the direction of the reflected vertex. There is therefore no need to look any further so the method can continue to its next iteration. If, however, the second condition holds, then the simplex lies on a slope that is moving down in the direction of the reflected vertex. It is worthwhile to see if and how far the slope continues downhill. This will be checked in the step where the expanded vertex will be calculated. If neither of the conditions were met, then the simplex is assumed to be in a sink or a valley and a better vertex would then only be present inside the simplex. So the method continues to calculate the contracted vertex.

- Determine the expanded vertex (see Figure 12-c):

$$\mathbf{x}_e = \mathbf{x}_0 + \gamma(\mathbf{x}_0 - \mathbf{x}_{n+1}) \tag{2.20}$$

With γ denoting the expansion coefficient, with a default value of 2 (always larger than α).

Now the following case will be evaluated and afterwards the next iteration will start:

$$\mathbf{x}_{n+1} = \begin{cases} \mathbf{x}_e, & \text{if } f(\mathbf{x}_e) < f(\mathbf{x}_r) \\ \mathbf{x}_r, & \text{else} \end{cases} \tag{2.21}$$

In the case that the expanded vertex is better than the reflected vertex, then it is probable that the simplex is on a slope that continues down in the direction of the new vertex. Since the expanded vertex is now chosen to be part of the new simplex, it becomes possible for the method to traverse the slope much quicker, since the simplex is larger. In any other case, the expanded vertex most probably hit the other side of a valley. In this case a small step is taken down and afterwards the next iteration is started.

- Determine the contracted vertex (see Figure 12-d):

$$\mathbf{x}_c = \mathbf{x}_{n+1} + \rho(\mathbf{x}_0 - \mathbf{x}_{n+1}) \tag{2.22}$$

With ρ denoting the contraction coefficient which lies between 0 and 1 with a default value of 0.5.

If the contracted vertex is better than the worst vertex ($f(\mathbf{x}_c) \leqslant f(\mathbf{x}_{n+1})$), then the worst vertex is replaced by the contracted vertex. Afterwards, the method continues to the next iteration. In all other cases, it is assumed the simplex is inside a sink and the simplex will be shrunk or reduced by calculating its reduced vertices.

- Replace all vertices by the reduced vertices (see Figure 12-e):

$$\mathbf{x}_i = \mathbf{x}_1 - \sigma(\mathbf{x}_i - \mathbf{x}_1) \quad \text{where } i \in \{2, \ldots, n+1\} \tag{2.23}$$

  With σ representing the reduction coefficient which lies between 0 and 1 with a default value of 0.5.

When these set of rules are followed, the method is guaranteed to find a minimum [34]. The problem of the Nelder-Mead method, as with other optimization methods, is that it will usually find a local minimum instead of the global minimum. This can partially be overcome by choosing the correct size for the starting simplex so that local minima are skipped. Another possibility is to choose multiple starting simplices. After each simplex converges to a certain point, the best vertex can be chosen as a true minimum of the objective function. Both possibilities will be investigated in Chapter 4.

2.7.3 *Simulated Annealing*

Simulated annealing (SA) is a probabilistic method that tries to find a close approximation to a global minimum of the objective function. It was originally proposed by Scott Kirkpatrick, C. Daniel Gelatt and Mario P. Vecchi [21], and by Vlado Černý [52] and it is derived from "annealing", a technique used in the field of metallurgy. Annealing involves melting a material and then cooling it slowly to increase the size and amount of crystals and decrease defects in the material. Heating a material causes the atoms to become unstuck and wander around randomly. When the material is then cooled down slowly, it becomes possible for the atoms to find a configuration with lower internal energy than the original configuration.

SA simulates the process by making a random move each iteration. Either the random move results in a better configuration than the original and the new move is accepted, or the random move is accepted with a certain probability. This probability is tied to a virtual temperature that is lowered during the iteration process. This causes the method not to get stuck in local minima but instead has a chance to find the global minimum. The probability of SA finding the global minimum of a finite problem approaches 1, given enough time [17].

In Listing 1 the SA process is depicted using pseudo code. Notice that the pseudo code does not contain code to decrease the temperature. This is explicitly left out since there are many ways of doing this. Also, how a new random neighbor is chosen is not specified since this is an application-specific problem. Each application demands a different "neighbor-generator."

SA is hardly ever implemented in its pure form since there are several disadvantages to it. First, the final configuration that it finds might not be the best that it has found during its complete process. Even though this would be impossible to do with physical annealing, a possibility

would be to store the best configuration it has come across during its entire process. This would be a simple solution if it is possible to store the configuration without too much performance loss. Another disadvantage is that sometimes the configuration drifts away from the minima if the temperature is too high, since it mostly or only accepts the random neighbors during that phase. A possible solution to this problem is to restart the annealing process to its previously found best configuration.

```
x = initial configuration

while p < maximum iterations:
  i = random neighbor

  if f(move(x, i)) is better than f(x):
    x = move(x, i)
  else accept new move with a certain probability:
    x = move(x, i)

  p = p + 1
```

Listing 1: The simulated annealing process in pseudo code.

During the implementation phase a good neighbor function will be designed, as well as good working cooling schedule. This will be discussed in Chapter 4.

2.8 DETERMINING THE ERROR

During each iteration of any of the search algorithms discussed in Section 2.7, the algorithms need to know how good or bad the fit is with the object seen in the video frame. To do this, an error rate is established that uses a number between 0 and 1, where 0 indicates a perfect fit and 1 is no fit at all. The error rate is determined in step five of the HPE algorithm (see Section 2.4 for an overview of all steps). The edge enhanced video image and edge enhanced model from step two and four are used to determine the final error rate. In essence, the two edge enhanced images are subtracted from each other, leaving only the parts that do not match the original object from the video frame. The leftovers can then be counted and divided by the total number of pixels the original object has in the video frame, resulting in a scale that describes how good the model fits the original object.

2.9 SUMMARY

The requirements are set for the project. After analysis, it became clear that the best approach for the algorithm is a model-based approach. This approach tries to match the hand model onto the hand in the video frame. Since a model-based approach makes use of images, it became possible to use the GPU. The GPU provides more raw processing power

than a normal CPU so this is beneficial to the performance of the algorithm.

The hand model is made with all its 27 DOFs. The movement of the joints is constraint so that they can only perform natural hand movement, resulting in a decrease of DOFs to 9. Finally, all the seven steps of the algorithm are described. The search algorithms that are mentioned will be tested in the next chapter.

3

IMPLEMENTATION

This chapter will discuss all implementation details regarding the entire HPE algorithm. The choice for the programming language will be explained amongst other details. A general system overview will be given, to show how the system interacts with each individual part of the total project. The chapter will also discuss all drawbacks of the search algorithms as well as present possible solutions to the problems. The applied solutions will be tested and evaluated in the next chapter.

3.1 PROGRAMMING LANGUAGE DECISION

The HPE algorithm should be able to interface with other parts of the project since it needs them to function properly. For this reason, the programming language was a decision for the entire project rather then for the HPE algorithm alone. This resulted that the choice for the programming language was not only based on the requirements of the HPE algorithm itself, but also on the requirements of the entire project. Some requirements are therefore specific to the HPE algorithm, while others are global requirements that affect the entire project.

It was determined that the programming language should be able to:

- draw and render vertex data to render the objects, the 3D hand, and the interface;
- perform calculations on the video card to take advantage of the GPU as a fast processing unit;
- run relatively fast since the system needs to deliver updates of the 3D model every 1/15th of a second;
- run on multiple platforms without any change to the code because the project members run their code on various platforms.

Using these requirements, the project group came to the following conclusions:

- The 3D hand model and interface will be built and rendered using OpenGL. OpenGL is a widely used API to do graphics-related work. It can run on a range of different platforms such as Windows, Linux, and Mac OS X.

- The programming language needs OpenGL-bindings to make use of OpenGL. Most popular programming languages have these OpenGL-bindings, but there is one in particular that is very useful in this case, namely the programming language called "Python". Python is a high-level scripting language and is known for its ease of use. Since it is a high-level language, the programmer does not have to deal with low level operations like memory allocations. This will severely shorten the time to complete the final product since the programmer can focus on the logic instead of having to deal with memory leaks and the like. An added bonus is that Python is platform independent. It runs on a wide range of platforms. Much of the code the programmer writes, does not have to be changed in order to run on for instance Windows, Mac OS X, or Linux. It also supports easy integration with other programming languages.

- Since Python is a high level language, it is also relatively slow and memory inefficient when compared to C or C++. The parts that require high performance will therefore be implemented in C instead of Python. This would not put any extra work during the implementation phase, since Python has special facilities which eases integration with C.

- Finally, the OpenGL Shading Language, or GLSL, is used to program the GPU. It has an interface with OpenGL and can execute commands on video cards of multiple vendors without making any change to the code. The GLSL programs, or shaders, can run on any video card as long as the video driver supports GLSL.

After the programming languages were decided, it was time to see how each individual component will interact with the rest of the system. This will be discussed in the next section.

3.2 SYSTEM OVERVIEW

Each component of the systems sends and receives data from other components. What this data is and how it is communicated between each component is shown in Figure 13.

Starting from the beginning, the hand tracker receives frames from the webcams. For each webcam there is exactly one hand tracker. The hand trackers are synchronized with each other so that each frame is retrieved at the exact same moment. This is done so that the frames can be properly compared with each other. It is important to know that the hand tracker (or any other program) *asks* for the frame from the webcam driver. It does not determine when the actual frame is taken. In Section 3.4 it will be shown why this makes so much difference. After the hand tracker receives the frame from the webcam, it detects the hand in the frame and relays all information regarding the hand to the HPE algorithm.

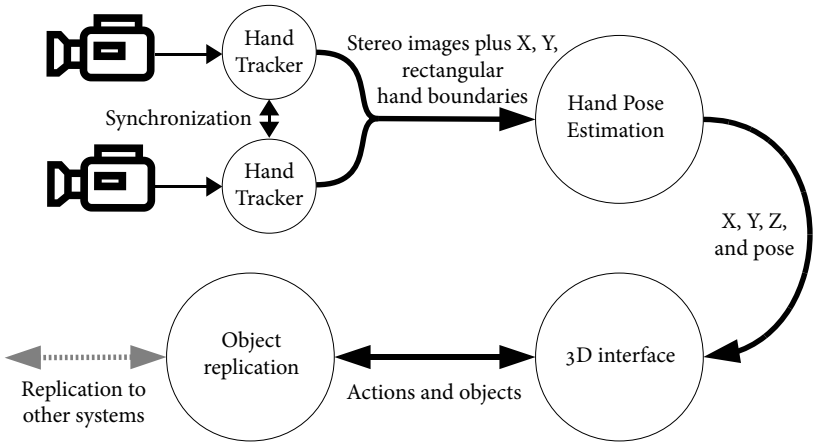


Figure 13: Global overview showing the information flows between all the components (original source: master thesis Fremouw [15]).

The HPE algorithm uses both frames and all the information received from the hand tracker to determine the correct DOFs of the hand model, including its position. All the DOFs of the model are transfered to the 3D interface. The 3D interface draws the interface along with all the objects, including the 3D hand model. The hand model is drawn using the new DOFs received from the HPE algorithm. This interface is shown to the user using an HMD (see Figure 2 for a picture of an HMD).

Knowing where the hand model is located allows the 3D interface to know which actions it should take. A user is able to manipulate a 3D object using the manipulation menu found inside the 3D interface. When a user wants to access this manipulation menu, he or she puts his or her hand inside a 3D object. The manipulation menu for that 3D object will now pop up once the user's thumb and index finger touches each other inside the object. Rotation, scaling and moving the object is all part of the manipulation menu. For more information on the 3D interface and all the possible user interactions, the reader is referred to the master thesis of Bruining [10].

So far all the information between the components is transferred inside one computer. Since the goal of project ARMI is to allow multiple users to interact with the same shared environment (see Section 1.4), it should somehow send all interaction information to other clients. This is what the object replication component is for. It makes sure that the changes occurring inside the virtual world are transferred to other computers connected to the same environment. This allows multiple users to interact with the same environment without having to be physically near each other. Apart from seeing your own virtual hand, the object replication also transfers information regarding the hands of the other users. This way each user sees the hands of the other users, allowing each user to point at objects to let others know what he or she is talking about. The reader is referred to the master thesis of Lenting [26] for details on how the object replication component manages to transfer all data while maintaining a consistent state on all clients.

Combining all components allows a user to perform interactions in a virtual environment with his or her bare hands. Also, other users are able to connect to the same environment and see what others are doing. This gives multiple users the ability to accomplish a task together.

3.3 HPE ALGORITHM IMPLEMENTATION

In Section 2.4 all the steps and theoretical details of the HPE algorithm were explained. In this section it will be explained how each step is implemented and what kind of practical problems there were. As a reminder, here are the steps as presented in Section 2.4:

1. Receive data from the hand tracking algorithm.
2. Perform edge enhancement on the video frames.
3. Adjust the 3D hand model using a multidimensional search algorithm.
4. Perform edge enhancement on the 3D hand model.
5. Determine the error of the 3D hand model by subtracting the edges of the 3D model from the edges of the hand in the video-feed.
6. Repeat from step 3, if time allows, or stop if the error is acceptably low.
7. Return the 3D hand model with the smallest error.

Each step had its problems, most of which will be explained in the following sections. Apart from the problems that were encountered, there were also other details to work out during implementation. Since one of the requirements is that the system should be able to deliver real-time updates, much time is spent on trying to find the fastest implementation for each particular problem. Solutions that did not make it to the final implementation are mentioned as possible improvements in Chapter 6.

3.4 RECEIVE DATA FROM THE HAND TRACKING ALGORITHM

The HPE algorithm receives two sets of data from the hand tracking algorithm, as previously explained in Section 2.5. First, the frames of the video cameras in which all the visual and hand recognition data is stored. And second, an array containing the coordinates and sizes of bounding boxes that surround the object in the frame. The frames are delivered in such a way that they can be loaded directly into an OpenGL texture without any change to the data.

Listing 2 shows how the hand tracker is used inside the program. Lines 2 and 3 show how the hand tracker object is created. The histograms that are given to the hand tracker are trained on the color of the object and on the background of the scene. This gives the hand tracker the ability to distinguish between the background and the object itself. For more details about the histograms, see the master thesis of Fremouw [15]. Between lines 15 and 17, the frames are copied from the hand tracker arrays to textures so that the HPE algorithm can work with them. Line 23 is where the HPE algorithm is invoked. It continues on the next cycle of the while-loop after it is finished executing. Important to note is that the hand tracker makes sure that the frames per second (FPS) are maintained at a steady 15 FPS. It does this by blocking the get_image function call, at line 12 and 13, until a total of 1/15th of a second has passed between the former call and the current call. It does not block if more time has passed since this would mean that the system waits while nothing actually happens. Using this method, it is possible to sustain a preset FPS, in this case 15 FPS. If more time is needed to process a frame, the FPS would drop below 15 FPS, but it will never go over 15 FPS since the call to the hand tracker would block until 1/15th of a second has passed.

```
1  # initialize both hand trackers
2  left_tracker  = handtracker.new(cameraID_0, width, height,
       framerate, object_histogram, non_object_histogram)
3  right_tracker = handtracker.new(cameraID_1, width, height,
       framerate, object_histogram, non_object_histogram)
4  right_tracker.keep_synchronized_with(left_tracker)
5
6  # start trackers with object tracking enabled
7  right_tracker.start(True)
8  left_tracker.start(True)
9
10 # continuously read frames and process them
11 while True:
12   left_camera_img , left_blobs  = left_tracker.get_image()
13   right_camera_img, right_blobs = right_tracker.get_image()
14
15   # ...
16   # copy frames into OpenGL controlled memory
17   # ...
18
19   right_tracker.release_image()
20   left_tracker.release_image()
21
22   # ...
23   # HPE algorithm
24   # ...
```

Listing 2: Using the hand tracker.

By copying the frames into OpenGL memory, it becomes possible to work with the frame using shaders. Important to note is that the coordinate systems of OpenGL and the hand tracker differ from each other. In OpenGL (in 2D drawing mode) the origin is in the top left corner whereas the origin of the frame of the hand tracker lies in the bottom left corner. It is relatively easy to correct this problem since OpenGL allows the programmer to specify how a texture is drawn on

the screen. This is done by specifying that the bottom left corner should be drawn in the top left corner, using the code from Listing 3.

```
glBegin(GL_QUADS)
glTexCoord2f(0.0, tex_height)
glVertex2f(0.0, 0.0)
glTexCoord2f(tex_width, tex_height)
glVertex2f(screen_width, 0.0)
glTexCoord2f(tex_width, 0.0)
glVertex2f(screen_width, screen_height)
glTexCoord2f(0.0, 0.0)
glVertex2f(0.0, screen_height)
glEnd()
```

Listing 3: Drawing the frame texture.

The difference in coordinate system also caused trouble with the location of the bounding boxes that surrounds the object inside the frame. The bounding boxes are used inside the edge enhance shader so that pixels are not processed that are not inside the bounding boxes. Therefore, the coordinates of the bounding boxes also had to be recalculated to match the coordinate system of OpenGL. Another problem with the bounding boxes is that they contain an X and Y starting point and the size of the box in X and Y direction. However, in OpenGL the coordinates are specified in absolute coordinates. So these coordinates had to be rewritten into the form (x1, y1), (x2, y2), representing the top-left and right-bottom corner respectively.

Even though the hand tracker performs very well, it does not detect the object in each and every frame. It would sometimes give the HPE algorithm an empty list of bounding boxes, even if the object itself is present inside the frame. The alpha channel of the frame is still filled with the information whether a pixel is part of the object or not. So this information is still present even though the algorithm does not receive any bounding box coordinates. Since the pixel information can still be used, it was decided that if there is no bounding box, the algorithm sets a bounding box across the entire frame. This way all the pixels are processed which results in a decrease in performance, but it will allow the algorithm to detect the object.

As mentioned before, the hand tracker *asks* from the webcam driver for the frame. It does not determine when the actual frame is taken. Initially this was not thought to be a problem since the difference would be very small. However, during preliminary tests it became clear that the difference between some frames are much larger than expected. This poses a problem since the HPE algorithm can never know whether both frames are taken at the exact same moment and it has no way of compensating for the unknown time difference.

A test is performed to determine what the difference between each frame is on both Ubuntu-Linux and Mac OS X and whether there is a possible solution for it (see Appendix A for the specifications of the test systems). The reason for testing both operating systems is that it is presumed they have different schedulers and thereby give different results. The test consists of recording 100 stereo-frames of the screen of the system that is tested. On the screen a program runs, printing out an

increasing number every 1/15th of a second. The program remembers when each number is printed on the screen. The time difference between the frames of both cameras is determined by looking up the numbers shown in the frames, with the time that the numbers were printed. When several test frames were analyzed, several numbers showed up multiple times in the same frame. Since each number is unique and never printed twice, it showed that there even was a difference between the top line of the frame and the bottom line. The camera scans from bottom to top, from left to right. So while the frame itself is captured, time progresses which causes that some numbers are shown more than once in the same frame. The cameras were turned on their sides to remedy this problem, to make sure that the correct numbers are read from the frames. The FPS of the hand tracker itself are also analyzed, since it seemed to fluctuate up and down during preliminary testing. The results of the test can be seen in Graphs 14 and 15.

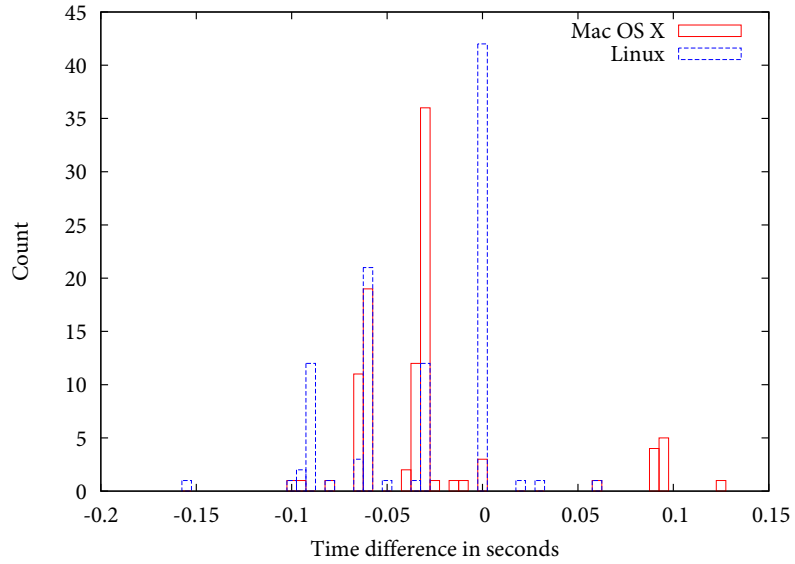


Figure 14: Time difference between the left and right frame received from the hand tracker.

Graph 14 clearly shows that Linux performs better than Mac OS X since it manages to retrieve more stereo-frames at the exact same moment. However, the average time difference between both frames are more or less the same on both operating systems. Mac OS X achieves an average of -27 milliseconds while Linux has an average of -34 milliseconds between the left and right frame. Unfortunately, it is not possible to change when the actual frame is taken using the cameras that are used for the project. A possible solution to provide a better synchronization between both cameras is to use a frame grabber. A frame grabber is a device that sits between the computer and the cameras. It simultaneously requests frames from all the cameras that are connected to it and afterwards sends all frames to the computer. An example of such a frame grabber is the Sensoray 2255 which can be seen in Figure 16. Normal USB webcams cannot be connected to this frame grabber. Only more expensive cameras that provide PAL or NTSC output through a BNC cable can be used. Another solution that can be used in conjunction with a frame grabber is to use cameras that support an external synchronization signal. An extra digital or analog master signal is used

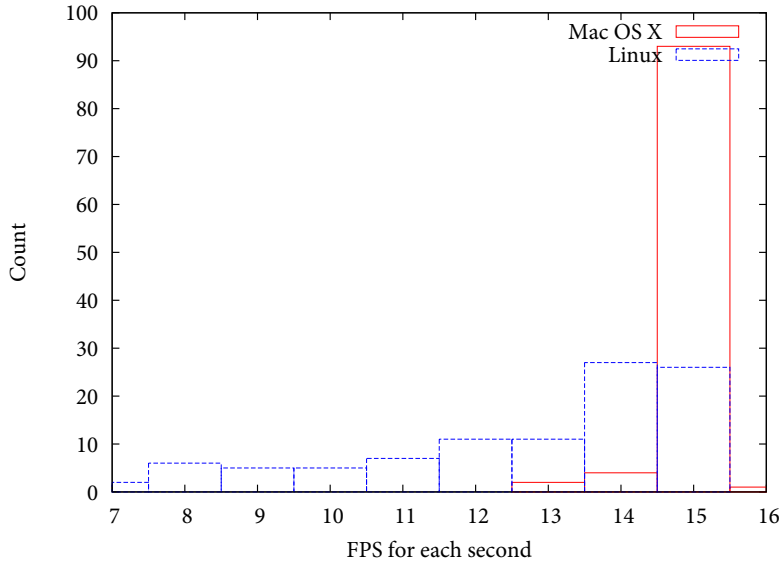


Figure 15: Average FPS of the second before each stereo-frame is taken.

to synchronize all the cameras with one master synchronization signal. Both systems can be used to provide a solution to the time difference between the stereo frames, however one of the requirements was that the system would be built using cheap COTS hardware. It was therefore decided that the system would use the webcams instead of more expensive solutions.

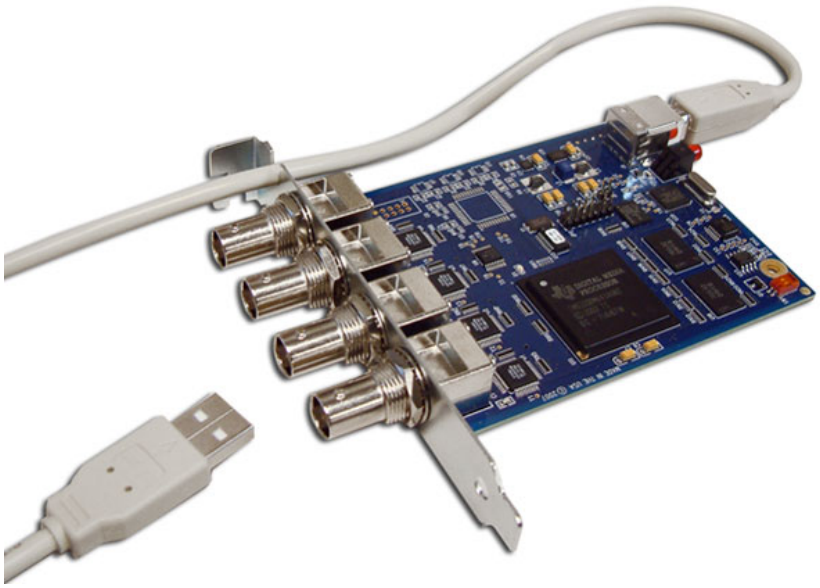


Figure 16: The Sensoray 2255 (source: Sensoray).

It is unclear why Mac OS X is able to provide such consistent FPS, shown in Graph 15, while the stereo frames themselves are not consistently delivered at the same time. The FPS is important since the search algorithms tested in this thesis are all iterative approaches. This means that the solution would preferably be as close as possible to the previous solution because it allows for faster converging of the

algorithm. A very low FPS would give the search algorithm a very hard time to find the correct position and pose, if it would be able to find the object at all.

3.5 EDGE ENHANCEMENT ON THE VIDEO FRAMES

The Sobel operator is used to perform edge enhancement on the video frames. It is implemented to run on the GPU using shaders. Before explaining how the Sobel operator is implemented, it is important to understand what a shader is and how everything is communicated between GPU and CPU. Also, it is important to understand what the difference is between performing calculations on the GPU and CPU.

3.5.1 *GPGPU*

The GPU is designed to perform calculations for visual display such as video games. However, the calculations done on the GPU are substantially different from the calculations done during scientific research. Video games use shaders to make a scene look as real as possible through the use of physics, lighting, and shading, among other things. However, the GPU can also be used to perform calculations that are normally done on the CPU. Making use of the GPU to do such calculations is called "general-purpose computing on graphics processing units" (GPGPU). An important distinction between normal usage of the GPU and GPGPU is that GPGPU requires no visual output. Visual output is normally displayed on the screen by making use of a frame buffer that contains all the pixel data that needs to be displayed on the screen. Since no visual output is required for GPGPU, the output is written to a different type of frame buffer, also called a "frame buffer object" (FBO). Writing to an FBO is also called off-screen rendering, since no output ends up on the screen. An FBO has no actual memory to store anything but the programmer is instead required to attach a texture to the FBO, which serves as the memory for the output buffer. A texture in this case is a one or two dimensional matrix that contains pixel data. The textures that are attached to an FBO are also called color attachments. Depending on the video card, it is possible to attach up to eight or more different textures to an FBO, making it possible for a shader to write output to more than just one texture.

3.5.2 *Shaders*

A shader is at the heart of every GPGPU program. It is in essence a program that runs on the GPU and determines what should be done with the input. A shader is very different from a normal program that runs on the CPU. It is specifically designed to run in parallel, while normal CPU programs can run in parallel or in sequence. This distinct difference changes the way a shader is programmed quite substantially

from a normal program. One of the differences that has the most impact is the fact that you cannot access the results from other threads within a thread, nor is there any other communication between threads. Also, input and output is handled differently for each type of shader. Recently, there has been a lot of development regarding shaders however the two most common ones that are supported by most video cards are:

- Vertex shader; the scene is made up out of vertices, where each vertex is processed by a thread of a vertex shader:
  - Input: vertex position, normal direction (for lighting), vertex color.
  - Output: vertex position.
- Pixel shader; each thread processes exactly one pixel for every pixel present in the output buffer:
  - Input: pixel color, texture coordinates, position of the pixel in the output buffer.
  - Output: pixel data for every attached and activated color attachment.

Apart from the default input data that every shader receives, it is also possible to give input to shaders using special variables called "uniform variables." These uniform variables are declared inside a shader which are set before the shader is run. An example of basic vertex and pixel shaders are shown in Listings 4 and 5. The pixel shader shows an example of a uniform variable that is set beforehand.

In every shader there are several predefined variables that are always present. For example, gl_Vertex is always accessible inside a vertex shader, which describes the vertex given to the shader. Also, the matrix gl_ModelViewProjectionMatrix describes the model-view-projection matrix to transform vertex coordinates to screen coordinates. Apart from predefined variables, there are also variables that must be set inside a shader: gl_Position must be set inside a vertex shader and the first element of the gl_FragData must be set inside a pixel shader. These serve as output for the shader itself.

```
1 void main()
2 {
3   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
4 }
```

Listing 4: A basic vertex shader which transforms the input vertex world coordinates to screen coordinates.

As mentioned before, the GPU is built to perform graphics calculations. This is the reason why in a normal shader, like the one shown in Listing 5, colors are assigned to the first color attachment buffer (gl_FragData[0]). However, colors are defined as vectors of four elements which hold numbers describing how much color should be

```
1 uniform vec4 outputColor;
2
3 void main()
4 {
5   gl_FragData[0] = outputColor;
6 }
```

Listing 5: A basic pixel shader that outputs pixels with the color that is set in the uniform variable outputColor.

displayed for each color channel. These numbers might hold color information but nothing stops the programmer from giving different meanings to the numbers.

```
1 uniform sampler2DRect input_data;
2
3 void main()
4 {
5   gl_FragData[0] = texture2DRect(input_data, gl_TexCoord[0].xy)
        *2.0;
6 }
```

Listing 6: A shader which multiplies every data element of the input texture by two.

A GPGPU pixel shader always uses one or more textures as input and also one or more textures as output. The input textures are supplied via uniform variables and the output textures are bound as color attachments to the FBO. This way every pixel, or data element, of the input texture is processed and transformed in some way, as specified by the shader, finally ending up in the output texture to form the complete result. An example of a pixel shader which performs these actions is shown in Listing 6. On the first line the uniform variable input_data is defined. This will be set with OpenGL to contain the input texture. The function call texture2DRect on line 5 will find the pixel data of the input texture at the position identified by gl_TexCoord[0]. The pixel of the input texture will be multiplied by two and afterwards it will be assigned to gl_FragData[0] as output.

3.5.3 *GPGPU techniques*

So far, all the shaders transform some kind of input texture to an output texture of the same size. In other words, each input element has an output element. This technique is called a map operation. Determining the sum of a dataset would be highly inefficient using a map operation. Since the sum of a dataset requires one single answer, or exactly one output element, the shader would need to loop over the entire input texture and give the final answer to the output texture. This output texture would have a size of one by one pixel. However, this does not take advantage of the possibilities of the GPU as a parallel processing unit since everything happens in sequence. Instead, a technique called reduction is used to perform operations like these which is much

more efficient on the GPU by making use of parallelism. The reduction operation is performed in multiple passes, where each pass reduces the size of the output texture. Each output pixel determines the sum, or any other kind of operation, of four input pixels. This process is repeated until the output texture has a size of one by one pixel. This can be read back to CPU memory to be used in the rest of the program or it can be used by other shaders as an input texture. An example of how a reduce operation works, is shown in Figure 17.

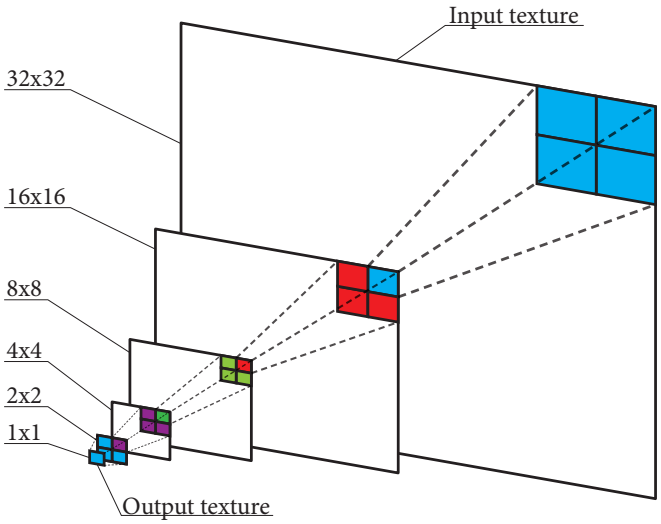


Figure 17: The GPGPU reduce operation. The four blue pixels of the input texture are for example summed together. Finally the output is delivered as a pixel in the next texture. Each pass reduces the size of the texture, until a texture with a size of one by one pixel remains with the final answer. The pixels of the two large textures are displayed larger than they actually would be. This is done for visual aesthetics. (author R. Boer)

An example of a pixel shader that can be used to determine the sum of an input texture is shown in Listing 7. This pixel shader halves the size of the output texture with each pass. Finally ending up with one pixel with the final answer. Line 5 determines the correct coordinates of the data element to find. In line 7 through 10, the elements on the left, bottom, and bottom left are added to the element at the coordinates determined in line 5.

Theoretically, the reduction process would only work if the first input texture has a size which is a power of two. However, in practice this hardly ever happens. To solve this problem, the first input texture could be mapped onto a texture which is a power of two but is larger than its input. For example, an input texture of 640x480 (the resolution of the frames that are received from the hand tracker), can be mapped onto a texture which is 1024x512. With this texture the reduction operation can be performed. Note that it is not needed to have an input texture of 1024x1024 since texture lookups which lie outside the current input return zero.

Sometimes it might not be necessary to process the entire input texture, but rather just a piece. This is done by using the scatter operation. The

```
uniform sampler2DRect input_image[0];

void main()
{
  vec2 coord = gl_TexCoord[0].xy * 2.0 - 0.5;

  gl_FragData[0] = texture2DRect(input_image[0], coord) +
                   texture2DRect(input_image[0], coord + vec2(0.0,
                       1.0)) +
                   texture2DRect(input_image[0], coord + vec2(1.0,
                       0.0)) +
                   texture2DRect(input_image[0], coord + vec2(1.0,
                       1.0));
}
```

Listing 7: An example of a shader which sums up all the elements of the input texture into an output texture which is half the size of the input texture.

scatter operation uses vertex shaders to alter which pieces of the texture need to be written to by altering the vertex positions. Normally a square polygon is drawn on the entire screen so that each pixel of the input is processed. The scatter operation changes the location of this polygon so that only parts of the input texture are processed.

3.5.4 *Making optimal use of the GPU*

It should now be clear how different it is to program a GPU as opposed to a CPU. Apart from having different techniques to solve problems, there are also a lot of different factors to consider. Some can increase or decrease performance, depending on the way they are used. One of the things that was considered during the creation of the HPE algorithm, was the ability to write to multiple color attachments at once. If a shader would be able to perform a map operation on eight input and eight output textures at once, would this be faster than doing the same operation while using only one input and output texture? A test was performed to confirm whether this was the case. During the test, the GPU receives a test image for each color attachment. The shader then applied a preliminary version of the Sobel operator on every input texture. This process was repeated for a full 30 seconds. The textures that were processed per second were averaged over 30 seconds. The results were recorded to see which number of color attachments would give the most performance.

Figure 18 shows the final result of the test. As can be seen from the graph, most textures were processed when two color attachments were used. This setting will be used in final implementation of all shaders, where possible.

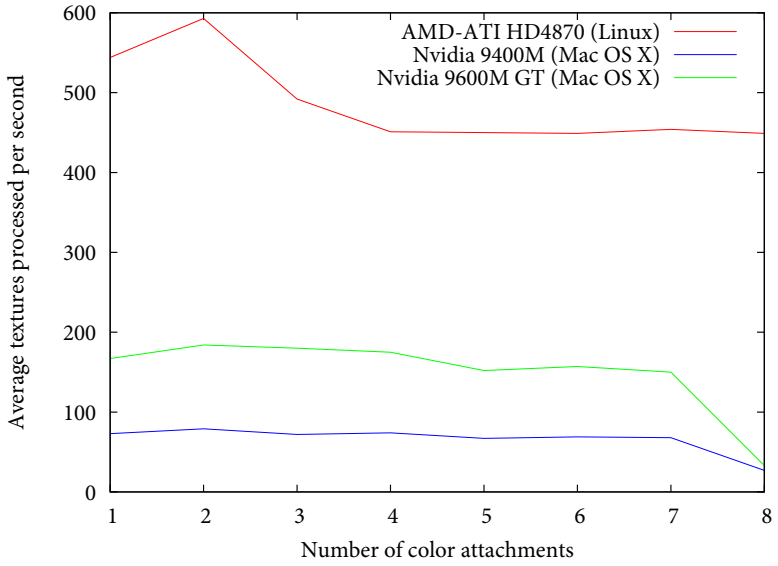


Figure 18: Texture processing speed with a different number of color attachments used. Tested on different systems and different video cards. Test system one and two described in Appendix A were used.

3.5.5 *Linear separable filter*

Research was performed to see what the best way would be to implement the Sobel operator. It turned out that an extra change could be made to the Sobel operator to decrease the number of calculations. This change can be made because the kernels of the Sobel operator are linearly separable. By linearly separable is meant that the kernel can be split up in two vectors as displayed in Equations 3.1 (for the horizontal kernel) and 3.2 (for the vertical kernel).

$$\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} +1 & 0 & -1 \end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} +1 \\ 0 \\ -1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \tag{3.2}$$

Now the horizontal and vertical derivatives can be calculated using the Equations displayed in 3.3 and 3.4.

$$G_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \left(\begin{bmatrix} +1 & 0 & -1 \end{bmatrix} * A \right) \tag{3.3}$$

$$G_y = \begin{bmatrix} +1 \\ 0 \\ -1 \end{bmatrix} * (\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * A) \tag{3.4}$$

The fact that the kernels are linearly separable matters because fewer calculations are needed to get to the same end result. If the Sobel operator is applied to an image of size $M * N$, with the default kernels of size $P * G$, this means that $M * N * P * G$ additions and multiplications are needed. However, using the vectors of the linearly separated kernels, the number of additions and multiplications is brought down to $M * N * (P + G)$.

The implementation of the linearly separated kernels, along with the Sobel operator, uses two render passes. In the first render pass, the image is convolved using the first set of vectors for the horizontal and vertical direction. This render pass uses two color attachments to write the output of the horizontal and vertical direction to two different textures. The second render pass uses both textures as input to produce one output texture. In this output texture, the last two vectors are used to convolve both input textures to produce the final result.

Initially, this implementation did not produce the required result. The problem was that the intermediate results between the first and second pass were clamped to a number between zero and one. This is done by the GPU because it normally only works with colors. Colors are defined by four 8-bit numbers, all between zero and one, hence the clamping between zero and one. Textures can be defined with different internal formats to let the GPU know that it should not clamp the values. The internal formats where clamping does not occur uses either 16-bit or 32-bit float numbers for each color channel. The problem with these formats is that the GPU is optimized for 8-bit numbers. So by using 16-bit or 32-bit numbers, the performance decreases.

Another solution to the clamping problem, is to make sure the result does not exceed a value of one. The vertical direction is the only one with the possibility to exceed the value of one after the first pass, since it can have a maximum value of four $(1 + 2 + 1)$, whereas the horizontal direction has a maximum of 1 $(1 + 0 + 0)$. So the vertical direction is the only one that needs to be altered. By dividing the vertical direction by four in the first pass and multiply it again by four in the second pass, it becomes possible to use 8-bit numbers. The number four is used since this is the maximum value which a pixel in the vertical direction could have. A bit of accuracy is lost between the division and multiplication, but the end result is still acceptable.

The final test uses the standard Sobel operator, as well as the 8-bit, 16-bit, and 32-bit variant. The 8-bit variant will use the division and multiplication by four to ensure clamping does not occur. The results of the test can be seen in Table 1. They show a rather peculiar result. As expected, the 16-bit and 32-bit implementation are much slower than the default implementation. However, it seems that the Nvidia video

	AMD-ATI HD4870	Nvidia 9600M GT
Standard Sobel operator	1.0	1.0
8-bit textures	1.7	0.8
16-bit textures	2.2	1.3
32-bit textures	3.6	2.4

Table 1: Normalized FPS to indicate speed increase or decrease for the linearly separated Sobel filter, compared to the default Sobel filter. The vertical direction of the first pass of the 8-bit textures is divided by four and multiplied again by four in the second pass to ensure clamping does not occur.

card used in the Mac OS X system is better suited to handle these kind of calculations, since it delivers a speed increase with 8-bit textures. The results also show, that even though the number of calculations is decreased using the linearly separated filters, the ATI video card has a harder time executing the linearly separated filter than the default implementation. The final implementation might use either the default or the linearly separated filter, depending on the speed the platform delivers.

3.5.6 *Thresholding*

Thresholding can be used with the Sobel filter to ensure that only edges are returned. The lower the threshold, the more edges are returned. However, a threshold that is too low can return noise and irrelevant features of the image. An appropriate threshold would need to be selected that minimizes noise and returns only relevant features of the images. During the process of selecting an appropriate threshold, it came to the attention that it might be better to not apply a threshold at all. A test was run to see which setting would give a result that has a smooth fitness landscape which is easily traversable by the selected algorithms. Or in other words, the search space that is traversed by the search algorithms should have an inverted bell shaped curve, or a well curve, shown in Figure 19. Preferable, the well curve should be as large as possible to give the search algorithm a large basin of attraction. Also, it should contain as few local minima as possible to prevent search algorithms from getting stuck in local minima. If local minima do exist, they should allow a search algorithm to jump out of them with little effort so that little time is wasted inside local minima.

The test used an image of a hand as input. This image was fed through the Sobel operator, with and without a threshold. The 3D hand model was fed through the Sobel operator as well, also with and without a threshold. The model was turned around 360 degrees on its x axis, to simulate a brute-force search algorithm trying to find the correct angle of rotation for the hand model. After each degree turned, the Sobel filtered 3D hand model was subtracted from the edge image of the

hand. Then all values would be summed together to see what kind of error the model would have.

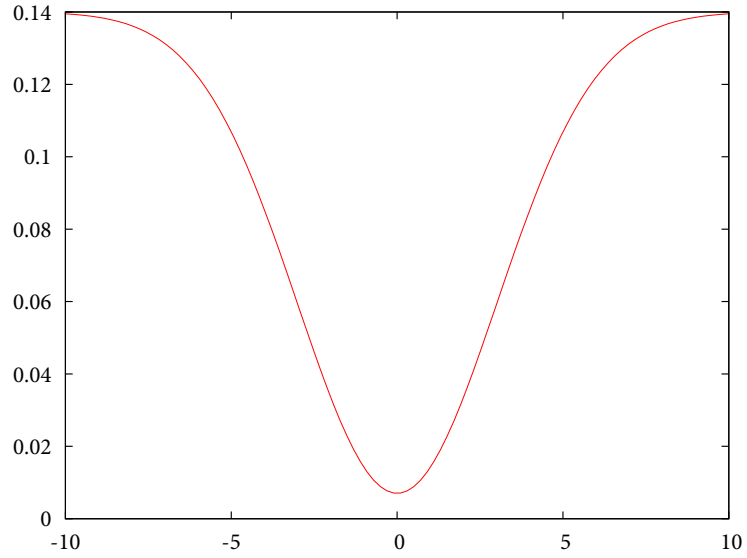


Figure 19: An example of an inverse bell curve, also know as a well curve.

The result of the test can be seen in Figure 20. In the case of the test, 88 degrees put the 3D model right on top of the hand in the image. So this is also where the lowest error is recorded. As can be seen from the figure, if both images are thresholded, the shape of the fitness landscape is almost flat. If, however, both images are not thresholded, the result is a much smoother area and much more like the well curve that is sought after, along with the other properties described earlier.

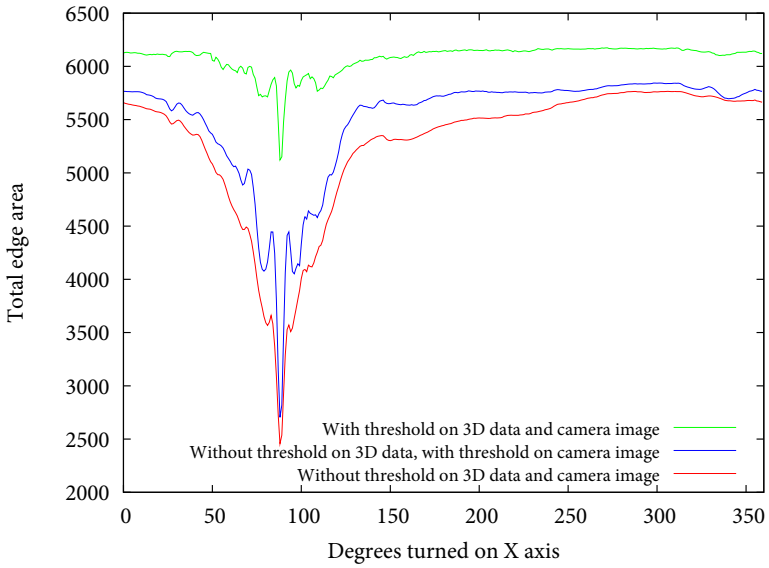


Figure 20: Total edge area after subtracting the 3D hand model image from the camera image with and without thresholding.

The test showed that thresholding would not give the search algorithm any advantage. By not applying thresholding, execution time would be saved, since fewer calculations have to be made. Also, it would give the

search algorithms a better search space that would allow it to converge faster.

3.5.7 *Implementation of the Sobel operator*

Before the Sobel operator can be applied to the video frames, the frames first need to be transferred to GPU memory. The frames are copied to the GPU memory as textures so that the Sobel shader can use them as input. Also, the bounding boxes that are received from the hand tracker are supplied to the Sobel shader as uniform variables. This way the Sobel shader knows which pixels to process so that the execution time is decreased. As explained in Section 3.5.4, the shader can process two images at once for optimal performance. In this case it is possible to make great use of this, since both frames of each of the cameras can be processed in the same pass. Section 3.5.6 showed that it is not necessary to apply thresholding, so the Sobel shader makes no use of thresholding and only outputs the edges without any post processing.

The Sobel shader makes no use of linearly separated filters because the final test system will be test system 1, described in Appendix A. This system is equipped with an HD4870 ATI video card. Section 3.5.5 showed that using linearly separated filters had an adverse effect on the performance with that particular video card.

The code that asks the GPU to execute the Sobel shader is implemented entirely in C. Since this code is executed many times during each frame it has to be as fast as possible. Python can not be used for this since it is several times slower than C. The shader that is used to detect edges on the 3D model is exactly the same as the Sobel shader that is used to detect edges on the video frames.

3.6 ADJUSTING THE 3D HAND MODEL USING A MULTIDIMENSIONAL SEARCH ALGORITHM

Section 2.7 showed which algorithms would be used to determine the correct pose and position of the hand. As a reminder, these are the chosen algorithms:

- Secant method,
- Nelder-Mead method,
- Simulated Annealing.

The video frames of the hand tracker are fed through the Sobel filter before the algorithms can perform their search. After this, a search algorithm is called to find the best fit. Each time a search algorithm wants to know the error of a particular location and pose, the 3D

model is rendered and edged with that particular location and pose. Afterwards, the edged 3D model is subtracted from the edged video frame and finally the remaining pixels are summed together, to indicate what the total error is. This total error is cached so that other calls with the same DOFs are immediately returned from the cache.

After the search method has converged, it will return the best position and pose it has found for the object. This pose and position is returned to the interface described in the master thesis of Bruining [10].

The search algorithms do not start from scratch every frame, but instead they use the last known position and pose. However, during the startup phase of the system, no position or pose is known beforehand. To know where the hand is in the frames, a global search would need to be performed. However, this cannot be performed during the startup phase since one of the requirements of the system is that it should only require a brief calibration period. A global search would simply take too long. Instead, the user is required to put his or her hand in a position and pose that is known beforehand. From there, the search algorithms can perform a brief calibration to match the hand in the frames. This saves a considerable amount of startup time.

Before this can be done, however, several problems need to be solved. Every search algorithm has their own practical problems that need to be solved before it can give an adequate performance. The next sections will describe the practical problems of each of the algorithm and will try to solve them. In Chapter 4 each algorithm will be evaluated with different settings, to see which setting performs best.

3.6.1 *Secant method*

The Secant method has undergone many small tests to see how the method would perform in relatively simple search space with one and two DOFs. It turned out that there were many small details that had to be changed before the method would give a satisfactory result.

The first thing that had to be changed was to let the method search through multiple dimensions. Since the method itself is actually meant to only search through one dimension, it was defined that the method would perform one dimensional steps through the multidimensional space. So each step the method would change the search dimension to the next dimension and do a one dimensional step in this dimension, and so on and so forth.

Another problem was that the method tends to go to infinity very quickly if the search space is very flat. The search space is flat unless the object (partially) overlaps the object from the video frame. So if the method would ever try to do a step to infinity, the step size would be replaced by a predefined value. This value is variable and can be changed to find the optimal configuration during the final tests.

```
x_n                       = start position
x_{n-1}                   = start position + start position offset
current search dimension = N - 1
dimension multiplier      = vector of zeros with size N

for (i = 0; i < maximum steps; i++):
  dimension multiplier[current search dimension] = 0
  current search dimension  = (current search dimension + 1) mod N
  dimension multiplier[current search dimension] = 1

  d = (x_n - x_{n-1}) / (f(x_n) - f(x_{n-1})) * f(x_n)

  if any value of d is infinite:
    d = replacy any infinite by the infinite replace value
  else if any value of d is -infinite:
    d = replacy any -infinite by the infinite replace value

  # make sure the best 2 values are continued with in the next step
  if f(x_{n-1}) > f(x_n) and f(x_{n-1}) > f(x_n - d):
    if f(x_n) > f(x_n - d):
      x_{n-1} = x_n
      x_n = x_n - d
    else:
      x_{n-1} = x_n - d
  else if f(x_n) > f(x_{n-1}) and f(x_n) > f(x_n - d):
    if f(x_{n-1}) > f(x_n - d):
      x_n = x_n - d
    else:
      x_n = x_{n-1}
      x_{n-1} = x_n - d
  else if f(x_n) == f(x_{n-1}) and f(x_n) > f(x_n - d):
    x_n = x_n - d
  else:
    # since x_n - d is worser than x_n and x_{n-1}, the method would
    # come to a halt if nothing is changed here
    if f(x_n) > f(x_{n-1}):
      if sum(x_n * dimension multiplier) > sum(x_{n-1} * dimension
          multiplier):
        x_n = x_{n-1} - (x_n - x_{n-1})
      else:
        x_n = x_{n-1} + (x_{n-1} - x_n)
    else:
      if sum(x_n * dimension multiplier) > sum(x_{n-1} * dimension
          multiplier):
        x_{n-1} = x_n + (x_n - x_{n-1})
      else:
        x_{n-1} = x_n - (x_{n-1} - x_{n-1})

return the setting with the smallest error (either x_n, x_{n-1}, or
    x_n - d)
```

Listing 8: The Secant method in pseudo code. N represent the number of search dimensions.

Other problems that were encountered, was the way the method chooses its next candidates to work with in the next step. How this is dealt with can be seen in Listing 8. Several variables can be changed inside the code. These are:

- Start position offset,
- maximum steps,
- the infinite replace value.

The variables will be tested in Chapter 4 to see which setting performs best. However, the maximum steps will be set to a value that will assure that the method uses roughly the same amount of search space information as the other methods.

3.6.2 *Nelder-Mead method*

The Nelder-Mead method itself is exactly implemented as described in Section 2.7.2. As explained in the same section, it seemed that the Nelder-Mead method is stuck in local minima very quickly since it does not allow any steps that are worse than the previous steps. This was shown during preliminary tests. Graph 21 gives an idea of how many small and large minima there are with only two DOFs. In this, the X and Y rotation DOF.

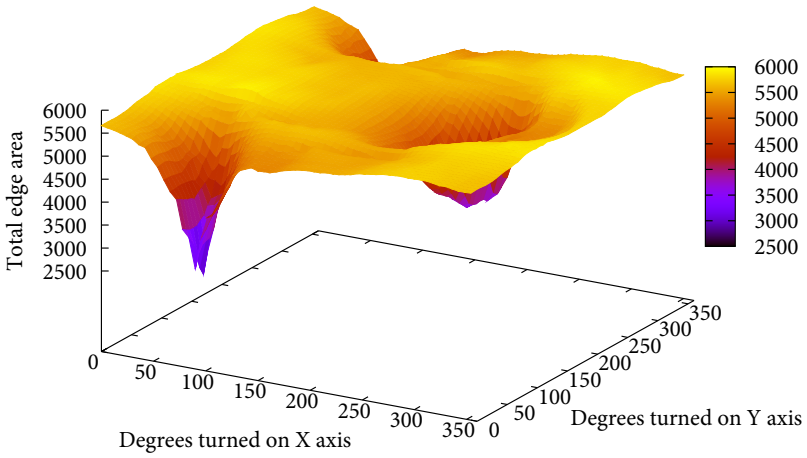


Figure 21: Results of the error measurements done while rotating the hand model on its X and Y axis. At 88 degrees on its X axis and 0/360 degrees on its Y axis lies the best result with the lowest error.

Several additions are made to give the Nelder-Mead method a chance to find the global optimum. The first being the size of the starting simplex. By specifying a large starting simplex, it becomes possible for the Nelder-Mead method to search through a larger area, while

hopefully not becoming stuck in one of the local minima. The second addition is that the method is started several times and not just once. It is started once at the starting position and another two times for every search dimension.

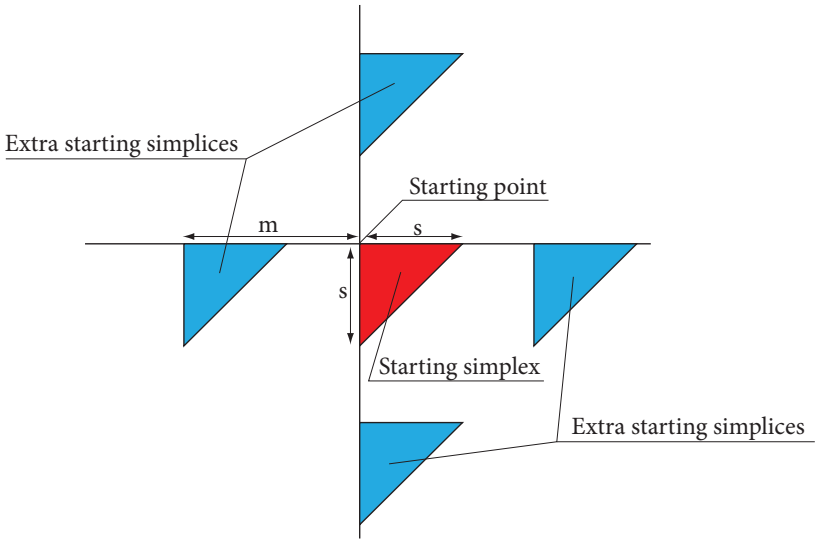


Figure 22: A visual representation of all starting simplices for the Nelder-Mead method, with a search space of two dimensions. Where S indicates the size of the simplices and M indicates the amount of movement along each dimension for each extra starting simplex. M is equal across all dimensions.

Figure 22 displays an example how the Nelder-Mead method starts for each search dimension. The variables that can be changed are the size of the starting simplices (S) and the amount of movement (M) that is done for the extra simplices around the starting position. Another variable that is normally restricted to a certain maximum is the number of steps the method may do. However, several tests showed that 10 steps were enough for the method to sufficiently converge to a point where further steps were not necessary anymore since the simplex was simply too small after 10 steps to make any visual difference. So the maximum steps of the method is set to 10.

3.6.3 *Simulated Annealing*

The implementation of Simulated Annealing (SA) has more freedom than the Nelder-Mead method. The Nelder-Mead method describes exactly what steps need to be taken, whereas SA only describes what should be done but not how. Listing 9 shows the pseudo code of SA again, to see what needs to be filled in.

The initial configuration is set in line 1. As previously explained, the starting position of all algorithms is the last known position. So for each successive frame the last known position will be set in line 1. For the first frame, the position is a position that is known beforehand. This position is where the user puts his or her hand when it wants to use

```
1  x = initial configuration
2
3  while p < maximum iterations:
4    i = random neighbor
5
6    if f(move(x, i)) is better than f(x):
7      x = move(x, i)
8    else accept new move with a certain probability:
9      x = move(x, i)
10
11   p = p + 1
```

Listing 9: The Simulated Annealing process in pseudo code.

the system. The position can also be used when the system loses track of the hand.

Line 3 gives us the first variable of the algorithm. However, this variable will be set to a fixed number. This fixed number is derived from the number of iterations the Nelder-Mead method has to do. By doing this, it becomes possible to give a good comparison between both methods since both methods use roughly the same amount information of the search space.

Lines 4, 6, 7, and 9 require a random neighbor and move function. The implementation of this function is shown in Equation 3.5. Where ND is a vector consisting of random values from a normal distribution and radius is a variable to change how big the maximum steps are.

$$new\ position = current\ position + radius * \mathrm{ND} \tag{3.5}$$

The final pseudo code of the implementation is shown in Listing 10. The code shows the part where the best settings are always remembered. This is important since SA is not guaranteed to follow the path with the best settings. It is also the strongest point of SA, since this allows the method to escape out of local minima. However, after the maximum iterations are passed, it is not guaranteed to return with the best settings. Exactly the reason why the best settings are remembered during the entire process.

Another shortcoming of SA is that it is possible for SA to drift away from the global minimum when the temperature is high. Since SA accepts detrimental steps with a high probability during this phase of the process. To make sure this does not happen, the number of detrimental steps are remembered. If the number of detrimental steps are greater than the maximum detrimental steps it is allowed to do, then the current settings are replaced by the best settings it has found during the entire process. Several tests were done to see what the optimal number of maximum detrimental steps would be. After the maximum of 10 detrimental steps no improvements were seen in the results, so the maximum detrimental steps are set to 10.

```
current DOF = start DOF
best DOF    = start DOF

while temperature >= 0.0 and iterations < maximum iterations:
  new DOF = current DOF + (radius * vector random values)

  if f(current DOF) > f(new DOF):
    # current DOF is better than the last
    current DOF = new DOF
    temperature = temperature - temperature step
    detrimental steps = 0
  else if random() > (1.0 - temperature):
    # accept worser DOF with certain probability
    current DOF = new DOF
    detrimental steps = detrimental steps + 1

  if f(best DOF) > f(current DOF):
    # remember the best DOF
    best DOF = new DOF

  if detrimental steps > maximum detrimental steps:
    # reset to the best DOF because too many detrimental steps were
        taken
    current DOF = best DOF
    detrimental steps = 0

  iterations = iterations + 1

return best DOF
```

Listing 10: The final implementation of Simulated Annealing in pseudo code.

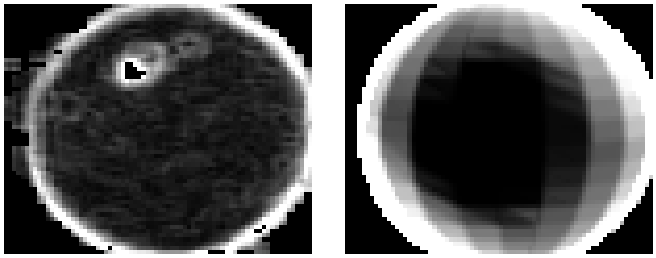
The variables that are present inside the pseudo code can be changed to see what kind of settings give the best performance. These are:

- Radius: changes the size of the steps inside the search space.
- Temperature step: the bigger the temperature step size, the faster the algorithm only accepts better settings. It is a number between 0 and 1. The method could get stuck in local minima if the temperature step size is too large.
- Initial temperature: a high temperature allows the algorithm to get out of local minima. A low temperature assures that it only accepts settings that are better than the former.

The best settings for the variables are found during the tests discussed in Chapter 4.

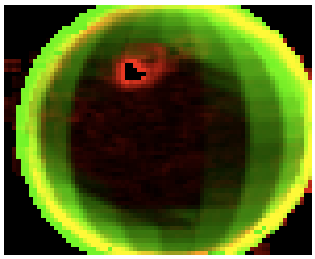
3.7 DETERMINING THE ERROR OF THE 3D HAND MODEL

As mentioned before in this thesis, the error of the model is determined by subtracting the edges from the video frame from the edges of the 3D model. A visual representation of how this works is shown in Figures 23a, 23b, 23c, and 23d.

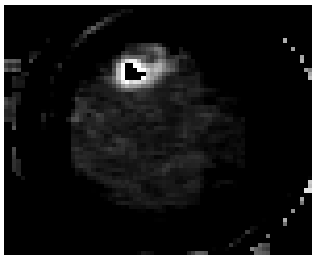


(a) Real ball Sobel filtered

(b) 3D ball Sobel filtered



(c) Real and 3D balls difference in color



(d) Real and 3D balls subtracted

Figure 23: An example of how the subtraction process works. The Sobel operator is applied to both images A and B. Image C shows the difference between images A and B in color. Red indicates where image B has no edges, green indicates where image A has no edges and yellow indicates where both images have edges. Image D shows what remains after image B is subtracted from image A.

All the pixels in the final texture, that can be seen in Figure 23d, are summed together. This gives the algorithms a measure how good the fit is when compared to other settings of the model. Also, the maximum error can be determined by summing the pixels together of the original Sobel filtered video frame. This maximum error can be used to see how good a fit of a model is compared to other video frames with their best fit.

The summation of all the pixels is done by using the reduction technique described in Section 3.5.3. An extra pre-processing step is performed before the actual summation. This is to reduce the number of wasted calculations. Normally, during the summation process, each shader thread sums up all the values for each pixel and returns the sum of it. However, each pixel consists of four color channels while the edge information needs only one channel since it consists of only one number. So each time an addition is made, three color channels are also added together which are not needed for the final result. The pre-processing step makes use of this by applying a reduction step but without performing the actual summation. Each thread of the post processing shader puts the first color channel of four input pixels into one pixel, each value in one color channel. This way the additions done during the actual summation of the entire input texture does not waste any calculations. This reduces the processing time by halve.

As stated before, the transfer from GPU textures to CPU memory is quite expensive. All communication between the CPU and GPU, is therefore, kept at a minimum. The only time the values from a texture are read back to CPU memory is when the error from the fit of the model is needed. By making good use of the GPU-CPU asynchronous communication, it is possible to decrease the time that is needed to read values from textures. The only way to use this asynchronicity is by making use of a special buffer called "pixel buffer object" (PBO). A PBO defines a block of memory which can be written to and read from using direct memory access (DMA), without using any actual CPU cycles. OpenGL can be asked to read from or write to a PBO and it will decide for itself when this actually happens. Since both error values reside in two different textures, the program can ask for the value from the first texture and then ask for the value from the second texture. After asking for the value, the GPU transfers the values to the PBOs. Another operation is used to read the values from the first and then the second PBO. This order of operations is important since it creates time between asking for the value and actually reading the value. This time can be used by the GPU to fulfill the request from the program by scheduling the appropriate operations in optimal order. Using PBOs decreased the time to read values from the textures by roughly 20%. Unfortunately, OpenGL might seem cross-platform compatible, Linux had trouble with reading from PBOs. On Mac OS X it was possible to use two different PBOs, one PBO for every texture to read from. However, this gives problems on Linux, or at least the driver that was used. Using one PBO was possible, but when more were used, the other PBOs would give results that did not make any sense. So in the end Linux uses one PBO and Mac OS X two, decreasing the benefit of using PBOs.

Initially, the errors of both webcams were added together as the final error measurement for the search algorithms, however, it turned out that this does not always give the correct result. When the object is closer to one webcam and further away from the other, the object appears larger on one camera while being smaller on the other. For the error measurement, this means that one webcam will give a higher error since the object is bigger, even though the visual fit might be very good but the 3D object just does not occlude enough of the actual object. However, there are also instances where the search algorithm might get stuck in a local minimum because of this problem where one webcam might overrule the other. This can happen when the object in the other camera is so small, that it becomes insignificant for the final error. When this happens, the search algorithms are in essence fitting the object only on one webcam while disregarding the other. The result is that the object has a proper fit on one webcam, while the other might not even be close. Several different formulas were defined to try and solve this problem. The following formulas are tested:

1. $A + B$,
   this function is tested to serve as a baseline for the other functions;

2. $max(A, B)$,
   the max function is used to only use the error of the worst camera, so that even though the fit is very good on another camera, this will be disregarded until the fit is better on the other camera;

3. $(A + B)^2$,
   this creates a search space which has steeper hills and valleys;

4. $A^2 + B^2$,
   a function that lies between the sum and first and third function;

5. $\begin{cases} A * 1.5 + B, & \text{if } A < B \\ A + B * 1.5, & \text{else} \end{cases}$,
   this function tries to give the camera with a bad fit a penalty so that the other camera becomes more important;

6. $\frac{A}{\text{maximum error}} + \frac{B}{\text{maximum error}}$,
   instead of using the error value itself, a value is calculated that describes how bad the fit is when compared to the maximum error, hereby creating two values that can be compared to each other regardless of the size of the object in each frame.

Where A and B are the errors from each camera. Tests in Chapter 4 will show which of the functions increases the performance of the search algorithm.

3.8 SUMMARY: FINAL IMPLEMENTATION OVERVIEW

Figure 24 shows the entire HPE system and its internal processes. The ARToolkit matrices describe how both cameras are oriented and what their position is, in relation to the tag. These matrices are determined once when the system is first started. Since the camera locations and orientations do not change, it is not necessary to perform the detection sequence every frame. The search algorithm performs as many loops as its settings allows it to do. Once the search algorithm is done, it returns the best position it has found during its iterations. This is given to the user interface which sets the correct DOFs for the complete model. The Sobel shader is displayed twice in the figure, but the object location is only given once as input. Both Sobel shaders know the object location but the lines have been left out to decrease the complexity of the figure. The post processing step that is done before the actual summation shader is also left out for the same reason.

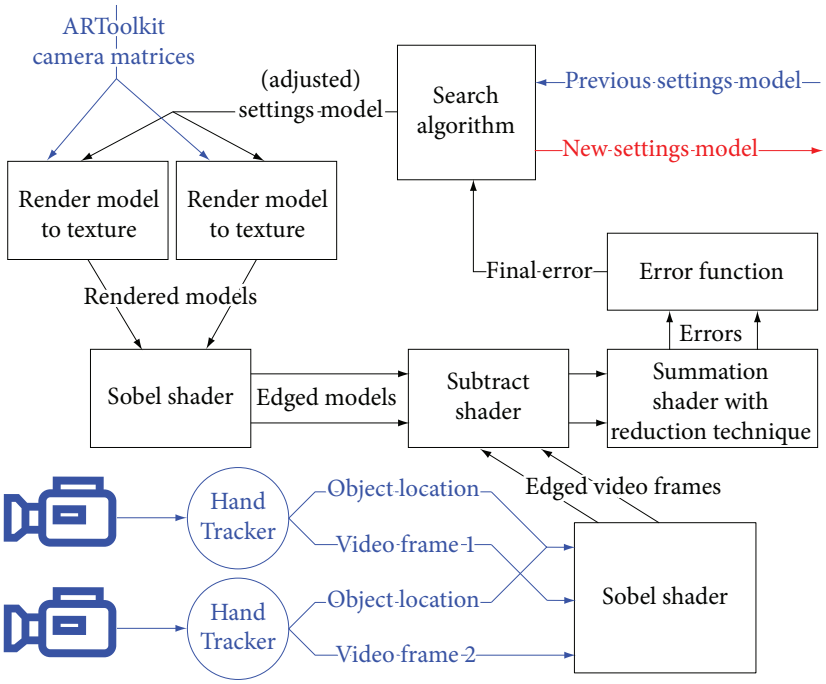


Figure 24: Implementation overview. Blue represents input and red represents output of the HPE algorithm.

Now the entire system is known, every algorithm is implemented and most of the implementation problems have been solved, it is time to test the system and see how it performs. The results of all tests are described in the next chapter.

4

EVALUATION

This chapter will describe several tests that have been done to see if the algorithms are really able to track the objects. Each algorithm has a range of different variables that will be tested to find the best settings.

The first tests are not done with the hand model, but instead a ball and an oblong is used. This is done to see whether the algorithms are be able to track such relatively simple objects. During this phase, the different error functions are also tested to see which function performs best.

Each test uses recordings of a number of frames. During each test, the recording is fed to the algorithms so that each algorithm receives the same input. This allows a proper comparison between the algorithms.

4.1 TEST WITH A BALL

The first test is done with a small ball which can be seen in Figure 25. During this phase the algorithms are tested to see if they can properly estimate the X, Y, and Z DOFs.

Five different recordings are made. Each test moves the ball in a particular way:

1. The ball is moved upwards. A total of 22 frames are recorded over 1.9 seconds.

2. The ball is moved downwards with 19 recorded frames over 1.6 seconds.

3. The ball is moved from back to front to back again. 44 frames are recorded over 3.7 seconds.

4. The ball is moved from side to side and back again. 49 frames are recorded over 4.1 seconds.

5. During this last test the ball is moved randomly up and down, and side to side for 3.8 seconds with a total 40 recorded frames.

The 15 FPS is not achieved as can be seen from the duration of the tests and the total number of frames. Instead, an average of 12 FPS was maintained. Even though 15 FPS would be preferable, test system 1

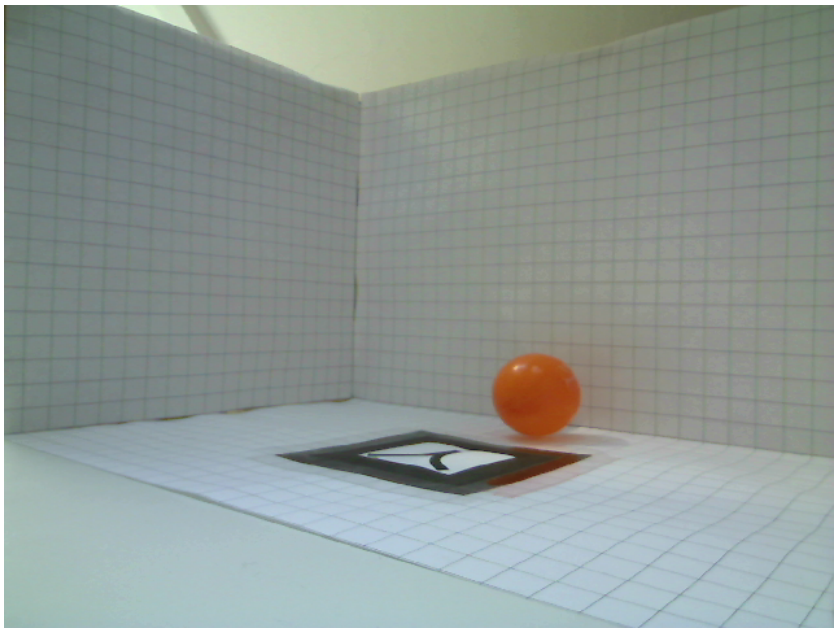


Figure 25: One of the frames of the test recordings with the ball. The tag that is shown is used to estimate the position and orientation of the camera. The recording is done inside a cube that has lines a centimeter apart so that both frames of the cameras can be visually compared to each other. This might be needed to see if both frames are taken at the exact same moment.

(see Appendix A for the specifications of the test systems) is not able to achieve this, as can be seen from the test results in Figure 15 in Section 3.4.

During normal operation of the system, the algorithms know the starting position along with other DOFs of the object. Therefore, the best DOFs of the object in the first frames of every test set is found to simulate this initial knowledge. The algorithms are responsible for determining the best DOFs for each successive frame. After each frame, the error is determined of the final DOFs that are returned by the algorithms. This is divided by the maximum error of that frame to get a number between 0 and 1, indicating how good or bad the fit is. A low number indicates a good fit, a high number a bad fit. The error rates for all frames are summed together and divided by the total number of frames to get an average error rate for the total test set. From this, the best, or lowest, number is retrieved to find the best settings, along with the corresponding error function.

The number of iterations or steps each method can do is limited to make sure that they use a comparable amount of information about the fitness landscape. It was decided that each method is allowed to do roughly 200 different error measurements for this first test. For the Nelder-Mead method this means that it can do a maximum of 10 steps for each starting point. The Secant method is allowed to do 45 steps and Simulated Annealing can be set to a maximum of 200 iterations before the method stops.

The results for the Secant method, the Nelder-Mead method, and the Simulated Annealing method can be seen in Tables 2, 3, and 4 respectively. Each set of results shows a red and green value, indicating the worst and best value of that method.

Function	Infinite replace value	Start position offset	Average over all sets
$A + B$	4	6	0.45
$\max(A, B)$	7	6	0.40
$(A + B)^2$	9	5	0.45
$A^2 + B^2$	4	3	0.45
$\begin{cases} A * 1.5 + B, & \text{if } A < B \\ A + B * 1.5, & \text{else} \end{cases}$	10	4	0.42
$\frac{A}{error_{max}} + \frac{B}{error_{max}}$	9	6	0.47

Table 2: The results of the Secant method test.

Function	Simplex size	M	Average over all sets
$A + B$	25	25	0.46
$\max(A, B)$	29	28	0.46
$(A + B)^2$	25	25	0.46
$A^2 + B^2$	26	23	0.45
$\begin{cases} A * 1.5 + B, & \text{if } A < B \\ A + B * 1.5, & \text{else} \end{cases}$	26	29	0.49
$\frac{A}{error_{max}} + \frac{B}{error_{max}}$	28	25	0.47

Table 3: Final results of the Nelder-Mead simplex method. For a description about the parameters, see Figure 22.

Somewhat unexpectedly, the Secant method actually performs better than the Nelder-Mead method. However, both methods are not able to come close to the performance of Simulated Annealing. Simulated Annealing has the ability to climb out of a local minimum and, is therefore, able to find a better position for the ball in this test. The Secant and Nelder-Mead method both have the problem that they get stuck in local minima, which would explain the difference in the results.

Simulated Annealing always performs best when the temperature increase for each iteration is set to 0. One of the most plausible reasons for this is that it allows the method to use all 200 iterations, while higher values decrease the amount of iterations it can do.

Function	Initial temperature	Radius	Average over all sets
$A + B$	0.00	5	0.30
$max(A, B)$	0.04	3	0.31
$(A + B)^2$	0.01	3	0.30
$A^2 + B^2$	0.07	3	0.29
$\begin{cases} A * 1.5 + B, & \text{if } A < B \\ A + B * 1.5, & \text{else} \end{cases}$	0.08	3	0.30
$\frac{A}{error_{max}} + \frac{B}{error_{max}}$	0.06	2	0.30

Table 4: The results of the Simulated Annealing method. The increase of the temperature for each iteration always gave the best result when the increase was set to 0 so this column is not shown here since the value is the same in each row.

Figure 26 shows the difference in performance between each method for test set 4. It must be noted that an error rate of 0.4 does not necessarily mean that the fit is bad. The model that is used to subtract from the real model is a close approximation but not a perfect match, hence the error rates never drop to zero. Also, the distance between the cameras and the ball matters as well. A higher error rate can be seen where the distance between one camera differs from the other. When the distance between the ball and both cameras is roughly the same, the error rate also decreases, as can be seen between frame 9 through 13.

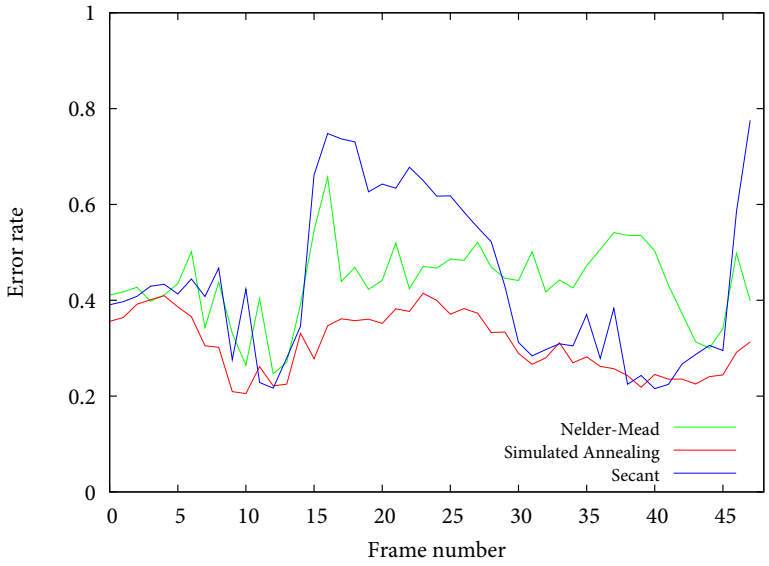


Figure 26: The results with the best settings of each method of the fourth test set where an error rate of 0 indicates a perfect match.

4.2 TEST WITH AN OBLONG

Now that all methods are shown to be able to estimate the X, Y, and Z DOFs, it is possible to continue to a slightly more complicated object. The oblong has six DOFs: X, Y, Z, roll, yaw, and pitch. The maximum number of error measurements for the methods is increased since six DOFs increases the difficulty for the methods to find the correct DOFs. For the Secant method, this means that it can now do a maximum of 135 steps. The Nelder-Mead method is still set to 10 maximum steps since the number of DOFs also increases the number of starting points to seven. The maximum iterations the Simulated Annealing can do is set to 540. The number of error measurements is roughly the same for each method when these settings are used. The maximum steps or iterations is the only change that is made to the parameters of all methods.

An example of a frame of one of the recordings can be seen in Figure 27. Again, several different recordings are made with different movements in each recording. These are:

1. The oblong is moved from back to front and upwards, while rotating the body around its X axis. 20 frames are recorded with a total time length of 2.6 seconds.

2. The object is moved from side to side and at the same time it is rotated around its Z axis. The total recording consists of 124 frames over 9.2 seconds.

3. The last recording moves the oblong around from back to front and from side to side, while rotating it around the Y axis. The recording has 75 frames, good for 6.5 seconds.

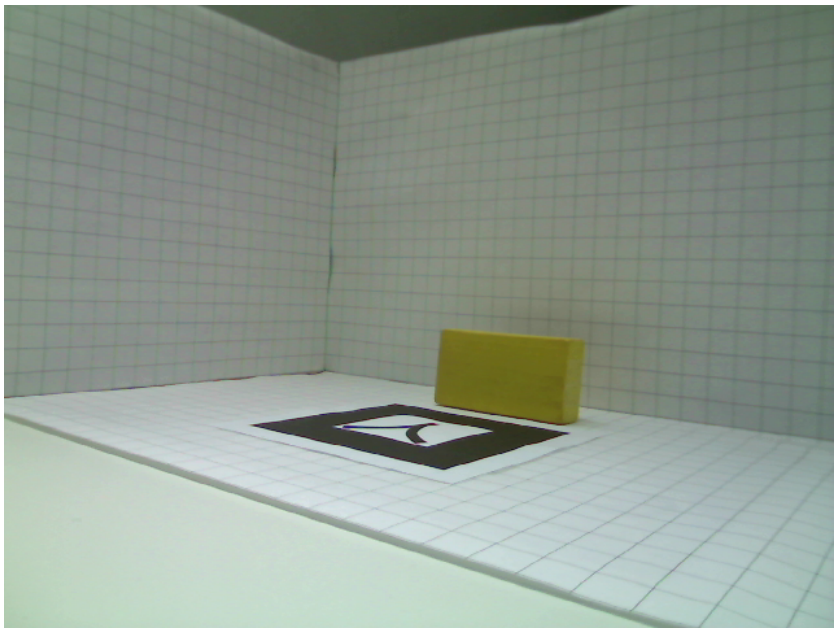


Figure 27: An example of one of the frames of the oblong test.

Due to time constraints it was not possible to test all functions, so only the baseline function ($A + B$) and the best functions for each method are tested. The results of these tests are shown in Tables 5, 6, and 7.

Function	Infinite replace value	Start position offset	Total average over all sets
$A + B$	3	9	0.80
$max(A, B)$	3	5	0.81

Table 5: The results of the Secant method test.

Function	Simplex size	M	Total average over all sets
$A + B$	27	29	0.78
$A^2 + B^2$	20	29	0.78

Table 6: Final results of the Nelder-Mead simplex method. For a description about the parameters, see Figure 22.

As can be seen from the results, the error is quite high when compared to the results from the test of the ball. This is mainly due to two reasons. The first being that the model does not exactly fit the actual object so a perfect fit is never found. The edges of the 3D object are much sharper than the actual oblong in the video frame. An effort was made to create corners that were round, like the actual object. This did have some impact however there was a second problem. The matrix returned from ARToolkit did not give a proper estimation of the camera pose and location. The object matched relatively well when the object was close to the tag, however, when the object was moved away from the tag,

Function	Initial temperature	Radius	Total average over all sets
$A + B$	0.01	3	0.69
$A^2 + B^2$	0.01	2	0.69

Table 7: The results of the Simulated Annealing method.

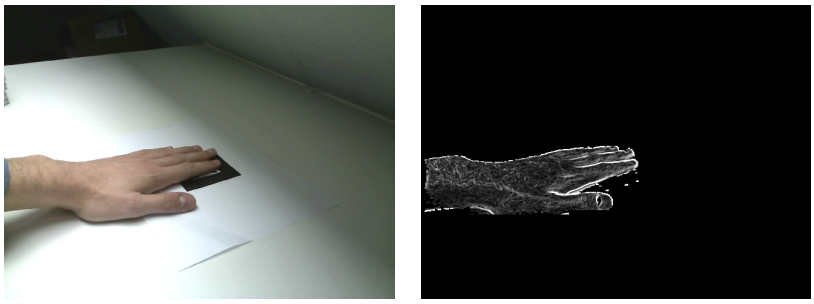
the height became less than the actual object. Multiple tags might have solved this problem by averaging the matrices for each tag returned by ARToolkit but this was not tested.

This test case also shows that the search algorithms are capable of tracking an object, even though the 3D object does not completely match the real object. This is important since the 3D hand is also an approximation of the real hand but never a perfect match.

Again, the Simulated Annealing method showed that it is was superior to the Nelder-Mead and Secant methods so for the final test, the Simulated Annealing method will be used.

4.3 TEST USING A REAL HAND

The final test is done with a normal hand. The model of the hand is first manually calibrated to roughly match the real hand. The model of the hand is limited in its joint movement and the DOFs are decreased to a total of 15 DOFs as described in Section 2.3. That is, three DOFs for the X, Y, and Z coordinates, another three DOFs for the yaw, pitch, and roll DOFs. The final 9 DOFs describe the angles for all joints of the hand. The weak constraints are not applied to the model.



(a) Example of a frame of one of the recording sets.

(b) The same frame after the Sobel filter has been applied to it. Also, the pixels that are not classified as hand pixels are black.

Figure 28: An example of one of the frames of the recording sets with its Sobel filtered version.

Recordings are made of the hand. One of the frames of the recordings is shown in Figure 28a. Figure 28b shows the Sobel filtered version of this

frame. Again, as the previous tests, each recorded set has a particular movement of the hand:

1. The hand is moved upwards without any finger movement. 65 frames are recorded over 4.9 seconds.

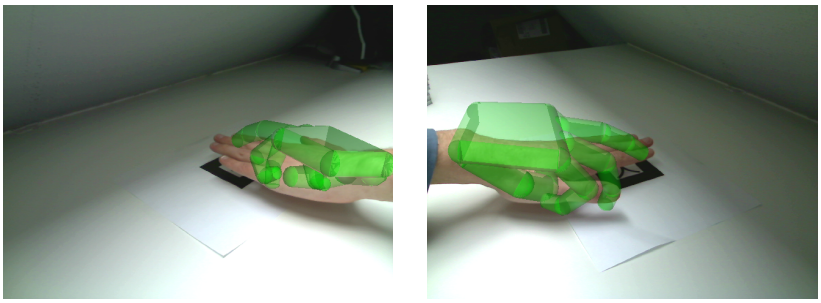
2. The hand is moved upwards and rolled 180 degrees and again rolled back 180 degrees with minimal joint movement. Total of 68 frames are recorded over 5.4 seconds.

3. The index finger is moved completely down and up after the hand is moved upwards during 3.7 seconds with a total of 34 frames recorded.

4. A fist is created after the hand is moved upwards from its starting position. The recording lasts for 3.1 seconds with 34 frames.

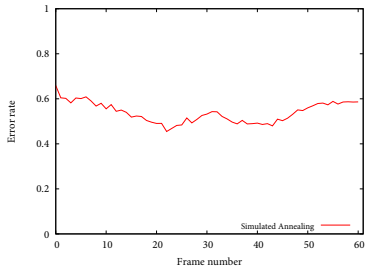
5. Same as the previous recording but this time the hand is rolled 180 degrees. 38 frames are recorded over 3 seconds.

This test will only use Simulated Annealing since it performed better than Nelder-Mead and the Secant method during the previous tests. The error function $A^2 + B^2$ will be used since it performed slightly better than $A + B$. Also, the following settings will be used: 0.01 for the initial temperature and 2 for the radius. The maximum iterations will be set at 1000 iterations.

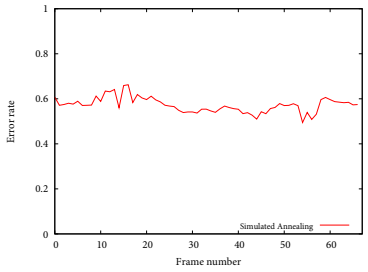


(a) Left camera (b) Right camera

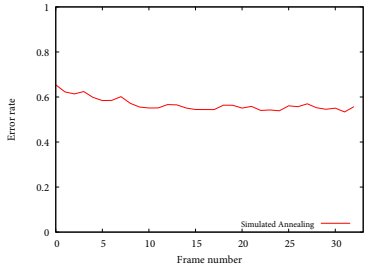
Figure 29: One of the poses that is returned by Simulated Annealing as the "best" pose (shown in translucent green).



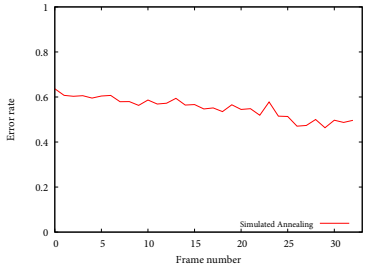
(a) Results test 1



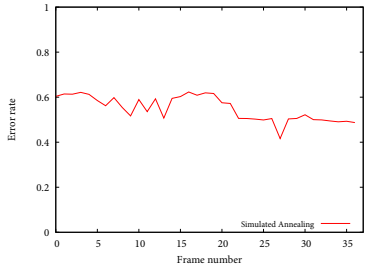
(b) Results test 2



(c) Results test 3



(d) Results test 4



(e) Results test 5

Figure 30: The final results for each recording.

The results of the tests are shown in Figures 30a, 30b, 30c, 30d, and 30e. As the results show, the algorithm never loses track of the hand. However, even though the average error value is lower than during the tests with the oblong, the visual match is unfortunately far from correct. An example of this problem is shown in Figures 29a and 29b. Increasing the maximum iterations to 10000 did not solve this problem so it is assumed that the search algorithm is not able to improve the result, even with more iterations. However, the result does not indicate the best visual match.

As a final remark, the number of iterations that can be done per 1/15th of a second is roughly 35 on test system 1. This is far from enough for a real-time pose estimation. Faster hardware would of course increase this, however, the implementation itself also leaves much room for improvement, since the current GPU is only used for roughly 40%.

4.4 SUMMARY

All search algorithms are capable of following a simple object like a ball. However, the Secant and Nelder-Mead method have difficulty to recover when the object is lost. Simulated Annealing performs better in this respect. Following an object like an oblong poses an additional problem for all algorithms. Three extra DOFs increase the necessary iterations, that are needed to sufficiently track the object, to 540 from 200. Simulated Annealing was used to perform the final test with the hand since Simulated Annealing performed better than the Nelder-Mead and Secant method. It showed that it could perform tracking reasonably well, however it is not very accurate.

5

CONCLUSION

At the beginning of this thesis several goals were set that the research will try to reach. The main goal was to provide a real-time hand pose estimation algorithm with reasonable accuracy that can be used in an AR environment. This would have to be done without resorting to gloves and other marking systems on the hand itself and it would have to be build from COTS hardware. A second goal was to provide automatic calibration so that the user would be able to get to work right away.

The real-time aspect of the goal has not been met. Even though the GPU proved to be relatively fast, it is quite difficult to use the full potential. Many improvements can be made in this area so that real-time estimation becomes possible. Unfortunately, it was not possible to implement these due to time constraints. The algorithm was able to follow the hand, however it estimated the angles of the fingers incorrectly. The hand tracker did not always give the correct result, although this does not completely explain the estimation problem. In several frames, the hand is not completely detected so that parts of the hand is missing. Also, the edges are not always shown because the pixels were not correctly labeled as hand pixels. The GUI of the ARMI project only uses three different poses. These poses are a pointing pose, a pinch pose where the index finger and the thumb are touch each other, and a pinch release pose. It might be possible that the algorithm is able to sufficiently estimate these poses when the DOFs are decreased so that only these three poses are matched. However, this was not tested due to insufficient time.

The system is entirely build out of COTS hardware. Even though the hardware used is relatively cheap, better (possibly more expensive) hardware should provide the algorithm with better input and more processing power. Especially the cameras are important since they supply the algorithm with the only input of the hand.

Unfortunately, automatic calibration could not be implemented due to time constraints. However, if the pose estimation is done accurately, it should also be possible to estimate the size of the hand correctly.

The system shows promise but much has to be improved first if it would be used in an AR environment where all fingers are needed for input. Even though the original idea was to support interactions with an architect and his or her customer, the system can be used for many other goals. Improved teleconferencing, easily access virtual 3D data and talk about it with multiple users. Virtual training surgery when the accuracy is sufficiently improved with the option to record and replay

the surgery to find possible mistakes. A virtual security system where camera input is augmented onto a 3D model of a building. Allowing the user to easily zoom in and move around the building.

6

FUTURE WORK

There are many parts of the system that can be improved but due to time constrains not all of them could be implemented. Some of them are related to increasing the performance of the error measurements, others are related to improving the search algorithm. Aside from improvements, the scientific world also makes progress in the field of augmented reality. The achievements that could have impact on a future implementation of the system are mentioned here.

6.1 IMPROVEMENTS TO THE ERROR MEASUREMENTS

There are several improvements possible to make the system work faster, possibly ending up with a system that can reach real-time performance. One of the first and probably one of the biggest improvements is the conversion from Python code to C code. All search algorithms are implemented completely in Python which is considerably slower than C. Due to time constraints it was not possible to implement everything in C code, however C code should be several times faster than Python code.

Another option would be to completely implement the search algorithm inside the GPU. The only communication between the CPU and GPU would be the activation of the FBO and shaders to execute the necessary calculations. A fixed number of iterations could be set beforehand, after which the system assumes that the search algorithm has found a good fit and passes it on to the interface to draw the model. The GPU implementation would roughly do the following steps:

1. The Sobel filter is applied to the video frame.

2. A Simulated Annealing pixel shader is run once. Or, in other words, an output buffer of one pixel is used to run the shader. It receives all the DOFs via multiple textures, one for every four DOFs. Each pixel can contain four different values, hence four DOFs for every texture. Due to the limitation of the pixel shader, it can only write to one pixel. With multiple color attachment buffers it becomes possible to write to multiple output textures. So every output texture also contains four different DOFs. Another extra color attachment is needed to store the detrimental steps and the current temperature.

3. A vertex shader is now executed which receives the DOFs from the Simulated Annealing pixel shader. It uses these DOFs to adjust

the model accordingly. To do this, each part of the model that has to be able to translate or rotate has a specific number. This specific number is specified as a color value. Each number corresponds to a DOF. The vertex is adjusted according to this DOF. This is done for the entire model. The pixel shader of the same render pass makes sure the entire model is rendered to a texture. This process is executed once for every camera. Every camera uses its own projection matrix so that the view from the real camera matches the view from the rendered model.

4. The Sobel filter is applied to both rendered models. After calculating the edges, the Sobel filter also subtracts the edges from the Sobel filtered video frame.

5. The total error from each of the subtracted images is summed together. The output of this step is given to the Simulated Annealing pixel shader.

Step 2 through 5 are repeated until the fixed number of iterations are performed. The DOFs will not even have to be read back to the CPU memory. The shader that is used to properly draw the model (step 3) can be used to draw the model for the interface as well. The only data that has to be uploaded to the GPU is the video frame, the bounding box of the object, and the vertex data for the model. Eliminating the need for the CPU to read the results back every iteration should increase performance substantially.

There are also a number of smaller improvements possible which were not implemented due to time constraints:

- Change the last step of the summation process so that the output is not written to two different textures but to one texture. This saves one extra read from the GPU memory.

- Most vertex data can be converted to vertex buffer objects (VBO). This could prove to be faster than the current implementation which uses display lists to contain all of its vertex data.

- Use a scatter operation for the Sobel and subtraction shaders. Or, in other words, use a vertex or geometry shader to change the polygons that determine which parts of the textures need to be processed. The polygons can be changed to fit the bounding box that are returned from the hand tracker. This is now done inside the pixel shader which is a relatively expensive operation since each pixel needs to check whether it is inside the bounding box. The vertex shader only needs to check this four times, one time for each corner of the bounding box.

- The Sobel shader that processes the model can be changed to immediately subtract the Sobel filtered video frame. This saves one extra shader and also reduces communication between the GPU and CPU.

- The shaders have not undergone much performance enhancements. It should be possible to decrease the number of operations by looking at the number of assembly instructions.

- The summation shader could calculate the sum of 16 pixels instead of four. This decreases the amount of communication between the GPU and CPU since only halve of the steps are needed to get to the end result.

- Make more use of 8 bit textures. Currently, the subtraction and summation shaders all make use of 32 bit floating point textures to maintain sufficient accuracy. However, this accuracy might not be needed in certain areas of the system. This could considerably increase performance of the pixel shaders since it is already shown that 32 bit textures are roughly two or three times slower than 8 bit textures (see Table 1).

6.2 IMPROVEMENTS TO THE SEARCH ALGORITHMS

The input for the search algorithm has much room for improvement. This in turn will give the search algorithm a chance to find a better fit or it might even converge faster. One of the most basic things that can be changed are the cameras that are used to capture the object. The webcams are cheap but they suffer from several problems. As mentioned earlier in this thesis, it is impossible to tell whether the frames are taken at the same moment. Another problem the webcams suffer from is that they have trouble capturing a moving object. The faster the object moves, the more stretched the object seems to be in the video frame. This has to do with the speed the webcam takes a single frame. If it were to capture the entire image in one single moment, then this problem would not happen. The cameras that can be synchronized with other cameras are usually able to capture a frame in one single moment. They are almost definitely more expensive than the webcams that are used at the moment, but they should give the system a lot better input data to work with.

Another form of input is the error which ultimately describes the fitness landscape. The function that combines both errors from both cameras are relatively simple. Better functions might exist that could better describe the fit so that the visual and actual error is more aligned with each other.

The hand model itself could be improved with color information. Also, a better approximation of the hand in the video feed should lead to a far better fit.

The search algorithm itself could be improved by adjusting the way the DOFs are adjusted every iteration. The current implementation adjusts the DOFs equally, where for instance the X position has the same amount of movement as the angle of a finger in one iteration.

More research should find the best way the DOFs are adjusted every iteration.

6.3 NEW RESEARCH IN AUGMENTED REALITY

Using the ARToolkit system to detect the tags displayed some problems that could arise when a user uses the system. One of the biggest problems was that the tag system that is used by ARToolkit has a serious flaw when it tries to detect a tag. Only a tiny area of the tag has to be occluded by some object, resulting in ARToolkit unable to detect the tag. This can be solved by using multiple tags, hoping that no situation occurs where all tags are occluded. However, other techniques like ARTag [2] are much more robust. A great part of the tag can be occluded and the system is still able to detect the tag. Apart from that, ARTag also uses a different tag system that encodes numbers in the tags with the use of white and black blocks inside the tag. This is especially useful when multiple tags are used. ARToolkit searches for a specific pattern. Each pattern requires a specific amount of processing power or processing time. So the more tags are used, the more time it takes for ARToolkit to detect all tags. Unlike ARToolkit, ARTag simply detects a tag that matches the general outline of a tag, after which it decodes the tag itself. Whether it searches for one or 15 tags makes no difference to ARTag [2].

However, if it were possible not using any tags at all, this would give the system a much greater robustness. Also, the user is able to use the system with one less item. Research done by Klein and Murray is able to achieve this [3]. Even though research is ongoing, it shows good promise replacing tag detection systems.

The research done by Lingley and Parviz is aimed at creating contact lenses for augmented reality [30]. Even though this research will probably not give any immediate practical results, it shows amazing promise in the field of augmented reality. They would be solar powered and see-through, unlike most augmented reality glasses at the moment. So no bulky glasses are needed anymore, but two contact lenses suffice. There would still be a need for a camera to make sure the augmented reality is aligned with reality, however, these are becoming smaller and smaller as well.

APPENDIX

A

SPECIFICATIONS TEST SYSTEMS

	System 1	System 2
System	Custom build	MacBook Pro
Operating system	Linux Ubuntu 9.04	Mac OS X 10.5.8
Kernel version	2.6.18	9.8.0
CPU	AMD Phenom II X4 940	Intel Core 2 Duo 2.4 Ghz P8600
CPU cores	4	2
GPU	AMD-ATI HD4870 512 MB	nVidia GeForce 9600M GT 256MB nVidia GeForce 9400M 256MB (onboard)
Motherboard	Asus M4A78 PRO	-
Hard drive	Samsung HD103SJ	Hitachi HTS723225L9SA62
Memory	2x 1GB, 2x 2GB DDR2 800 MHz	2x 2GB DDR3 1067 MHz

Table 8: Specifcation test systems.

BIBLIOGRAPHY

[1] *Articulated body motion capture by annealed particle filtering*, volume 2, August 2002. doi: 10.1109/CVPR.2000.854758. URL http://dx.doi.org/10.1109/CVPR.2000.854758.

[2] *ARTag, a fiducial marker system using digital techniques*, volume 2, 2005. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1467495.

[3] *Parallel Tracking and Mapping for Small AR Workspaces*, June 2008. doi: 10.1109/ISMAR.2007.4538852. URL http://dx.doi.org/10.1109/ISMAR.2007.4538852.

[4] Subutai Ahmad. A usable real-time 3d hand tracker. In *In Proceedings 28th Asilomar Conference on Signals, Systems and Computers*, pages 1257–1261. IEEE Computer Society Press, 1995.

[5] Vassilis Athitsos and Stan Sclaroff. Estimating 3d hand pose from a cluttered image. *CVPR*, 02:432, 2003. ISSN 1063-6919. doi: http://doi.ieeecomputersociety.org/10.1109/CVPR.2003.1211500.

[6] Ronald T. Azuma. A survey of augmented reality. *Presence*, 6, 1997.

[7] Matthieu Bray, Esther Koller-meier, and Luc Van Gool. Smart particle filtering for 3d hand tracking. In *In AFGR04*, pages 675–680, 2004.

[8] Matthieu Bray, Esther Koller-meier, Pascal Müller, Luc Van Gool, and Nicol N. Schraudolph. 3d hand tracking by rapid stochastic gradient descent using a skinning model. In *In 1st European Conference on Visual Media Production (CVMP)*, pages 59–68, 2004.

[9] C. G. Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation*, 19(92):577–593, October 1965.

[10] Pieter Bruining. Creating a three dimensional concurrent interface. Master's thesis, University of Groningen, 2009.

[11] Haiying Guan Chin-Seng Chua and Yeong-Khing Ho. Model-based 3d hand posture estimation from a single 2d image. *Image and Vision Computing*, 20:191–202(12), 1 March 2002. doi: doi:10.1016/S0262-8856(01)00094-4. URL http://www.ingentaconnect.com/content/els/02628856/2002/00000020/00000003/art00094.

[12] Benjamin Close. Interactive outdoor augmented reality collaboration system. URL http://wearables.unisa.edu.au/arquake/.

[13] Quentin Delamarre and Olivier Faugeras. Finding pose of hand in video images: a stereo-based approach. In *In IEEE Proc. of the third International Conference on Automatic Face and Gesture Recognition*, pages 585–590. IEEE Computer Society, 1998.

[14] Ali Erol, George Bebis, Mircea Nicolescu, Richard D. Boyle, and Xander Twombly. Vision-based hand pose estimation: A review. *Comput. Vis. Image Underst.*, 108(1-2):52–73, 2007. ISSN 1077-3142. doi: http://dx.doi.org/10.1016/j.cviu.2006.10.012.

[15] Maarten Fremouw. Real-time hand tracking using standard computer hardware. Master's thesis, University of Groningen, 2009.

[16] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 013168728X.

[17] V. Granville, M. Krivánek, and J.P. Rasson. Simulated annealing: A proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:652–656, 1994. ISSN 0162-8828. doi: http://doi.ieeecomputersociety.org/10.1109/34.295910.

[18] Tony Heap and David Hogg. Towards 3d hand tracking using a deformable model. In *In Face and Gesture Recognition*, pages 140–145, 1996.

[19] HowStuffWorks. How the first-down line works. URL http://www.howstuffworks.com/first-down-line.htm.

[20] Michael Isard and Andrew Blake. Condensation - conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29:5–28, 1998.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598: 671–680, 1983.

[22] Wolfgang Kruger, Christian a. Bohn, Bernd Frohlich, Heinrich Schuth, Wolfgang Strauss, and Gerold Wesche. The responsive workbench. *IEEE Computer Graphics and Applications*, 14:12–15, 1994.

[23] James J. Kuch and Thomas S. Huang. Vision based hand modeling and tracking for virtual teleconferencing and telecollaboration. In *ICCV*, pages 666–671, 1995. URL http://dblp.uni-trier.de/db/conf/iccv/iccv1995.html#KuchH95.

[24] Martin La and Gorce Nikos Paragios. Monocular hand pose estimation using variable metric gradient-descent, 2006.

[25] J. Lee and T. L. Kunii. Constraint-based hand animation. *Tokyo: Springerverlag*, pages 110–127, 1993.

[26] Heino Lenting. Replicating augmented reality objects for multi-user interactions. Master's thesis, University of Groningen, 2009.

[27] Cheng-Chang Lien. A scalable model-based hand posture analysis system. *Mach. Vis. Appl.*, 16(3):157–169, 2005.

[28] John Lin, Ying Wu, and T. S. Huang. Modeling the constraints of human hand motion. In *HUMO '00: Proceedings of the Workshop on Human Motion (HUMO'00)*, page 121, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0939-8.

[29] John Y. Lin, Ying Wu, and Thomas S. Huang. 3d model-based hand tracking using stochastic direct search method. *FG*, 0:693, 2004. doi: http://doi.ieeecomputersociety.org/10.1109/AFGR.2004.1301615.

[30] Andrew Lingley and Babak Parviz. Multipurpose integrated active contact lenses. 2008. doi: 10.2417/1200806.0056.

[31] Shan Lu and Dimitris Metaxas. Using multiple cues for hand tracking and model refinement. In *Proc. CVPR*, pages 443–450, 2003.

[32] Shahzad Malik. *Real-time hand tracking and finger tracking for interaction*. Department of Computer Science, Toronto University, Sandford Fleming Building, 10 Kings College Road, Room 3302, Toronto, Ontario M5S 3G4, 2002.

[33] C. Manresa, J. Varona, R. Mas, and F.J. Perales. Hand tracking and gesture recognition for human-computer interaction. *ELCVIA*, 5 (3):96–104, 2005.

[34] J.A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[35] Jitse Niesen. Illustration of the secant method. URL http://en.wikipedia.org/wiki/File:Secant_method.svg.

[36] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, August 1999. ISBN 0387987932. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0387987932.

[37] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. URL http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x.

[38] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-43108-5.

[39] James Rehg and Takeo Kanade. Visual tracking of high dof articulated structures: An application to human hand tracking. pages 35–46. 1994. doi: 10.1007/BFb0028333. URL http://dx.doi.org/10.1007/BFb0028333.

[40] Hans Rijpkema and Michael Girard. Computer animation of knowledge-based human grasping. *SIGGRAPH Comput. Graph.*, 25(4):339–348, 1991. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/127719.122754.

[41] Rómer Rosales, Vassilis Athitsos, Leonid Sigal, and Stan Sclaroff. 3d hand pose reconstruction using specialized mappings. *ICCV*, 01:378, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/ICCV.2001.10067.

[42] Markus Schlattmann, Ferenc Kahlesz, Ralf Sarlette, and Reinhard Klein. Markerless 4 gestures 6 dof real-time visual tracking of the human hand with automatic initialization. *Computer Graphics Forum*, 26:467–476(10), September 2007. doi: doi:10.1111/j.1467-8659.2007.01069.x. URL http://www.ingentaconnect.com/content/bpl/cgf/2007/00000026/00000003/art00027.

[43] Lisa Sherwin. Pci-sig delivers pci express 2.0 specification, 2007. URL http://www.pcisig.com/news_room/PCIe2_0_Spec_Release_FINAL2.pdf.

[44] B. Stenger, P. R. S. Mendonça, and R. Cipolla. Model-based 3d tracking of an articulated hand. *CVPR*, 2:310, 2001. ISSN 1063-6919. doi: http://doi.ieeecomputersociety.org/10.1109/CVPR.2001.990976.

[45] B. Stenger, P. R. S. Mendonça, and R. Cipolla. Model-based hand tracking using an unscented kalman filter. In *In Proc. British Machine Vision Conference, volume I*, pages 63–72, 2001.

[46] B. Stenger, A. Thayananthan, P. H. S. Torr, and R. Cipolla. Model-based hand tracking using a hierarchical bayesian filter. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(9):1372–1384, 2006. doi: 10.1109/TPAMI.2006.189. URL http://dx.doi.org/10.1109/TPAMI.2006.189.

[47] David Joel Sturman. *Whole-hand input*. PhD thesis, Cambridge, MA, USA, 1992.

[48] A. Torige and T. Kono. Human-interface by recognition of human gesture with imageprocessing-recognition of gesture to specify moving direction. *IEEE International Workshop*, pages 105–110, 1992.

[49] Pedro Trancoso and Maria Charalambous. Exploring graphics processor performance for general purpose applications. In *in 8th Euromicro Conference on Digital System Design (DSD*, pages 306–313, 2005.

[50] Unknown. Intel microprocessor export compliance metrics. URL http://www.intel.com/support/processors/xeon/sb/CS-020863.htm.

[51] Unknown. Ati radeon hd 5870 gpu feature summary, 2009. URL http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx.

[52] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, January 1985. doi: 10.1007/BF00940812. URL http://dx.doi.org/10.1007/BF00940812.

[53] Robert Y. Wang and Jovan Popović. Real-time hand-tracking with a color glove. *ACM Trans. Graph.*, 28(3):1–8, 2009. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1531326.1531369.

[54] T.D. White and P.A. Folkens. *Human Osteology*. Academic Press, 1991.

[55] Hanning Zhou and Thomas Huang. Okapi-chamfer matching for articulated object recognition. *ICCV*, 2:1026–1033, 2005. ISSN 1550-5499. doi: http://doi.ieeecomputersociety.org/10.1109/ICCV.2005.176.