Computer Science, Degree Project, Advanced Course,
15 Credits

A HAPTIC DEVICE INTERFACE FOR MEDICAL SIMULATIONS USING OPENCL

Mattias Machwirth

Computer Engineering Programme, 180 Credits

Örebro, Sweden, Spring 2013

Examiner: Lars Karlsson

Ett haptiskt gränssnitt för medicinska simuleringar med OpenCL

Örebro universitet
Institutionen för
naturvetenskap och teknik
701 82 Örebro

Örebro University
School of Science and Technology
SE-701 82 Örebro, Sweden

Abstract

The project evaluates how well a haptic device can be used to interact with a visualization of volumetric data. Since the interface to the haptic device require explicit surface descriptions, triangles had to be constructed from the volumetric data. The algorithm used to extract these triangles is marching cubes. The triangles produced by marching cubes are then transmitted to the haptic device to enable the force feedback. Marching cubes was suitable for parallelization and it was executed using OpenCL. Graphs in the report shows how this parallelization ran almost 70 times faster than the sequential CPU counterpart of the same algorithm.
Further development of the project would give medical students the opportunity to practice difficult procedures on a simulation instead of a real patient. This would give a realistic and accurate simulation to practice on.

Sammanfattning

Projektet går ut på att utvärdera hur väl en haptisk utrustning går att använda för att interagera med en visualisering av volymetrisk data. Eftersom haptikutrustningen krävde explicit beskrivna ytor, krävdes först en triangelgenerering utifrån den volymetriska datan. Algoritmen som används till detta är marching cubes. Trianglarna som producerades med hjälp av marching cubes skickas sedan vidare till den haptiska utrustningen för att kunna få gensvar i form av krafter för att utnyttja sig av känsel och inte bara syn. Eftersom marching cubes lämpas för en parallelisering användes OpenCL för att snabba upp algoritmen. Grafer i projektet visar hur denna algoritm exekveras upp emot 70 gånger snabbare när algoritmen körs som en kernel i OpenCL istället för sekvensiellt på CPUn. Tanken är att när vidareutveckling av projektet har gjorts i god mån, kan detta användas av läkarstuderande där övning av svåra snitt kan ske i en verklighetstrogen simulering innan samma ingrepp utförs på en individ.

Preface

I would like to thank Science Park, Karlskoga, for letting me use their laboratory and their haptic device whenever I needed. I would also like to thank Mathias Broxvall, who provided great ideas of implementation and also helped me understand some of the more complicated algorithms used. I found the project to be very interesting and right in time.

Contents

1 INTRODUCTION....5
1.1 BACKGROUND....5
1.1.1 Previous work....5
1.2 PROJECT....6
1.3 OBJECTIVE....7
1.4 REQUIREMENTS....7
2 METHODS AND TOOLS....8
2.1 EXTERNAL LIBRARIES....8
2.1.1 HAPI....8
2.1.2 OpenCL....9
2.1.2.1 Global memory / Constant memory....9
2.1.2.2 Local memory....9
2.1.2.3 Private memory....10
2.1.3 Boost....10
2.1.4 GLM....11
2.2 ALGORITHMS....12
2.2.1 Marching cubes....12
2.2.2 Modified reduction....12
2.3 TOOLS AND LANGUAGES....14
2.3.1 Programming....14
2.3.2 Profiling....14
2.3.3 Software....14
2.4 OTHER RESOURCES....14
3 IMPLEMENTATION....15
3.1 COMMUNICATION....15
3.1.1 Forcing EPP-mode....15
3.1.2 Backup system....15
3.2 SYNCHRONIZING COORDINATE SYSTEMS....16
3.3 MESH GENERATION....17
3.3.1 Point cloud....17
3.3.2 Cubes....17
3.3.3 Marching cubes....18
3.4 DRAMATIC SPEED UP BY USING OPENCL....21
3.5 HAPTIC TRANSFER....22
3.5.1 Numerical precision problems....22
3.6 MEMORY CONSTRAINTS....23
3.6.1 Downsampling the dataset....24
3.6.2 Multipass kernel execution....25
3.6.3 Recount the triangles....25
3.6.4 Precalculated normals and reuse of vertices....26
4 RESULT....29
4.1 TESTING....29
4.1.1 Qualitative testing....29
4.1.2 Quantitative testing....32
5 DISCUSSION....34
5.1 ETHICAL ASPECTS....34
5.2 SOCIO-ECONOMICAL CONSIDERATIONS....34
5.3 ACHIEVEMENT OF THE COURSE OBJECTIVES....34
5.4 COMPLIANCE WITH THE PROJECT REQUIREMENTS....35
5.5 PROJECT DEVELOPMENT....35
6 REFERENCES....36

APPENDICES

A: Driver rewrite to force EPP-mode (parport_pc.c)

1 Introduction

In this brief chapter the project as well as some previous work is described in detail. The project requirements are presented for an overview of what was supposed to be included in the project.

1.1 Background

Science Park is a part of Campus Alfred Nobel in Karlskoga, Sweden. A lot of high-tech equipment is stationed here including 3D-printers, a supercomputer and a haptic device. The project was carried out on behalf of the Centre for Applied Autonomous Sensor Systems (AASS).

A haptic device is an actuated device, often a robotic arm, containing a tool capable of being manipulated by the user and provide force feedback to the user depending on the position of the tool. Such haptic devices typically allow the tool to be manipulated in an limited 3D space defined as the workspace of the arm. The device also tracks this tool at all times. The internal representation of this tool is either a point or a sphere, as described later, and this representation is called the probe.

The main idea for the project was to take medical ultrasound volumes, visualize a surface of the volume, and then use a haptic device to interact with that visualization. The haptic device should then provide force-feedback in a realistic fashion.

1.1.1 Previous work

There has been similar research about the different steps preformed. To generate a polygonal mesh from a 3D dataset marching cubes was presented in 1987 [1]. The technique is described later in the report.

Archirapatkave et al. describes an accelerated method to generate this mesh and also compares the speedups from the generation on GPU instead of the CPU [2]. Another report that shows the benefit of executing marching cubes on the GPU is published by Smistad et al. [9]. This report shows how to do marching cubes on large datasets by using a technique called histogram pyramids. These pyramids are built up by 3D-textures and are exploiting the GPUs texture caching abilities for really impressive times on large datasets.

Karadogan et al. describes how medical students learn to diagnose different disorders by using a modified haptic device [3].

Another interesting publication is published by Mizakami et al. where haptics is used for intracytoplasmic sperm injection (ICSI) simulation [7]. This describes the problematic nature of fertilizing an ovum without destructing the cell structure when doing the procedure under normal circumstances using micro-manipulation devices. If the cell structure gets destructed during the process several different kinds of issues can appear later on, including miscarriage. Even though Mizakami's paper does not use volumetric data it shows the benefit of a haptic simulation.

1.2 Project

The first part of the project consist of generating a mesh surface out of a 3D image. The actual 3D image, or dataset, consist of a X*Y*Z grid filled with intensities. These intensities goes from 0.0 to 1.0 where a higher number indicate a greater intensity.

The exact meaning of intensity is dependent on the modality used. These modalities can be computed tomography (CT), magnetic resonance imaging (MRI) or many others. These different techniques calculate the intensity in different ways. For example MRI uses magnetism and radio waves to calculate the intensity. The procedure starts with aligning the protons of the hydrogen atoms by using a large magnet. When the protons are aligned, they get exposed to radio waves which makes the protons to spin. Since this spin is detectable by the MRI-equipment, an image can be constructed [14]. For the project described in this report we capture an ultrasound dataset for the visualization which have the advantage of being relatively cheap and non-invasive.

The dataset is then used to construct a surface that later is transformed to the haptic device. The generation of the surface is dependent on a threshold which the user chooses, and this threshold corresponds to the intensities of the dataset. The software then iterates over the dataset and extracts points over this threshold.

The next step is to render this surface at both the haptic device and the computer, meaning that a visualization is produced on the computer, and a triangle constructed object is sent to the haptic device. If the rendering only occurred on the haptic side, then the force-feedback would work like expected but the user would not know where the haptic device was positioned or how the “haptic-space” looked like. This is somewhat similar to how a blind man perceives the world.

The actual force is dependent on the surface material of the object and if any external force effects are added to the scene. The probe is also rendered, much like a mouse pointer, to provide a visual aid for the user.

The information flow in the project is shown in figure 1.

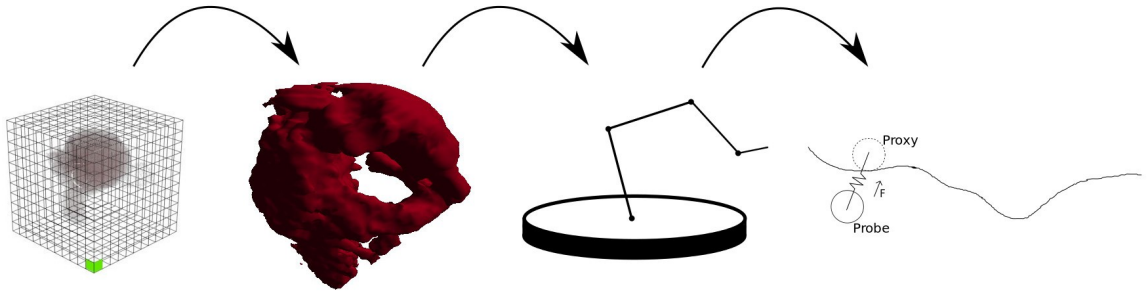


Figure 1: The data flow. Dataset -> polygonial mesh -> haptic transfer -> force feedback

1.3 Objective

The objective with the project was to be able to visualize volumetric data and interact with it in a realistic fashion. The realism was provided by the haptic device, which gave a force-feedback-effect when the probe interacted with the surface of the visualization.

An example of use would be to give medical students an opportunity to practice the interaction with the internal organs in a fully simulated way. In the case of a mishap the student would learn a lesson without jeopardizing the well-being of an actual patient.

Some diseases are very rare and training on these patients are hard to do. By simulating the same disease this problem disappears. Although this sounds very appealing, the simulation must be very lifelike else the training probably would do more harm than good, as described in the discussion chapter.

1.4 Requirements

When the project started certain requirements were set. There was two different kind of requirements, hard and soft. Hard ones had to succeed while soft ones only were to be considered if the time allowed and the hard ones already were done.

The hard ones were:

- Generate polygons from the extracted points of the 3D ultrasound

  The program should be able to take a dataset and extract the points over a certain threshold. These points were later used to build up graphic primitives.

- Interaction with the volumetric data extracted from the 3D ultrasound

  The extracted data should then be rendered on both the screen and on the haptic device. The screen should provide a visual feedback of what happened in the simulation. Additionally the haptic device provided force-feedback when the probe collided with the surface of the extracted data.

- Using H3D to handle the interface to the haptic device

  An open source library for haptic rendering and communication, H3D, is available and the goal was to use this for the haptic side of the project [4].

- Use OpenCL on computations that can be made in parallel

  To speed up the simulation a requirement was to use OpenCL on those computations that could be made in parallel.

The soft ones were:

- Use biosim to provide spring-damper feedback at the surface of the visualization

  Emil Eriksson, made a program in 2012 that simulates volumetric data with a mass-spring-damper model [10]. This project models all the points in the dataset to be connected with the neighboring points. This makes it possible to simulate tissue and other elastic materials. If the time allowed an integration with biosim was considered. This would add realistic physics to the tissue and if that succeeded the force-feedback would be much more lifelike.

2 Methods and tools

The project rely on existing tools and algorithms, and these are described in this chapter. The project was also highly dependent of various libraries like OpenGL, OpenCL and HAPI. This chapter will present these libraries in a short and concise manner.

2.1 External libraries

For the project to accomplish its requirements certain libraries were used for the different tasks. For the graphics OpenGL was used. This gave the project a graphic renderer which were platform independent. For the window creation SDL was used. Since the high parallelism factor of the marching cubes algorithm OpenCL was used to produce the vertices.

For the haptic rendering HAPI was used. HAPI is a part of the H3DAPI and it's responsible for the rendering part of the haptic device.

Boost was also used for the parts of the code which handled shared memory, smart pointers and parsing the command line.

2.1.1 HAPI

HAPI is a part of the H3DAPI library which is a library made for haptic interaction [4]. To be able to feel forces and to communicate with the haptic device this library was used in the project.

HAPI contains classes for the objects rendered at the device including surfaces, force effects, devices and renderers. A surface class in HAPI describes how the outer surface of the rendered object should feel and it is dependent on variables like friction, stiffness, etc. Force effects classes simulate forces without a collision by a rendered object. Applications that simulates magnetic fields or gravity might find these effects useful. The device class keeps track of the position and rotation of the probe as well as the current button status.

The generated force is calculated using a proxy object. This object always stays on the surface of the visualization, even if the probe has penetrated the surface. Similarly when the probe moves inside the object, the proxy moves on the surface of it. The force generated is dependent of the distance between the probe and the proxy and it tries to push the probe towards the proxy as shown in figure 2 [4].

The library also consist of two integrated renderers and two external renderers. Three of these renderers models the proxy as a point while the last one models it as a sphere. The sphere modelling was discovered to be an important part later in the project because of numerical precision problems.

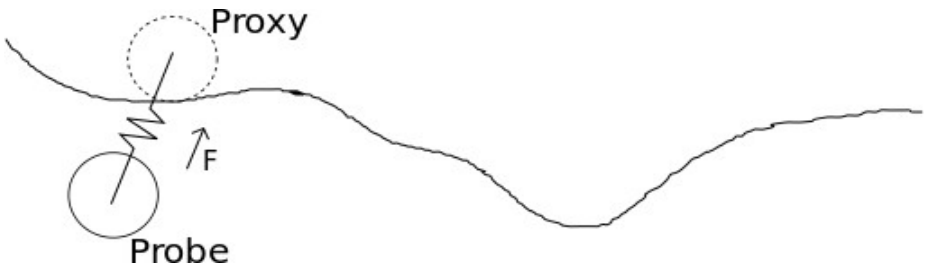


Figure 2: The probe is pushed towards the proxy. The distance determines the actual force applied.

2.1.2 OpenCL

OpenCL is an API developed by Khronos Group for creating programs that runs in parallel on different kind of devices [6]. These devices are often graphic cards but can also be CPUs or accelerators like blade servers. This also means that for those who has more than one OpenCL device the workload can be spread to multiple devices.

A program in OpenCL is a construct that holds kernels, and kernels can be thought of like C-functions that are sent to the device for execution in parallel. The kernels are programmed in a language that is a subset of C99 with additional data types and functions. These kernels excel at vector processing but are not very fast at branching statements, like if or for.

The kernel execution inside OpenCL is done by so called work items. These work items can be thought of as threads executing in parallel. Furthermore these work items are collected in work groups. The maximum number of work items in a work group is device dependent but usually some multiple of 256. This value can be requested with the following:

```
size_t items;
clGetDeviceInfo(device,CL_DEVICE_MAX_WORK_GROUP_SIZE,
                sizeof(items),&items,nullptr);
```

The OpenCL memory model consist of four different memory spaces, where each has a different size and transfer speed.

2.1.2.1 Global memory / Constant memory

This is by far the greatest chunk of memory available and it is also the slowest. This is the place where arguments sent from the host are stored, and where the calculated kernel data is stored. All work groups can reach this memory but since the speed is the slowest work groups usually only access this memory two times, one time to copy the work groups data to local memory and the second time to store the data that was calculated inside the work group. There can be a separate area for the constant memory but mostly it is the global memory with the constraint that it is read-only.

2.1.2.2 Local memory

This area is an up to 100 times faster memory area that is shared with the work group [5:p88]. Because of the speedup it's wise to copy data needed by the work group from the global memory area to the local memory area, and then use this local memory in the upcoming calculations. This is accomplished with the following lines of code:

```
int lid = get_local_id(0);
int gid = get_group_id(0);
int nrGroups = get_num_groups(0);
__local lData[256];
lData[lid] = gData[gid*nrGroups + lid];
barrier(CLK_LOCAL_MEM_FENCE);
```

The barrier command is used to stop the execution until all work items has executed the barrier. This is needed because of the nature threads work. There is no way for the host to directly send its argument to local memory, therefore the data must be copied like above.

2.1.2.3 Private memory

This part of memory is very small but also the fastest. Variables without qualifiers like `lid`, `gid` and `nGroups` above are all stored in the private memory. Each work item has access to its own private memory.

The devices also has a number of physical units called compute units. On each compute unit a work group can execute in parallel.

OpenCL places the devices in a context and ties a command queue to each device. To make $device_n$ execute $kernel_n$, a command has to be sent to the command queue tied to $device_n$, telling this device to execute $kernel_n$. When the kernel is done executing, it probably has calculated something and stored it in the global memory. Then it is up to the user to fetch the data by sending another command on the queue, this time to read data. The queue direction is always one-way, host to device.

Scarpino talks about the success Java have had because of the platform independency and their motto ”Write once, run everywhere”. He later forms an own OpenCL motto ”Write once, run on anything” [5].

The curious reader might wonder why to use OpenCL to execute a kernel when OpenGL also is able to process data on a dedicated number cruncher like the GPU. Scarpino mentions three advantages by using OpenCL [5]:

- Kernels can invoke more functions
- Kernels reach local and private memory for faster computations
- Kernels can be synchronized in work groups

For example a vertex shader can only work with one vertex at a time where the OpenCL kernel reaches all the data on the device and synchronize the computations on that data. But furthermore the algorithm was much more suited for OpenCL.

For problems that are able to execute in parallel this library provides significant speedups compared to the same problem executed in sequence on a CPU. Although not all problems can be modelled in parallel. Problems that depend on recursion or previous calculations like an A* search is hard to parallelize.

2.1.3 Boost

Certain parts of the code used the well known Boost library [11]. Boost is a very wide library developed by dedicated individuals, where many of them are members of the C++ standard committee, and parts of boosts functionality has been added to the standard of the latest C++-version, C++11 [8]. Many of boosts libraries are header only libraries, which means that no separate compilation for the library is needed.

The project used three parts of the boost library:

- Program options
- Interprocess
- Smart ptr

Program options give the user a very effective and fast way of parsing the command line. This

made it easy to add certain software flags that the user typed at the command prompt when starting the program. Alternatives to this library is to use getopt instead, but using program options made the code cleaner and much shorter. This is also the only Boost library used that needed separate compilation.

Interprocess was used to create shared memory regions between processes. This was needed for the backup system explained later. Alternatives could be the function

```
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

but the use of the boost equivalent made more sense and made the shared memory handling easier.

Smart pointers was also used at certain parts of the code. The smart pointer is a construct that handles a pointer and more importantly releases the memory occupied by a pointer at destruction. Using correctly, this construct frees the programmer for ever typing an explicit delete, which removes the probability of memory leaks.

2.1.4 GLM

Boost has a brilliant math library as well but the project used the GLM library for the math instead because of the close connection to the OpenGL types and constructs. The library is a header only library used for the linear algebra part of OpenGL as well as holding data types that corresponds to their equivalent in GLSL [12].

2.2 Algorithms

2.2.1 Marching cubes

To make the visualization realistic marching cubes was chosen to generate the mesh. Marching cubes is a technique to extract an polygonal mesh out of a three-dimensional scalar field and is a great method for polygon based graphics [1].

2.2.2 Modified reduction

The regular reduction algorithm uses parallelism to sum the elements of a vector with N elements in $\log_2(N)$ steps. This algorithm loads a subset of the input data to the local memory of the work group where each work item loads a unique index. After this the local memory is processed by $\frac{N}{2^{i+1}}$ work items every iteration of the loop. Every work item fetches the value $\frac{N}{2^{i+1}}$ to the right and adds it to its own value. When the algorithm is complete a sum is calculated for all the elements of the local memory in the work group and the sum is stored at the first element.

A kernel implementation could look something like this:

```
int lid = get_local_id(0);
for(int i = N/2; i > 0; i >>= 1){
     if(lid < i)
          array[lid] += array[lid + i];
     barrier(CLK_LOCAL_MEM_FENCE);
}
```

All steps for an example array are shown in figure 3.

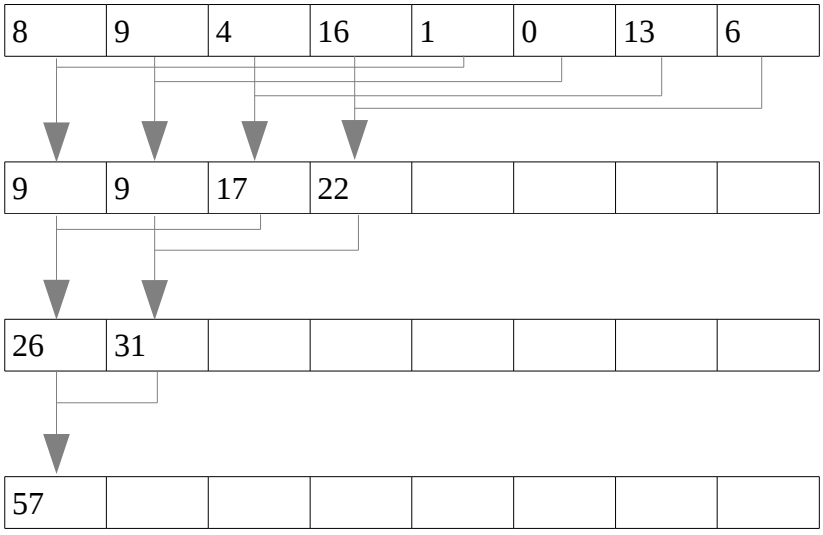


Figure 3: Reduction algorithm. Adding the eight elements in an array.

The project used a modified form of this algorithm. In this modified form the algorithm instead computes the sum of the elements to the left of the element in question. This changes the algorithm from calculating a scalar to recalculate the full array. The point of having the array where each element holds the sum of the elements to the left is that it can be used as an offset-indexer when data needs to be placed in a large one dimensional array. After this calculation is complete every element subtracts its own initial value in order to compute the offset in a buffer where the output should be written. This guarantees that the elements are placed adjacently (and non-overlapping) to each other.

An example kernel for this modification is shown below

```
int lid = get_local_id(0);
for(int i = 0; i < log2(N); i++){
     if((lid - (1 << i)) >= 0)
          array[lid] += array[lid - (1 << i)];
     barrier(CLK_LOCAL_MEM_FENCE);
}
```

Figure 4 shows the behavior of this algorithm.

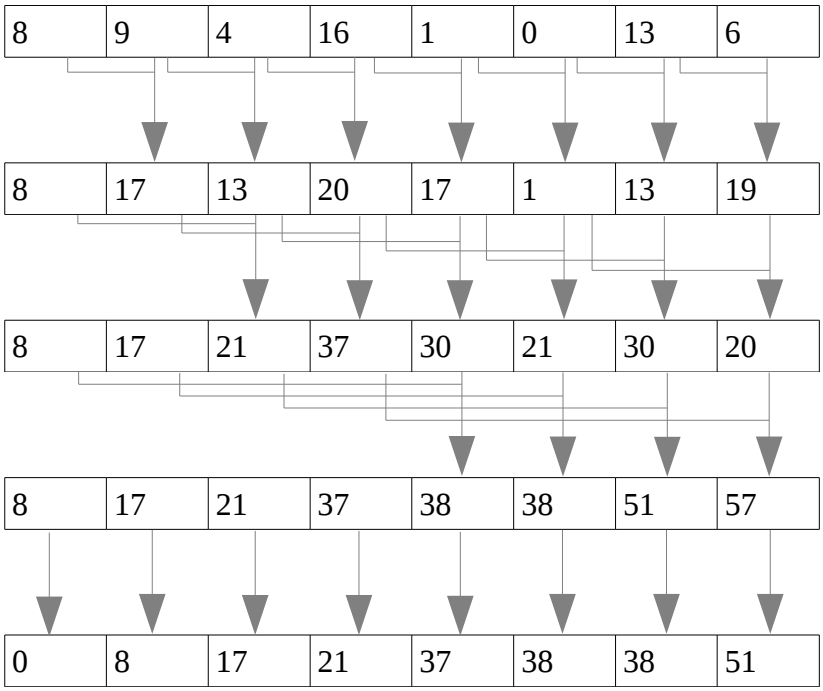


Figure 4: Modified reduction. The algorithm now calculates the sum of the elements to the left. A last step is a subtraction with its own value. This makes it suitable for indexing.

Now the array holds the offsets from the start depending on what is modelled. In the project the array held the offset of the number of triangles. Two of these arrays were generated, one for a local offset in a group and one for a global offset.

When the vertices then were computed in OpenCL it was possible to know where to place each work items triangles by using the two offset-indexers calculated above.

2.3 Tools and languages

2.3.1 Programming

C++ was used as programming language because of its efficiency and that the libraries above worked well with it. The ability to manipulate memory also was a requirement and C++ offered this. Some constructs in the project uses C++11 for shorter and more readable code.

2.3.2 Profiling

To be able to determine the bottlenecks and general time consuming operations a self-made class was developed to keep track of the execution time of certain parts of the C++-code. The class is very simple, and it holds functions to start and stop the timer in question. The class also holds a static data structure containing a sorted multiset of the times measured for each timer. This list implicitly holds the number of times the timer has executed and therefore the list can be used to calculate both the average time measured as well as the median.
When the average is calculated the standard deviation could be estimated as well.

When displaying the resulting times, these three measurements are shown.

One reason why a self developed class was made was to easily be able to chose which parts of the code to time. For example to measure the execution time of arbitrary lines of code:

```
Timer t(“Measure complicatedFunction”);
t.start();
int startValue = middleValue(x);
for(int i = 0; i < maxIterations; i++){
     calculateForce(complicatedFunction(startValue + i));
}
t.stop();
```

This would measure the time from start to stop one time. If the start and stop functions on the other hand were moved inside the loop the measure would happen maxIterations times.

For the profiling in OpenCL both AMDs profiling utility, CodeXL , and the cl_event data structures were used. These were very valuable in the comparison of CPU vs GPU performance of the mesh generators.

2.3.3 Software

Linux was used as operating system for the project. The programming was made in Vim and the compilation occurred in g++.

Git was used as version control for the project. Git is a freely available open source distributed version control system [13].

2.4 Other resources

Requirements for the project were a haptic device, a computer with a powerful GPU, and volumetric data. The haptic device used was a Phantom Premium 1.5/6DOF. This device has six degrees of freedom and is connected to the computer by a parallel port.

The haptic device was already stationed at Science Park and Mathias Broxvall provided the computer and the datasets.

3 Implementation

This chapter will describe the steps preformed from reading a dataset, to interacting with the visualization. Some problems that occurred are also brought to light and means to tackle these problems are presented.

As explained earlier the project consist of a computer and a haptic-device, where the computer generates triangles out of a dataset, and sends these to the haptic-device. When the performer then uses the haptic-device and collides with any of the triangles, it provides force-feedback.

3.1 Communication

The communication between the haptic device and the computer was handled by the parallel port. Since this is an old interface an external card had to be used because the motherboard on the computer used didn't provide this. This port can use several modes of bidirectional communication including EPP and ECP.

The haptic device needed the port to be in EPP-mode for the communication to succeed. Since EPP is older than ECP most new cards automatically chose ECP as mode of transfer, but this was incompatible with the haptic device. Since the parallel port driver, parport_pc.c, checks if ECP is present and then automatically chooses this mode of transfer, the card used was always initiated with the ECP transfer mode.

3.1.1 Forcing EPP-mode

Problem arose when the external card automatically chose ECP as mode of transfer. This caused the communication to fail since the device needed EPP.
The solution was to rewrite the Linux driver parport_pc.c, and this rewrite is found in the appendix of the report.

This made the program able to start but unfortunately timeouts happened on a semi-random basis, and these timeouts caused the program to terminate. This led to the construction of a backup system.

3.1.2 Backup system

To overcome the timeout problem a dual process solution was created. This made two processes where one process was in charge of the haptic rendering and the other was responsible for everything else. Lets call these H and R for the rest of the paragraph. This was the one part of the code where the shared memory constructs from Boost were needed. When R produced its vertices it stored these in the shared memory. This made it possible for H to find the vertices and send the appropriate triangles to the haptics system. In the H process a function was tied to the interruption of the process flow, by the atexit function. This function reached another part of shared memory and set a variable indicating that the program flow for H has been interrupted. This shared memory was checked in process R every iteration of its main loop. If the variable was set process R knew that the other process has failed. Process R now reset the shared memory and made a new fork to restart the haptic rendering.
This construct made it possible to continue executing the program even though the haptics had timed out. Some other shared memory object were used as well for the interleaving of the processes. The idea is shown in figure 5.

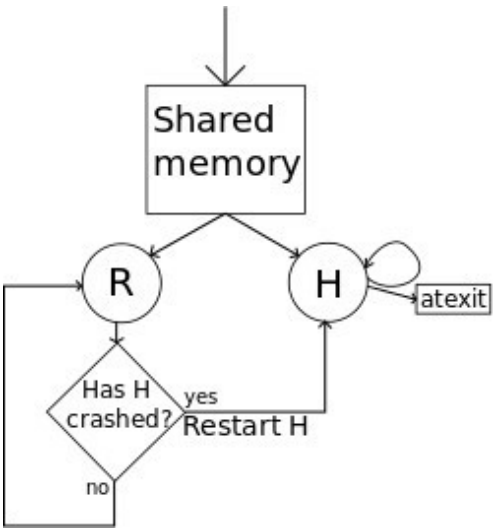


Figure 5: The shared memory is reachable by both processes. If R detects that H has crashed, R restart H.

This is a highly temporary solution that would not be necessary if the connection between the computer and the haptic device worked as expected. Recent models of the haptic device uses other forms of communication including FireWire. There is a high probability that these modes of transfer would succeed without this modification, although the project did not have a budget to buy one of these new devices.

Most new computers does not have a parallel port and this has a major implication of what simulations the haptic device can provide. By going back to an older computer that has a parallel port attached to the motherboard the communication might succeed, but then the computational power to use heavy algorithms like marching cubes would go much slower, if at all.

3.2 Synchronizing coordinate systems

Since the project consist of two different coordinate systems that has nothing to do with each other, the next step was to synchronize these. If a synchronization was done it would mean that an object spawned at the same position in both OpenGL and in H3D the visualization would be correct. By making sure that the origin at the haptic device is located near the origin at the OpenGL coordinate system, the both coordinates will define the same location. This is done by either sending a calibration matrix to the haptic device, or to restart the haptic device. In the later scenario the origin will be the probes location after the restart.

3.3 Mesh generation

The next step in the process was to read in the dataset and extract vertices depending on the threshold chosen. This makes it possible to visualize these vertices and tie them to certain OpenGL primitives. All of the below generators produce a normalized 1^3 set independent of what dimensions the original set has. This means that larger sets create higher triangle density that smaller sets. The different methods available in the software are explained below.

3.3.1 Point cloud

This is the fastest method by far and also the one method that does not generate a mesh. By just iterating through the dataset in three nested for-loops, every vertex that has a equal or higher threshold is saved. This is visualized using GL_POINTS and is shown in screen 1.

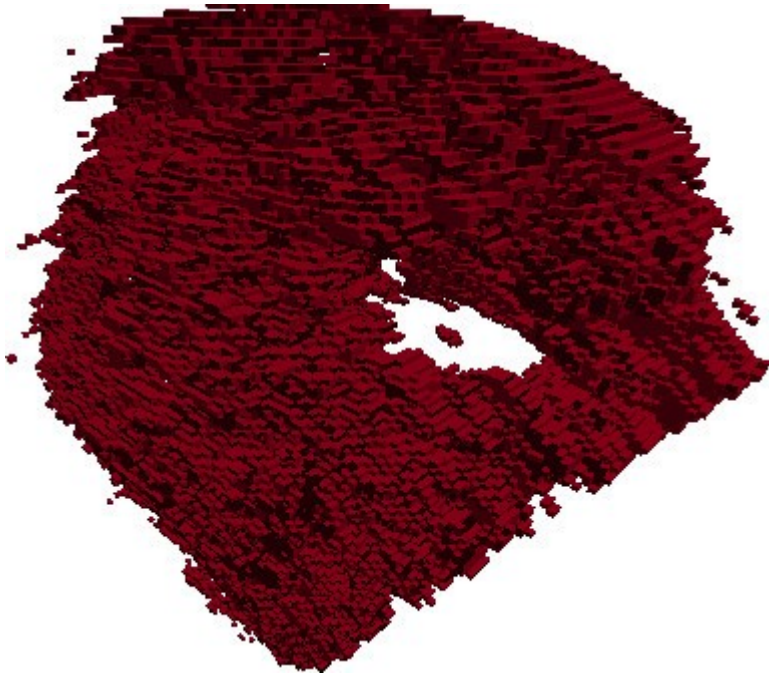
The drawback with this method is that it doesn't generate a mesh and that the visualization lack 3D-factor. Most of the renderers available to the haptic device cannot feel points or lines so this is also an indirect drawback of this generator.

Screen 1: The pointcloud generator

3.3.2 Cubes

This is similar to the point cloud but instead of saving each vertex over the threshold it generates a cube at that vertex. This gives a first impression of 3D and it is also fast.
The outcome of this method produces is shown in screen 2.

The OpenGL primitives here is GL_QUADS. Although this method is available for rendering on the haptics device, the realism wouldn't be of much use since the visualized object would have a “LEGO-like” structure, nothing like the actual organ from the dataset.



Screen 2: The cube generator

3.3.3 Marching cubes

This is the main method used to generate the mesh and it is also the most computational expensive even though it mostly depends on lookup tables. Lorensen et al. presented this algorithm in 1987 [1].

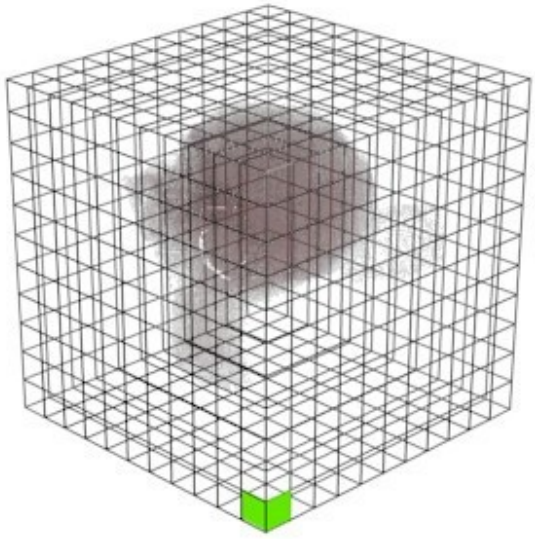


Figure 6: The dataset and its sub-cubes.
Every corner of each sub-cube is assigned an intesity value between 0.0 and 1.0.

The algorithm iterates over the dataset using cubes made of the voxel values at the eight vertex positions of the cube, as shown in figure 6. Each of these cube vertices is then assigned a zero or one, if that vertex is above resp. below the threshold. This information is saved in a bit field since the only options are zero or one. The next step is to see which edges this cube intersect by using the previous bit field as an index in a look-up table.

Since a vertex is either above or below the threshold, this creates a possible outcome of $2^8 = 256$ edge intersections. Although by using complements and rotational symmetry Lorensen et al. shows that the possible outcomes of the intersections can be dramatically decreased to 15 different patterns [1]. One complementary case is shown in figure 7.

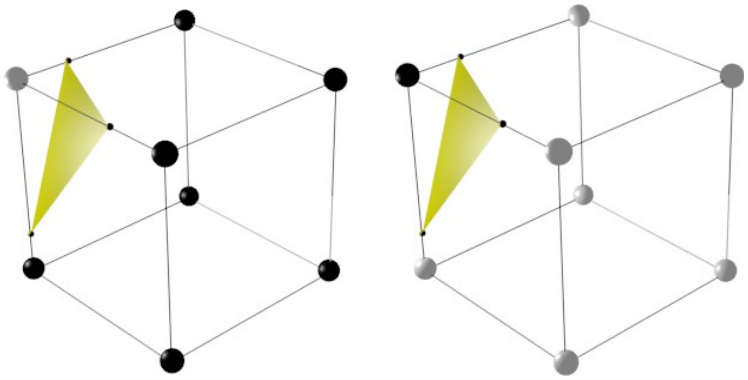


Figure 7: Complementary cases. One cube has seven vertices outside and one inside, while the other is the complementary version. Both uses the same edges for the interpolation.

Indexing with this bit field will return a 12 bit number where each set bit is representing an edge that is intersected.

Now the interesting part is if one vertex is assigned a one, while a neighboring is assigned a zero. This means that one vertex is above while the other is below the threshold. This implies that the mesh stored in the dataset intersects the edge between these vertices.

The actual intersection point is determined using linear interpolation between the two vertices. An interpolation between P_1 and P_2 is shown in figure 8. The interpolation needs to know the vertex values, V_1 and V_2, at those positions and then find the threshold point somewhere in between.

$$thresh = [0...1]$$
$$\vec{v} = \vec{P}_2 - \vec{P}_1$$
$$V_1 = vVal(\vec{P}_1)$$
$$V_2 = vVal(\vec{P}_2)$$

$$\lambda = \frac{(thresh - V_1)}{(V_2 - V_1)}$$
$$\vec{P}_{intersect} = \vec{P}_1 + \lambda * (\vec{v})$$

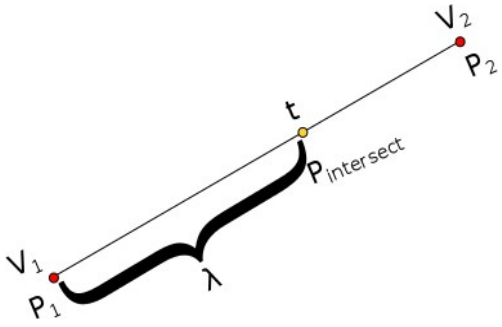


Figure 8: Interpolation of a point. The point ends up somewhere between the two corners of the cube.

In other words it finds the ratio where the threshold is between P_1 and P_2 and then multiplies the direction vector with this ratio and adds this to P_1 to find the point. This is repeated for all edges that has a zero-one relationship and when all cubes are processed this produces all the triangles needed for the mesh.

The next step in the algorithm is to find the normals at each point. This is calculated using the gradient at both vertices connected to the edge in question. The normal is then interpolated to the correct position. This procedure will produce weighted normals. The gradient calculation is shown in equation 1.

$$\nabla \vec{f} = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \approx \left(\frac{d(x+1,y,z) - d(x-1,y,z)}{\Delta x}, \frac{d(x,y+1,z) - d(x,y-1,z)}{\Delta y}, \frac{d(x,y,z+1) - d(x,y,z-1)}{\Delta z} \right) \quad (1)$$

$d(x,y,z)$ equals the intensity at point x,y,z. A final outcome of the algorithm is shown in screen 3.

If the shading isn't important the normals can also be calculated using the cross-product between two non-parallel vectors constructed from the three vertices of each triangle. This technique is faster than the interpolated normals but it wouldn't give a very realistic visual appearance. This would generate flat shading and every triangle would have the same color.

Of course the haptic feedback would not change since the triangles are independent of the shading used.

Screen 3: The marching cubes generator

3.4 Dramatic speed up by using OpenCL

The marching cubes algorithm was implemented both on the CPU and as an OpenCL kernel executed on the GPU. The nature of the algorithm makes it highly parallelizable because none of the cubes constructed from voxels are dependent on any of the other cubes.

By running the algorithm on the CPU all cubes are run one at a time in sequence. This can be optimized a number of times by utilizing OpenMP or other threading APIs but the need for this was nonexistent since the algorithm successfully ran on OpenCL.

One problem in the OpenCL part was that there was no way to know how many triangles that each dataset at a certain threshold would generate. This is a problem since OpenCL doesn't have dynamic memory allocation or container classes. Therefore the problem was divided into two kernels; counting and marching.

The counting kernel took the dataset and threshold as input parameters and gave two arrays of offsets as output parameters. The job was simply to count all the triangles in each cube. This value was then stored in a local offset array and when all work items stored their triangle count in this array it was reduced using the modified algorithm above. This local offset array now held the correct offset in the group. The size of this array was equal to the size of the number of work items.

A second offset array was needed and that was the global offset array. This array was as large as the number of work groups. The job for this array was to store the global offset for each work group. This lines to implement this were:

```
if(lid == 255)
     globalOffsets[get_group_id(0)] = groupTriangles[255] +
     numberOfOwnTriangles;
```

Since the reduced algorithm subtracts the own triangles this has to be added again to the global offsets.

Now the value in the last element of globalOffsets held the space needed for the triangles. When this space was known it was allocated and the second kernel was launched. This kernel was the marching cubes algorithm. This had the dataset, the local offset array, the global offset array and the threshold as inputs and the allocated array for the triangles as output.

Now the algorithm basically could run just like the CPU counterpart of the algorithm but care had to be taken where to put the computed triangles. This was a matter of indexation and this was done like this:

```
complete[( globalOffsets[groupId] + localOffsets[groupId*256 +
lid)*18 + i*6] = vertex1
```

This takes the global offset plus the local offset and then multiplies this number by 18. It is 18 because a triangle consist of 9 vertices, and vertices and normals are interleaved in the complete array.

The actual speed ups from OpenCL are shown in the result section.

3.5 Haptic transfer

When the mesh was generated it was time to send the shape to the haptic device. This was accomplished by making a tree of axis aligned bounding boxes that consisted of all triangles that made up the surface. HAPI has classes for both triangles and for bounding boxes so it was only needed to loop over all vertices that was generated and form triangles. These are later used in the constructor for the bounding box class.

HAPI also has a class called FeedbackBufferCollector which automatically takes vertices that OpenGL stores in the feedback buffer and construct a mesh out of these. This wasn't needed in the project but if a hierarchy of objects are transformed several times this is a welcome feature.

When the haptic transfer is complete the force-feedback could be provided by the haptic device, although cracks between the triangles could appear if one uses a point-based renderer as described below.

3.5.1 Numerical precision problems

Because of the computers inability to store arbitrary floating point numbers, numerical precision problems appeared during the interpolations. Since every dataset gets normalized to a 1^3 cube, every sub-cubes size is shown in equation 2. Furthermore, the vertices that construct these cubes holds values between 0.0 and 1.0 are shown in equation 3.

$$\begin{aligned} |X_{subcube}| &= \frac{1}{X_{original}} \\ |Y_{subcube}| &= \frac{1}{Y_{original}} \\ |Z_{subcube}| &= \frac{1}{Z_{original}} \end{aligned} \tag{2}$$

$$\begin{aligned} &(X_{pos}, Y_{pos}, Z_{pos}) \\ X_{pos} &= \frac{a}{X_{original}}, 0 \le a \le X_{original}, X_{pos} \in [0.0 \ldots 1.0] \\ Y_{pos} &= \frac{b}{Y_{original}}, 0 \le b \le Y_{original}, Y_{pos} \in [0.0 \ldots 1.0] \\ Z_{pos} &= \frac{c}{Z_{original}}, 0 \le c \le Z_{original}, Z_{pos} \in [0.0 \ldots 1.0] \end{aligned} \tag{3}$$

When the interpolation then occur, values in between these vertices are stored. This is shown to be problematic since the computers inability to store arbitrary floating point numbers. This led to small cracks between the triangles constructed, because an interpolation from A to B does not guarantee the same vertex as if the interpolation occurred from B to A.
As mentioned earlier HAPI consist of four renderer classes where only one models the proxy as a sphere. With the point based renderers in HAPI this caused the probe to fall through these gaps, since the point is infinitely small. The solution was to switch renderer to Ruspini renderer, the one that modelled the proxy as a sphere. This made it impossible for the probe to fall through these cracks as shown in figure 9.

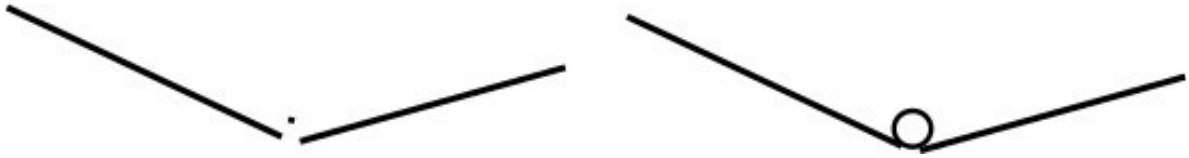


Figure 9: Sphere modelled proxy handles cracks

3.6 Memory constraints

A problem arises when using the OpenCL technique described earlier in the report and that is the space complexity. The GPU has a very limited amount of global memory available, although this can be modified up to about one half of the total global memory, using environment variables.

If a dataset has the dimensions 512^3 this would first run the count kernel by providing the dataset and producing the local and global offset-indexers. If assuming a float occupies 4 bytes, a short occupies 2 bytes and an integer occupies 4 bytes the global memory needed for the count kernel is shown below.

$$Workgroups = \lceil \frac{512^3}{256} \rceil = 524288$$
$$Workitems = Worgroups * 256 = 134217728$$
$$S_{dataset} = 512^3 * 4 = 536870912$$
$$S_{global} = Workgroups * 4 = 2097152$$
$$S_{local} = Workitems * 2 = 268435456$$
$$S_{dataset} + S_{global} + S_{local} = 807403520$$
$$S_{totalMB} = \frac{807403520}{1024^2} = 770$$

The ceiling function are of no use to sets like this, where any dimension is a multiple of the denominator, but it is needed once this division does not end up even, like 152*204*151.

This indicates that just to count the triangles 770 megabytes of GPU memory is needed. Furthermore since all of this information is provided to the next kernel, march, this leaves very little space to produce vertices and normals.

Since some datasets are as large as this example dataset or even larger this problem had to be considered. Four means to overcome this was considered and these were:

- Downsample the dataset
- Multipass kernel execution
- Recount the triangles in the march kernel
- Reuse vertices and procedurally calculate normals

3.6.1 Downsampling the dataset

This is the simplest of the four solutions. By downsampling, the dataset is recalculated by taking every 2^3 voxels and crate a new voxel which contains the avarage value of the intensities contained in the 2^3 voxels.

This method therefore creates a new downsampled dataset at 1/8th of the original size since all dimensions are divided by two. This implies that all memory need also scales at 1/8 as below.

$$Workgroups = \lceil \frac{256^3}{256} \rceil = 65536$$
$$Workitems = Worgroups * 256 = 16777216$$
$$S_{dataset} = 256^3 * 4 = 67108864$$
$$S_{global} = Workgroups * 4 = 262144$$
$$S_{local} = Workitems * 2 = 33554432$$
$$S_{dataset} + S_{global} + S_{local} = 100925440$$
$$S_{totalMB} = \frac{100925440}{1024^2} = 96.25$$

This drastically decreases the space needed but unfortunately this solution looses information.

The downsampling method was implemented as a command line flag, --d, followed by a number. The number indicated how many downsample iterations was to be performed. A way of determine the final downsampled size was $\frac{Originalset_{mb}}{8^d}$. This algorithm could be used with the number one, and sometimes two, without serious deformation of the dataset. Since every intensity in a downsampled set contain the average of 8^d intensities there is also an upper limit of how many times a downsample can occur. Screen 4 shows an outcome of the downsampling.

Screen 4: A downsampled set. 38104 triangles constructed instead of 299456

3.6.2 Multipass kernel execution

The multipass method relies on several kernel runs where each has access to a subset of the original dataset. This way a smaller amount of data needs to be handled by the GPU each run and no information about the original dataset is lost.

By slicing the dataset in half in all dimensions each subset will contain 1/8 of the original dataset. Each of these subsets will then be executed on OpenCL but in sequence. The individual work items in each subset will of course execute in parallel, the sequence execution only applies to the subsets. With this method each subset will need the same amount of data as the downsampling, but it won't be needed to store any data on the GPU once a subsets vertices and normals are calculated.

A slight problem with this approach is that there is no way of telling how many triangles the full dataset produces, only the subsets. This means that vertices and normals will be stored in dynamic memory at spread locations on the heap. By subdividing in half there would be eight of these locations. OpenGL wont be satisfied with spread memory so this has to be addressed. One way to do this would be to store each subsets output in a vector of pairs where the pairs holds the pointers to the data as well as the length of the data. This way an iteration over the vector could use memory management techniques like `memmove` to store all the vertices and normals in a continuous cluster of memory.

Another problem is that the kernel maps all vertices to $\left(\frac{x}{Width_{subs}}, \frac{y}{Height_{subs}}, \frac{z}{Depth_{subs}}\right)$ in other words, each component end up in the range $0 \leq x, y, z \leq 1$. Since this maps to the full 1^3 space, but each subset only applies to $\frac{1}{8^{subdivisions}}$ of the full subset the produced vertices needs to be scaled and translated to their correct position.

The algorithm scales very well until the allocated memory on the computer outruns the available memory that OpenGL has access to on the GPU. But in that case the only way to work with the dataset is some form of downsampling.

The drawbacks of the method is that instead of one execution of the kernel, $8^{subdivisions}$ executions are needed.

3.6.3 Recount the triangles

This method did not output the local offsets from the count kernel, but instead recalculated these offsets in the march kernel. This was space conservative with the expense of the overhead to calculate the offsets again. The space saved compared to the set above is shown below.

$$Workgroups = \lceil \frac{512^3}{256} \rceil = 524288$$

$$S_{dataset} = 512^3 * 4 = 536870912$$
$$S_{global} = Workgroups * 4 = 2097152$$
$$S_{dataset} + S_{global} = 538968064$$
$$S_{totalMB} = \frac{538968064}{1024^2} = 514$$

With only a little overhead by the recount of the offsets, over 250 mb was saved. Although this algorithm does not scale as well as the subdivision it is easily implemented. The only thing needed for it to work is to modify the kernel arguments and augment the march kernel to count the local offsets.

3.6.4 Precalculated normals and reuse of vertices

Another solution to the memory constraint would be to reuse shared vertices and to use procedurally calculated normals. By considering the triangles in figure 10, they all share the vertex V, but they all produce a similar copy of V in memory. In this example it means that the vertex is stored five times which makes four of the saves redundant.

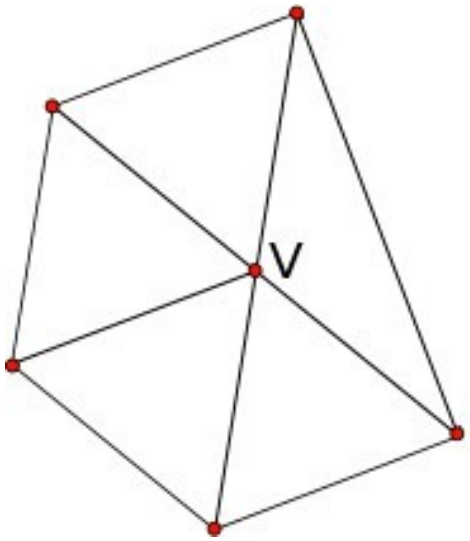


Figure 10: Redundant information. The vertex V is stored five times.

A lot of space could be saved by reusing the vertices and use indices to point out which vertices to use instead of storing them multiple times. If a vertex is shared among N triangles, this would save N-1 vertex storages in memory.

An even more drastic way to conserve memory would be to calculate the normals procedurally inside the vertex shader. This could be accomplished by exploiting the W-component of the homogeneous coordinate and use this value as a key to the procedure. Since the W-component is a floating point number its memory occupies 32 bits. By using these bits in a consistent manner, the normals doesn't have to be stored as three floating point numbers, but instead as one. The table below shows how the bit field can be used.

The red bits correspond to the X-component, the green to the Y-component and the blue to the Z-component. The last two bits are unused. Since every component is built up from 10 bits, where one has to be a sign bit this procedure would be able to produce 2^9 different components on both sides of the sign. In other words normals could be saved in steps of $\frac{1}{2^9} = \frac{1}{512}$. Now the bit field corresponding to the original normal will be found by dividing

the original normal with this factor. For example if the gradient is calculated to $(-0.432, 0.9432, 0.13321)$ the three approximated components are shown below:

$$N_{xapprox} = round\left(\frac{-0.432}{\frac{1}{512}}\right) = round(512 * (-0.432)) = -221$$

$$N_{yapprox} = round\left(\frac{0.9432}{\frac{1}{512}}\right) = round(512 * (0.9432)) = 483$$

$$N_{zapprox} = round\left(\frac{0.13321}{\frac{1}{512}}\right) = round(512 * (0.13321)) = 68$$

$$\vec{N_{approxed}} = \left(\frac{-221}{512}, \frac{483}{512}, \frac{68}{512}\right) \approx (-0.43164, 0.94336, 0.13281)$$

$$\left\| \vec{N_{real}} - \vec{N_{approxed}} \right\| \approx 5.6 * 10^{-4}$$

As shown the approximated normal is quite accurate and could be used without serious implications on the shading.

If the first bit in each sub field is the sign bit the above values should be stored like this:

1	0	1	1	0	1	1	1	0	1	0	1	1	1	1	0	0	0	1	1	0	0	0	1	0	0	0	1	0	0	X	X

Another possibility is to use all 32 bits and to make two of the three axis at a higher resolution, or to use any other form of storage.

Now these normals has to be extracted from the W-component in the vertex shader by using shifts and masks. Additionally the result should end up in the interval [-1.0...1.0]. This task could be accomplished using the calculations in equation 4, where ReadBit reads one of these sub fields.

$$0 \le ReadBit(X) \le 2^{10}$$

$$N_{extracted} = \frac{ReadBit(N_{component}) - 2^9}{2^9} \qquad (4)$$

$$-1.0 \le N_{extracted} \le 1.0$$

When the normal is extracted, the w-component is assigned 1.0 for the further process in the graphics pipeline.

This method would now require 1/3 of the memory compared to the explicitly stored normals. Even though this is an approximation, it will only be an approximation on the shading. The actual haptic feedback would still be accurate.

Some of the smaller datasets produce 300 000 triangles and larger ones produce over two million, and by using this method only a third of the memory to store the normals is needed.

Additionally, if we consider that the vertex positions are required to be stored as four floating point values (X, Y, Z, W) instead of (X, Y, Z) values, the storage of the normals becomes completely free by using the W value as outlined above and during the vertex shader stage give a value of 1 as W-component after extracting the normal from the bit-field.

4 Result

The project contains the three mesh generators; Pointcloud, Cubes and MarchingCubes. To create more generators one has to derive the class Meshalizer and then implement the method void generateVertices(), void generateVerticesCL() and void toggleWireFrame().

These three generators are different and therefore their complexity are varying. Pointcloud and Cubes both operate by looping through all intensity values and extract those over the threshold.

MarchingCubes on the other hand needs to make cubes for each intensity value inside the volume and this creates a large number of these cubes. For example, a dataset at the dimensions 512^3 would produce 134,217,728 of these cubes. Additionally each of these sub-cubes needs to go through these steps:

- Find out the bit field of the sub-cubes vertices
- Using this bit field to find out which edges are intersected
- Interpolate vertices
- Interpolate normals

This shows that the computational burden of the marching cubes algorithm is significantly greater than the previous two.

Since marching cubes is the one generator that produces the most realistic result, the below measurements are all corresponding to this generator. The following section present graphs and screens that shows different datasets.

4.1 Testing

In this chapter testing is conducted to assure that the project handles different datasets as well as different sizes. In the first section different datasets are tested and in the second section timing is presented.

4.1.1 Qualitative testing

In order to validate the designed haptic interface, qualitative test runs have been preformed on a number of different datasets. This made it possible to manually verify the visual accuracy of the generated triangles. All of the presented datasets have been visualized correctly by the marching cubes algorithm as well as have provided acceptable haptic feedback.

Test runs are preformed on the following datasets:

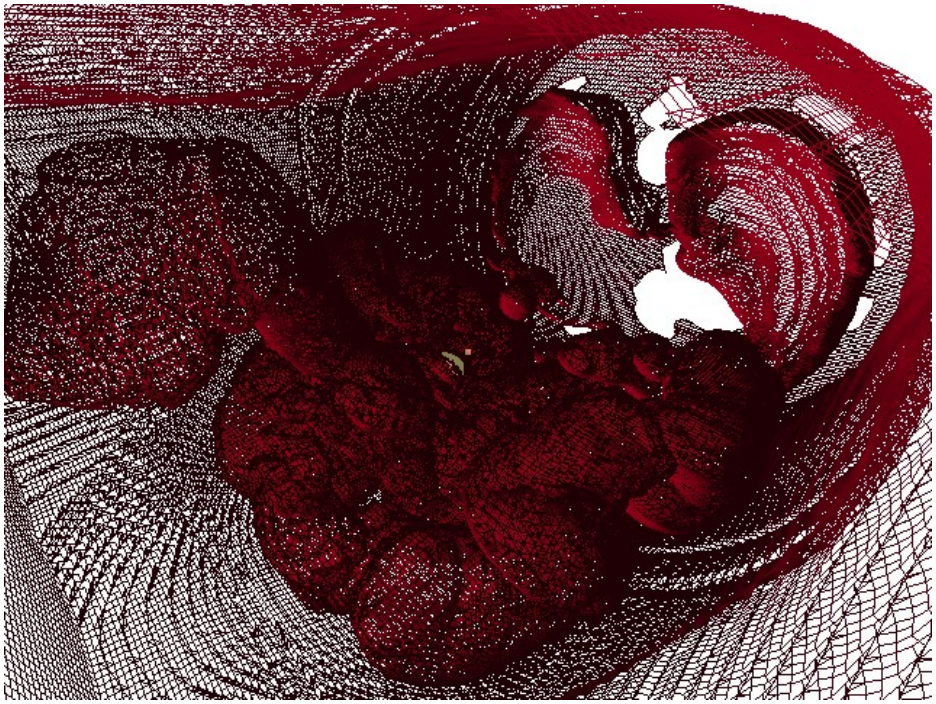
- Ultrasound from echocardiography

  Several ultrasound datasets were provided. Screen 5 shows one of several frames of a heart.

Screen 5: Heart. Dataset size : 148 * 201 * 150

- CT dataset of a torso

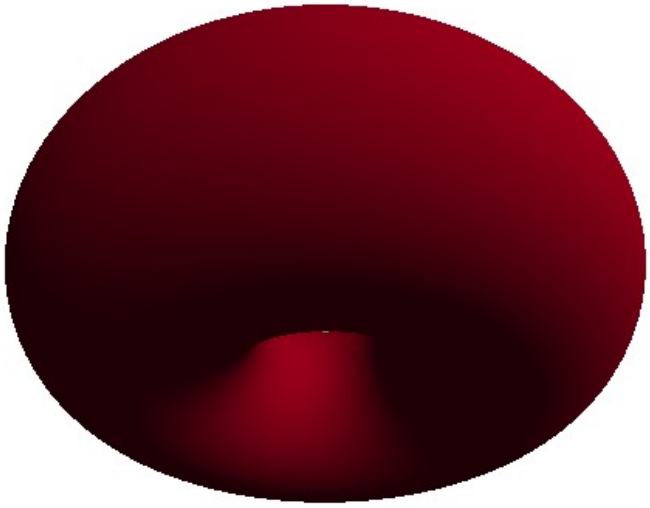
  Screen 6 represents the visualization of a dataset of a human torso. Wire frame mode has been activated for easier distinguishing of the organs.



Screen 6: Human torso. Dataset size : 512*512*891 (downsampled to 128*128*222)

- Procedurally generated torus

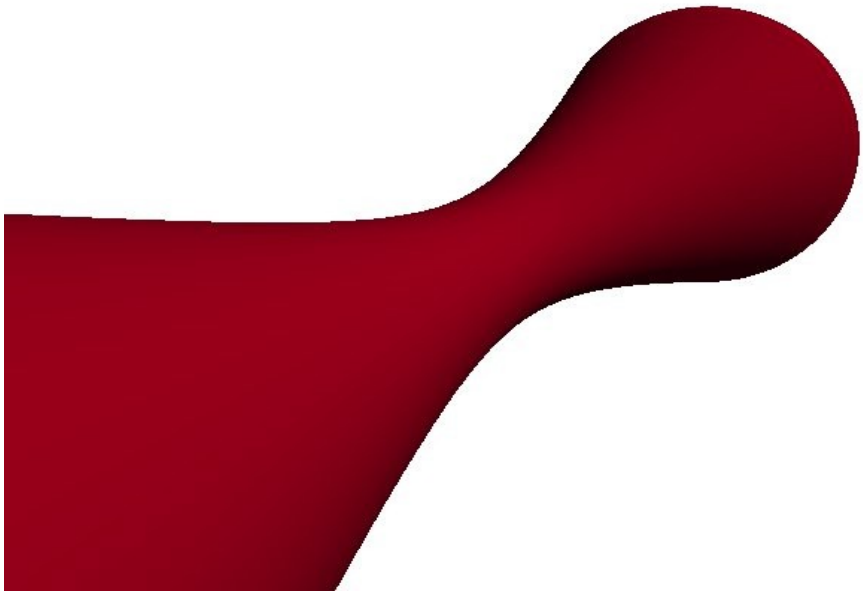
  Screen 7 shows the result of the non-parametric function $\left(1-\sqrt{\left(x^2+y^2\right)}\right)^2+z^2\leq 0.5$:



Screen 7: Torus. Dataset size 250*250*250

- Procedurally generated hyperboloid

  Screen 8 shows the representation of the non-parametric function $\frac{x^2+y^2}{0.5^2}-\frac{z^2}{1}\leq 1$:

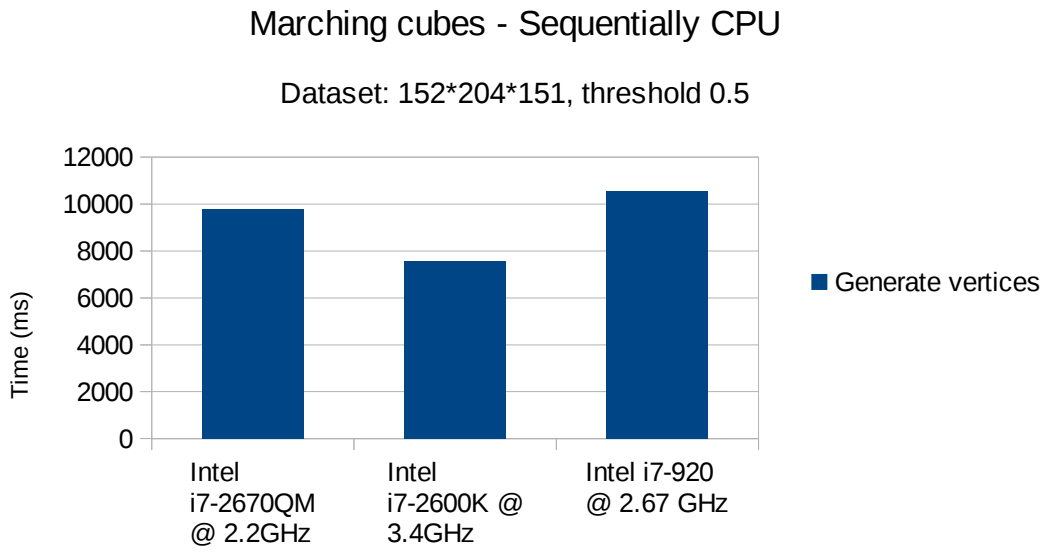


Screen 8: One-sheeted hyperboloid. Dataset size : 300*300*300

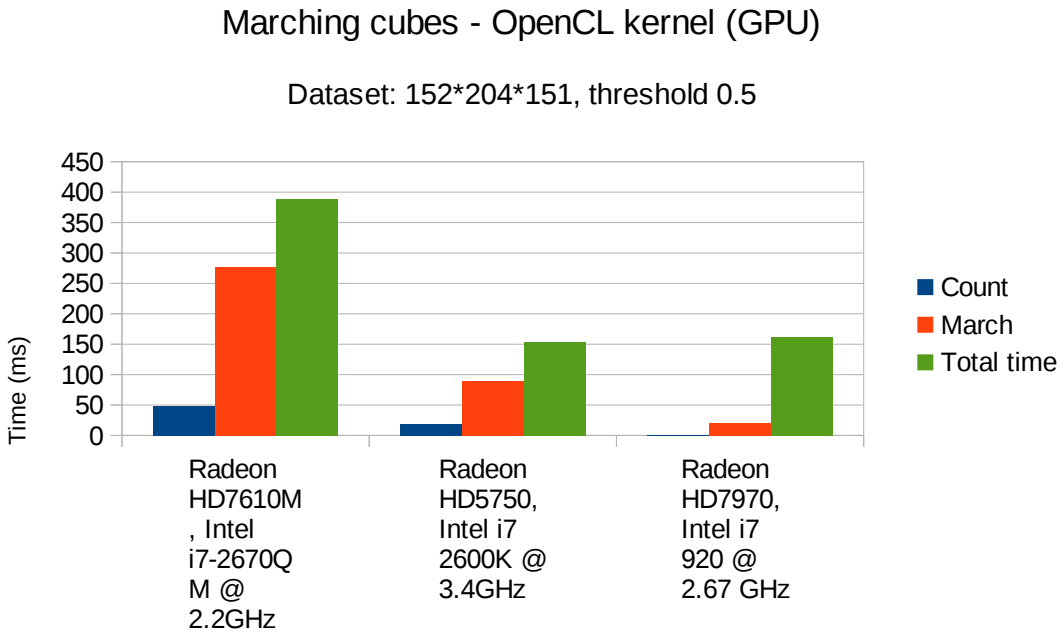
4.1.2 Quantitative testing

In this section the result of the profiling as well as the time estimate for arbitrary sets is presented.

The first graph shows the time it takes to do the marching cubes algorithm on the CPU sequentially. The measurement is done with the self made timer class. Three different systems are measured and these are shown along the x-axis.



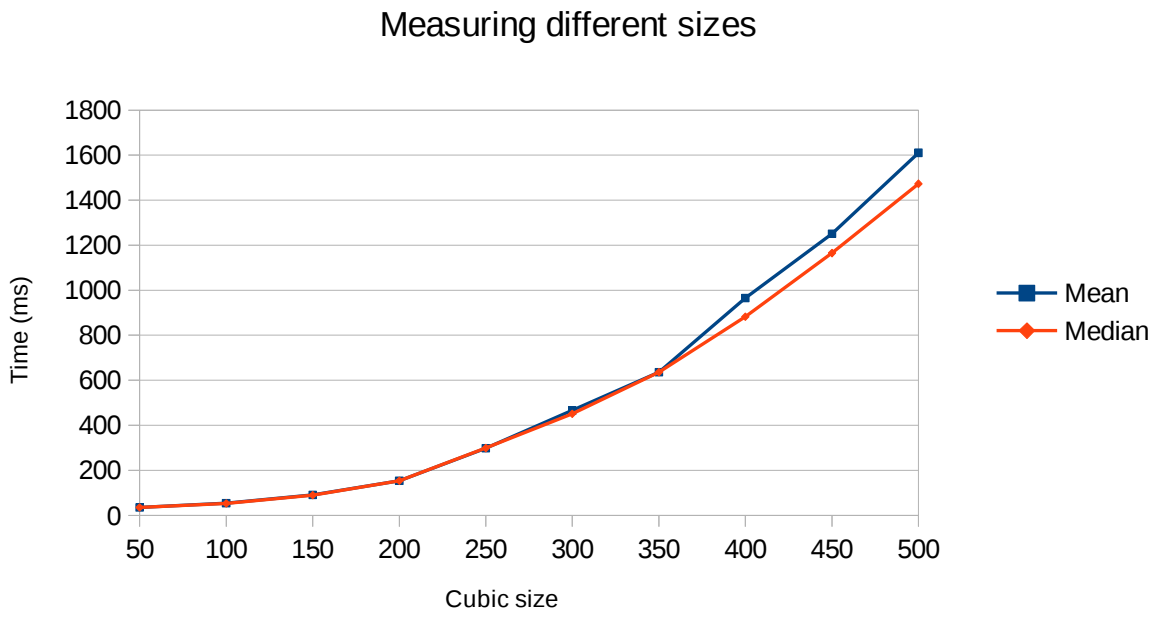
This second graph shows the time measured in CodeXL for executing the two kernels to complete the algorithm as well as the total time taken. The same systems were used.



The green bar shows the total time, including the overhead on the CPU. There is a number of ways to lower this bar, most significantly by sharing data between OpenGL and OpenCL. Currently the triangles gets calculated on the GPU (OpenCL), gets fetched by the CPU, and again sent to the GPU (OpenGL). This is not the optimal way to go, and since the device used by OpenCL is the GPU it would be more effective to share the data between OpenGL and OpenCL.

Although, the speedups from conducting the calculations on the GPU are over 60 times faster than the counterpart on the CPU, when executed on a modern graphic card.

To estimate the time needed to construct arbitrary dataset the ability to construct datasets was added. This generated a dataset of size N^3 out of a non-parametric equation. The actual object constructed did not matter, but rather the size of the set holding the object. The graph below shows the time needed to construct a torus where N goes from 50 to 500 in steps of 50.



5 Discussion

The project was successfully completed and further development of the product could lead to a working simulator for medical students. However, for this to happen a lot of further considerations has to be accounted for. Some of these topics are discussed in this chapter.

5.1 Ethical aspects

During the project a reflection about the simulation occurred. What if a heart was simulated and the simulation didn't provide forces that made the haptic feedback to behave like the sense of touch of a real heart. This would train the students to become familiar with this misleading sense of touch. When the time later comes for the students to preform an actual surgery they are trained wrong, and a mishap in reality would occur. During the training this would not show because of the misleading forces.

To really be able to use it for training further development is needed as well as professional surgeons that has to evaluate the product until the forces behave lifelike enough. Before professional use can be made of the product a proper evaluation including such human subjects would need to be performed, including any necessary ethical approvals and a sound methodological evaluation. This work is considered outside the scope of this thesis project.

For the performers using the simulation they also have to keep in mind that it is only a simulation.

5.2 Socio-economical considerations

In addition to the ethical aspects discussed above, the economical implications for the society must also be accounted for when developing tools for clinical use or for training of medical staff. Although the use of simulations may improve the medical quality, there is a cost tied to the dependence of the tools needed to preform these simulations. In this project these tools were the haptic device and the imaging equipment, but additional hardware may be needed for other simulations. Examples of this increased cost of advanced medical techniques can already be seen today as one of the problems faced whenever the medical companies introduce new products that can improve the life of patients but at a prohibitive cost to the local counties in countries with socialized health-care, such as Sweden.

Since the project is in its early stages, it is too early to say whether the potential gains of the project is worth its cost. A preliminary look at the cost of haptic devices compared to other common medical equipment such as an ultrasound scanner, indicates that this may make a cost-effective tool. However, this trade-off should be considered carefully before too much is invested in the technique.

5.3 Achievement of the course objectives

When the project began I had to do research about different methods that extracted a surface from a 3D scalar field, and there I came into contact with marching cubes [1]. The algorithm had to be studied in detail because of its several steps for mesh-generation. When implemented I had to learn how to use OpenCL to preform parallel tasks, as well as to “think” in parallel. Problems arose when barriers were forgotten or added, leading to different results every time respectively the lock-up of the system.

I also conducted a lot of research in the field of accelerated mesh generation as well as the field of haptics. The project also gave me a quick-course in OpenCL, which I find terrific and most certainly will continue to use on algorithms suited for it.

5.4 Compliance with the project requirements

All hard requirements for the project were completed successfully. The backup system which was not accounted for at the beginning of the project was also completed. Even though this was not a soft requirement in the start, this became apparent that this needed to be addressed somehow.

The physical simulation with biosim was not completed. Since the problems with the communication appeared, a lot of time was devoted to provide a temporary fix for the timeouts. Since these problems not were accounted for when the project started the time scheduling of the project had to change.

Finally, the conclusion considering the projects results are listed below:

- The haptic device is only compliant on computers with an parallel port that is attached to the motherboard.
- Few modern computers support the parallel port directly on the motherboard.
- To use an older computer with integrated parallel port would not be an option. This is due to the fact that even although the communication might be successful, OpenCL would not be able to run at the impressive speeds shown above, if at all.
- To rewrite the driver that communicates with the haptic device is not an option because of its closed-source status.

The trade off in communication/speed shouldn't even be a trade off, but the requirement of an integrated parallel port for communication and a powerful GPU for OpenCL forces it to be.

As a last resort an experienced haptic user, Jorge Solis, at Karlstad University was also contacted about the timeout problems, but unfortunately he hadn't experienced these problems.

5.5 Project development

The research in the field of haptics and visualized medical volumes could lead to an increase in remote surgery. This would therefore make it possible for a skilled surgeon, located in a research hospital, to do remote surgery on an individual far away. This would lead to an equilibrium of the medical quality worldwide.

Mizokami et al. describes that visual perception on average takes about 30msec to interpret while haptic perception only takes about 1 msec [7]. This thirty-folded speed up is needed for this remote surgery to succeed.

But other, more difficult factors to account for, has to be considered as well. For example what would happen if several packages were lost during the operation or if a power failure occurred?

A further addition to the project could also be to have a working physical simulation, like the one described earlier.

An additional future addition to the project could be to use the haptics on a sequence of frames instead of one frame, like it is now. That could simulate a beating heart with proper haptic feedback.

6 References

[1] Lorensen, W.E. & Cline, H.E. 1987, "MARCHING CUBES: A HIGH RESOLUTION 3D SURFACE CONSTRUCTION ALGORITHM.", *Computer Graphics (ACM),* vol. 21, no. 4, pp. 163-169.

[2] Archirapatkave, V., Sumilo, H., See, S.C.W. & Achalakul, T. 2011, "GPGPU acceleration algorithm for medical image reconstruction", *Proceedings - 9th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2011*, pp. 41.

[3] Karadogan, E. & Williams II, R.L. 2013, "Haptic modules for palpatory diagnosis training of medical students", *Virtual Reality,* vol. 17, no. 1, pp. 45-58.

[4] H3DAPI, Home page for H3D
Fetched: 2013-04-14
URL: http://www.h3d.org

[5] Scarpino, M. (2012). *OpenCL in action : How to accelerate graphics and computation*. Shelter Island, NY: Manning

[6] Aftab Munshi, *The OpenCL Specification, ver 1.2*, Revision 19.
Fetched: 2013-05-14
URL: http://www.khronos.org/registry/specs/opencl-1.2.pdf

[7] Mizokami, R.; Abe, N.; Kinoshita, Y.; He, S., "Simulation of ICSI Procedure Using Virtual Haptic Feedback Model,"*Complex Medical Engineering, 2007. CME 2007. IEEE/ICME International Conference on*, vol., no., pp.175,180, 23-27 May 2007

[8] Demming, Robert & Duffy, Daniel J. (2010). *Introduction to the Boost C++ Libraries*. Volume 1 – Foundations

[9] Smistad, Erik, Anne C Elster, and Frank Lindseth. 2011. Fast Surface Extraction and Visualization of Medical Images using OpenCL and GPUs. In *The Joint Workshop on High Performance and Distributed Computing for Medical Imaging 2011*.

[10] Eriksson E. Simulation of Biological Tissue using Mass-Spring-Damper Models. 2013.

[11] Boost 1.53.0 Library documentation, Homepage for Boost
Fetched: 2013-05-01
URL: http://www.boost.org/doc/libs/1_53_0/

[12] OpenGL Mathematics (GLM), Homepage for GLM
Fetched: 2013-04-20
URL: http://glm.g-truc.net

[13] Git, Homepage for git
Fetched: 2013-04-22
URL: http://git-scm.com

[14] Magnetic Resonance Imaging (MRI Scan), Homepage
Fetched: 2013-05-27
URL: http://www.medicinenet.com/mri_scan/article.htm

Appendix A – Driver rewrite to force EPP-mode (parport_pc.c)

This function below check if ECR is present on the parallel port. ECR is needed for ECP to be chosen as mode of transfer, so by pretending that the parallel port did not support ECR, ECP could not be chosen.

By making the change:

```
static int parport_ECR_present(struct parport *pb)
{
     struct parport_pc_private *priv = pb->private_data;
     unsigned char r = 0xc;

     outb(r, CONTROL(pb));
     if ((inb(ECONTROL(pb)) & 0x3) == (r & 0x3)) {
          outb(r ^ 0x2, CONTROL(pb)); /* Toggle bit 1 */

          r = inb(CONTROL(pb));
          if ((inb(ECONTROL(pb)) & 0x2) == (r & 0x2))
               goto no_reg;
     }

     if ((inb(ECONTROL(pb)) & 0x3) != 0x1)
          goto no_reg;

     ECR_WRITE(pb, 0x34);
     if (inb(ECONTROL(pb)) != 0x35)
          goto no_reg;

     /* Here */
     goto no_reg;

     priv->ecr = 1;
     outb(0xc, CONTROL(pb));

     /* Go to mode 000 */
     frob_set_mode(pb, ECR_SPP);

     return 1;

 no_reg:
     outb(0xc, CONTROL(pb));
     return 0;
}
```

and recompile the kernel the card was forced to go into EPP mode. This is due to the fact that the variable priv->ecr never gets assigned a one, and a consequence of this is that a later function will return that the card can not be run in ECP-mode.