Cardiac Surgery Simulation

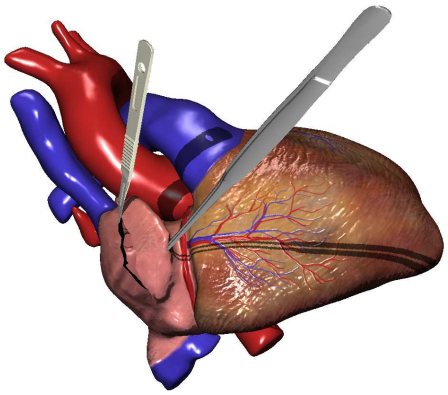
Graphics Hardware meets Congenital Heart Disease

Jesper Mosegaard

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Cardiac Surgery Simulation



A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
PhD Degree

by
Jesper Mosegaard
October 7, 2006

Abstract

The human heart has a very complex morphology, and combined with a congenital heart defect it becomes a challenge to understand the spatial arrangement in full. Pediatric cardiac surgeons must have a very good understanding of the morphology of the heart of a patient prior to a surgical procedure, including an overview of potential surgical strategies related to the heart defect.

In this PhD-thesis I present methods for the implementation of a real-time simulator for surgical procedures on congenital heart defects. A prototype has been developed and evaluated in cooperation with pediatric cardiac surgeons. The thesis is divided into three main parts; a presentation of surgical procedures in relation to congenital heart defects, an overview of the research field of surgical simulation, and lastly my main research contributions on the acceleration of surgical simulation through utilization of programmable graphics hardware, as well as a preliminary evaluation of the simulator in incision planning.

Four specific research contributions are presented. The first contribution presents methods to utilize the graphics processing unit (GPU) for soft-tissue deformation, resulting in a speedup of 20-30 times compared to a similar CPU implementation. The second contribution is a mapping from the simulation to an arbitrary visual surface, allowing a more detailed visual surface to be deformed by the GPU-based simulation in real-time on the GPU. The third contribution is a method for haptic-interaction with the GPU-based simulation, taking the slow memory transfer from GPU to CPU into account. The fourth contribution is the evaluation of the surgical simulator for training and pre-operative incision planning. This clinical research has been made possible through the development of a working prototype.

Acknowledgments

First of all thanks to Thomas Sangild Sørensen who has been my main collaborator throughout my PhD-project. Thanks also to pediatric heart surgeons Doctor Ole Kromann Hansen and Professor Vibeke E. Hjortdal from Aarhus University Hospital. Through the last four years they have introduced me to the topic of heart surgery, let me experience real surgical procedures, and participated in technical workshops. Naturally thanks to my adviser Peter Møller Nielsen. I am grateful for the time spend by Peter Bøgh Andersen, Michael E. Caspersen, Peter Møller Nielsen and Rory Andrew Wright Middleton for proof-reading. I am very grateful for the grant from the Danish Heart Foundation which enabled us to hire two student programmers to work on the simulator in the last 8 months of my PhD project. Thank you also to the current and previous people in and around the Centre for Advanced Visualization and Interaction where I have had my office and daily routine for the last four years which has been very enjoyable. As part of this PhD I have also participated in the activities of the Centre for Pervasive Healthcare which has represented a nice variation in topics of computer science.

Lastly, thanks to my family and friends for allowing me to talk endlessly about my passions and difficulties as a PhD-student. Thank you to my parents, Marianne and Per, my sister, Lene, and my girlfriend, Dorte, for everything else in life.

Jesper Mosegaard,
Aarhus, October 7, 2006.

Contents

Chapter 1

Introduction

In this chapter I will motivate the real world issues dealt with in this thesis through a specific case of a boy named Mikkel, who had heart surgery because of a congenial heart defect. Following that, I will define the problem formulation both from a clinical as well as a computer science perspective.

1.1 Mikkel

Mikkel is a four year old boy from Denmark. He is clearly a charming little boy, full of fun. But Mikkel is not all well; when Mikkel plays games with his older brothers they must be quite careful and they are very protective of him. Mikkel will sometimes tell his brothers; "Feel my heart, it's beating real hard". In Mikkel's case this is quite a serious problem since he is born with a congenital heart defect - more specifically a small hole inside his heart. In the television program "When the heart of Mikkel was stopped"[1] [116] we, as viewers, get a close look at the issues in heart surgery - both at a personal level through Mikkel and Mikkel's parents, and at a professional level through the pediatric cardiac surgeon Vibeke Hjortdal. It becomes very clear from all people involved, perhaps except little Mikkel, that heart surgery is a serious matter. The surgeon Vibeke Hjortdal says to the parents of Mikkel (First the original danish transcript, then the English translation):

> "Det plejer jo at gå godt med de her operationer - de fleste gange. Men når jeg kun siger de fleste gange og ikke alle gangene, så er det jo selvfølgelig et udtryk for, at engang imellem så går det ikke godt. Og det kan gå helt galt, han kan dø under operationen - det sker heldigvis meget meget sjældent."

[1] The author encourages any reader to view the broadcast since it is easily available on the Internet. The broadcast is in danish, but gives a very good view of the surgical procedure even for the English speaking.

> "These surgical procedures usually succeed - most times. But when I only say most times, and not every time, it is of course a reflection upon the fact that sometimes such surgery is not a success. It can go all wrong - he can die during the surgery. Fortunately, that happens very, very rarely."

To correct for the congenital heart defect of Mikkel, Vibeke Hjortdal must open his chest, stop the beating of his heart, and make an incision into his heart to reach the hole. The hole will then be patched by suturing a small piece of artificial material to the edges of the hole. In Vibeke Hjortdals own words;

> "Man er jo nødt til at være perfektionist når man står og laver det her [kirurgi]. "Attention to deails"; det der med virkeligt at have sans for detaljerne. Det er så lidt der skal til, før man er ude på den forkerte side af hvad der er muligt hos sådan nogle små størrelser. Så hver eneste lille ting man laver, det skal bare laves perfekt; alt lige fra åbne brystkassen, til at få de her katetre puttet ind i blodårene - det hele skal bare være rigtigt, fordi summen af små ting der er gået forkert kan være nok til at vælte det store læs og få operationen til at gå galt istedet for at gå godt. Det er helt oplagt; man er nødt til at være perfektionist."
>
> "You have to be a perfectionist when you do this kind of thing [surgery]. "Attention to details"; the thing about having a flair for details. It takes so little to be on the wrong side of what is possible with these small children. Therefore everything you do has to be perfect; everything from opening the chest, to inserting these catheters into the blood-vessels - everything needs to be done correctly, because the sum of small mistakes can result in total failure instead of success. It is so very obvious; one has to be a perfectionist."

1.2 Problem Formulation

The PhD has come to life in the cross-section between a real world problem for pediatric cardiac surgeons and the investigation into technical challenges derived from these problems. The process of dealing with these problems has thus been an interdisciplinary cooperation with pediatric surgeons, M.D. Ole Kromann Hansen and Professor Vibeke Hjortdal, Aarhus University Hospital. Ole Kromann and Vibeke Hjortdal represent both the users and experts in the problem area. Through his interdisciplinary background, a Masters in Computer Science and a PhD in medicine, Thomas Sangild Sørensen has played the role of bridging the clinical world and computer science. Concerning the role of my PhD-project I have primarily kept a computer science

perspective deriving interesting computational problems from the problem area with general applicability. The clinical problem area has been a great motivation and source of challenging problems in the development of the surgical simulator prototype. Data acquisition has mainly been done in cooperation with the MR-center from Skejby Hospital. Our cooperation has resulted in a synergistic effect with both clinical and technical results. In the next two sections I present the problem formulation from a clinical perspective as well as from a technical perspective. The major part of this thesis is naturally of a technical nature, but is very tightly bound to the requirements of the real-world problem of the surgeons.

1.2.1 Clinical Problem Formulation

Congenital heart defects are present in about one percent of live births and are the most common malformations in newborn children. In the last decade overall mortality rates have dropped to just 5% but in the case of complex defects, the mortality rate remains as high as 20-30%. One of the issues is that the heart is a complicated organ since both functionality and especially morphology of the heart is very complex. A congenital heart disease can introduce further complexity due to the inherent malformation. It is consequently difficult to get an overview of the spatial structure and morphology of the malformed heart. An accurate and detailed understanding is critical to the diagnosis and the very careful planning of surgical strategies. Surgeons must have an accurate understanding of the three-dimensional layout of the different components of the heart. The shape of the heart must also be understood in relation to the required functionality and the different potential strategies that can be used to correct for the defects. In general, there are three ways to reduce mortality in complex congenital heart diseases; through improved diagnosis, better early training of surgeons, and through better pre-operative planning. This PhD project aims to improve training and pre-operative planning through *surgical simulation*. In an interdisciplinary cooperation with pediatric cardiac surgeons it is our goal to build a *surgical simulator* specifically for pre-operational planning and training for surgical procedures on children with congenital heart defects - such as Mikkel from the previous section.

Problem Surgery on the heart of a child with a congenital heart defect requires good understanding of the heart morphology in relation to the steps of potential surgical strategies.

Hypothesis Going through the surgical procedure, experimenting and exploring the heart in a virtual setting, provides a better understanding of the heart morphology in relation to potential surgical strategies in both pre-operational planning and in training in general.

Method Develop a working prototype of a real-time surgical simulator for congenital heart defects.

Limitations Although we have a tight interdisciplinary cooperation, the major goal of the clinical problem formulation in terms of this PhD-thesis is as a facilitator for interesting aspects of computer science. As such a formal clinical evaluation and implications of the surgical simulator is not given primary attention in the remaining thesis. We have conducted a preliminary evaluation of the simulator for incision planning as presented in chapter 12.

1.2.2 Computer Science Problem Formulation

A simulation is an artificial model of a real procedure, phenomenon or system. The simulation defines rules of behavior that represent the real phenomena to a certain degree. Frasca Gonzalo has this simple definition of a simulation:

> "Simulation is the act of modeling a system A by a less complex system B, which retains some of A's original behavior" [68]

What this definition does not cover is the reason for doing simulation. Often a simulation is used in training, recreation of real situations or prediction of real world phenomena. In those cases there are many reasons to simulate a given procedure as an alternative to performing it in real life. Generally the real procedure might not be viable economically or ethically, and the elements needed for the procedure might not be easily available. According to Gorman [84], the field of surgical simulation began to gain real respect in the field of surgery in the nineties.

We often restrict the scope of a simulation based on the final scenario of use and the resources available to the simulation. Considering a computer-based simulation these resources are in the form of computational power and allowed execution time. We must therefore divide the resources amongst levels of importance for the simulation, i.e. a specific aspect of a simulation might be essential to one scenario of use, and indifferent to another.

Over the years, a variety of simulations have been constructed for different needs. Physical phenomena, such as colliding galaxies (e.g. the merger of the Milky Way and the Andromeda galaxies from 2001 [61]) and aerodynamic properties (e.g. improving the shape of Ferrari Formula One Cars [124]) are classic examples, but also social phenomena, such as panic in crowds, have been simulated to design better emergency plans [92].

The clinical problem formulation defined above very clearly transfers to technical issues that are at the edge of what the field of surgical simulation can do. There has been a constant push towards the simulation of more detailed and more realistic surgical procedures. The heart in particular

is such a complex organ that previous methods have not been adequate for a surgical simulator on the current generation of hardware. Another aspect of surgical simulation is the push towards open surgery simulators, i.e. opening up the patient through surgery. Open surgery has traditionally been considered the most difficult type of surgery to simulate. Although I do not deal with open surgery in general, heart surgery is considered open surgery and many of the techniques developed are generally applicable.

Problem Simulating and visualizing tissue deformation in models with complex morphology is not fast enough with existing methods and hardware. Heart surgery is such a case, and has thus not previously been simulated.

Hypothesis Through utilization of graphics hardware, a surgical simulator for complex morphology can be constructed - thereby meeting the demands of the previous item.

Method Develop and evaluate the technical components for a GPU-based surgical simulator including soft-tissue simulation, visualization and haptic-interaction.

Limitations The developed techniques should be generally applicable in physics based animation.

1.3 Overview of the Thesis and Contributions

This thesis is divided into three parts. In part I the discipline of heart surgery on children with congenital heart diseases is introduced. The reader should thereby be able to understand the clinical background for developing tools within the limitations and challenges of training and preparation of surgical procedures. Part II presents the research fields of surgical simulation. This part will frame the specific contributions of GPU accelerated surgical simulation presented in the last part III. In part III our research contributions based on the development of the GPU accelerated surgical simulator for surgical procedures on congenital heart defects are presented. The last part is mainly based on published papers. To get an overview of the categories of contributions of this thesis I present them here with references to publications and chapters of presentation.

GPU Accelerated Spring-Mass Systems

Implementing Spring-Mass systems using the computational resources of modern GPU's, with surgical simulations as one area of application, has been presented in two full papers:

[4] Jesper Mosegaard, Peder Herborg, and Thomas Sangild Sørensen. A GPU accelerated spring-mass system for surgical simulation. In *Proceedings of Medicine Meets Virtual Reality 13. Studies in Health Technology and Informatics*, 111:342-348, January 2005.

[5] Jesper Mosegaard and Thomas Sangild Sørensen. GPU accelerated surgical simulators for complex morphology. In *IEEE Virtual Reality,* 147-154, 323, March 2005.

The initial paper [4] dealt with a single method for implementing the Spring-Mass system and focused on clinical use. In [5] we presented and compared two methods of implementing the Spring-Mass system (of which one had been presented in [4]). In the context of this thesis, the two papers have been merged to form chapter 8.

An introduction to the topic surgical simulation on the GPU in general as well as an analysis of GPU performance was published as a full paper:

[10] Thomas Sangild Sørensen and Jesper Mosegaard. An introduction to GPU accelerated surgical simulation. In Matthias Harders and Gábor Székely, editors, *Biomedical Simulation: Third International Symposium, ISBMS 2006*, volume 4072 of *Lecture Notes in Computer Science*, 93-104, 2006.

The paper [10] presents a combined introduction and survey of the components available for surgical simulation on the GPU; calculation of tissue deformation, visualization and haptics. This discussion includes a short presentation of our work implementing the Spring-Mass system. The paper is presented in this thesis in chapter 7 as an introduction and overview of previous work within GPU accelerated surgical simulation in general.

Decoupling of Visualization and Simulation on the GPU

To visualize the Spring-Mass simulation we have developed a mapping from the simulation to an arbitrary surface - in our case motivated by the desire to visualize a smooth and detailed surface based on a more jagged simulation arranged in a grid. The GPU-based technique has been published in a full paper:

[6] Jesper Mosegaard and Thomas Sangild Sørensen. Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the GPU. In *Proceedings of Eurographics Workshop on Virtual Environments*, volume 11, 105-111, 2005.

The paper [6] presented our results in a general setting but with the surgical simulation as one of the examples. The content of the paper is presented in chapter 9.

Haptic Feedback for the GPU based Surgical Simulator

Using haptic interaction with the GPU based surgical simulator is a challenging issue, since communication from GPU to CPU can easily become a bottleneck. We developed effective techniques for haptic-interaction in:

[9] Thomas Sangild Sørensen and Jesper Mosegaard. Haptic Feedback for the GPU-based Surgical Simulator. In *Proceedings of Medicine Meets Virtual Reality 14. Studies in Health Technology and Informatics*, 119:523-528, 2006.

This paper is presented in chapter 12 in a slightly extended version.

Development of the Surgical Simulator as a Working Prototype

These contributions focus on the demonstration of the simulator as a working prototype through a SIGGRAPH video publication in 2005 and as an emerging technology and technical sketch at SIGGRAPH 2006:

[7] Jesper Mosegaard and Thomas Sangild Sørensen. Technical aspects of the GPU accelerated surgical simulator. In *SIGGRAPH Application Sketches*, Boston, USA, 2006. *in press.*

[8] Thomas Sangild Sørensen and Jesper Mosegaard. Surgical planning in congenital heart disease by means of real-time medical visualisation and simulation. In *ACM SIGGRAPH* Computer Animation Festival, 2005.

[11] Thomas Sangild Sørensen and Jesper Mosegaard. Virtual open heart surgery - training complex surgical procedures in congenital heart disease. In *ACM SIGGRAPH Emerging Technologies*, 2006.

The development of a simulator prototype has been a key element in the project, enabling us to cooperate closely with the pediatric heart surgeons and other clinical personnel. This interdisciplinary cooperation has resulted directly in the clinical contributions of the next section. Coming full circle, the discussions and clinical evaluation with our surgeons has had positive influence on the development of robust generic technical components. The concrete results of developing a prototype is naturally not presented in written form, but can be investigated on the included DVD through various videos of the simulator in action (in various generations of development).

Clinical Evaluation of the Surgical Simulator for Incision Planning

Results of our preliminary clinical evaluation have been presented as an abstract combined with an oral presentation and later as a journal article:

[12] Thomas Sangild Sørensen, Jesper Mosegaard, Gerald F. Greil, Ole Kromann Hansen, and Vibeke E. Hjortdal. Preoperative planning by surgical simulation on patient-specific high-resolution virtual models. In *The Fourth World Congress of Pediatric Cardiology and Cardiac Surgery*, volume 4, 230, 2005.

[13] Thomas Sangild Sørensen, Gerald F. Greil, Ole Kromann Hansen, and Jesper Mosegaard. Surgical simulation - a new tool to evaluate surgical incisions in congenital heart disease? *Interactive Cardiovascular and Thoracic Surgery, in press*, 2006.

These results are presented in chapter 12 as an extended version of the journal article [13].

Part I

Surgery on Hearts With Congenital Defects

Chapter 2

The Heart

The Heart Center Encyclopedia [46] has in-depth information for nonprofessionals about the heart, congenital heart defects, and surgical strategies. Where nothing else is noted, the overview provided of the heart and heart diseases in this thesis is based on this encyclopedia and discussions with pediatric surgeons.

2.1 The Healthy Heart

A simplified drawing of the heart is shown in figure 2.1. What the figure does not show is the actual size of a heart. In the small children I consider in this thesis, the heart is about the size of a table tennis ball - making it a very small structure to work on. The role of the heart is to sustain the body with a circulation of blood, essentially working as a pump. The heart-functionality is divided into the right and left part (of the patient), separated by the *septum*[1]. Each part is again divided into two chambers; an *atrium* and a *ventricle*.

The flow of the blood is a rather complicated system, please inspect figure 2.1 and 2.2 during the following explanation. As seen from the outside of the heart, the atrium receives blood while the ventricle sends out blood. The left part of the heart circulates oxygen-rich blood from the lungs to the rest of the body, while the right side of the heart circulates oxygen-poor blood from the body to the lungs. In the case of the left part of the heart circulation, blood comes from the lungs through the pulmonary veins to the *left atrium.* From here, the blood goes through the *mitral valve* to the *left ventricle*, and out through the *aortic valve* to the *aorta,* the single largest blood vessel in the body. In the case of the right part of the heart circulation, the blood comes back from the body through the *inferior vena cava* and the *superior vena cava* to the *right atrium.* From here, the blood goes through the *tricuspid valve* to the *right ventricle*, and finally through

[1] from Latin; "something that encloses"

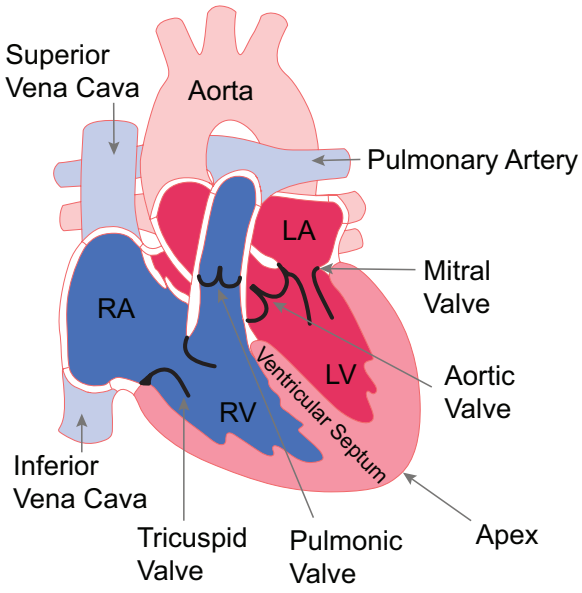


Figure 2.1: Basic Anatomy of the heart. LA: Left Atrium, RA: Right Atrium, LV: Left Ventricle, RV: Right Ventricle. Blue colors represent parts of the heart with oxygen-poor blood, and red colors represent parts of the heart with oxygen-rich blood. Image based on [46]

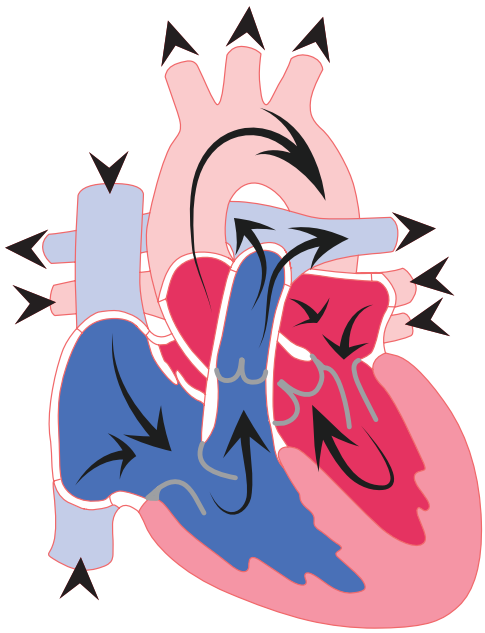


Figure 2.2: Flow of blood in a normal heart, symbolized by black arrows. Image based on [46].

the *pulmonic valve* and out through the *pulmonary arteries* to the lungs. These two parts correspond to the two circulatory systems of the body; the systemic circulation from heart to the body and back, and the pulmonary circulation from the heart to the lungs and back.

Around the heart and the roots of the major blood vessels lies a thin membrane, called the *pericardium*. Between the heart and the pericardium there is fluid to allow the heart to beat with low friction.

As indicated by this very short overview of the heart, it has a relatively complex shape (or morphology). To model even a basic representation of the healthy heart would require a high degree of geometric detail. Moving down a level, the tissue of the heart in itself is also a complicated structure, consisting of three layers of tissue with distinct physical characteristics as well as coronary arteries and the electrical system. The tissue is generally non-homogeneous due to muscle fibers in the heart.

2.2 Learning to Become a Surgeon

In this section I give a short introduction to the process of learning surgery in general. Through initial studies a medical student has a basic theoretical knowledge of surgery. In later studies of surgery, it is common for students to use a week operating on pigs or perhaps cadavers. Practicing to become a good surgeon will take many years though, as this is very much a practical skill that has to be learned by doing. The general teaching paradigm in surgery is:

> See one, do one, teach one

This is an indication of the fact that surgeons specializing within a surgical discipline are taught through the principle of a master-apprentice relationship. The master surgeon will take an apprentice and this apprentice will be taught surgery through real surgical procedures by the master-surgeon. The apprentice will start out observing the master surgeon, “See one”, and will later be allowed to try basic parts of the procedure, “do one”. Slowly he will be given more responsibility and in the end, the apprentice is allowed to do entire surgical procedures on his own. When the apprentice has reached the proficiency level of master-surgeon, he himself can take on an apprentice; “teach one”. The above saying is also an indication of the relative risk involved in going from having seen *one* to actually do *one* and finally teach *one* based on that knowledge.

At Aarhus University Hospital I have observed this master-apprentice principle in full. Vibeke E. Hjortdal was an apprentice of Ole Kromann Hansen, and would in most cases begin the surgical procedures I observed. She herself had an apprentice, who was allowed to do some basic parts of the

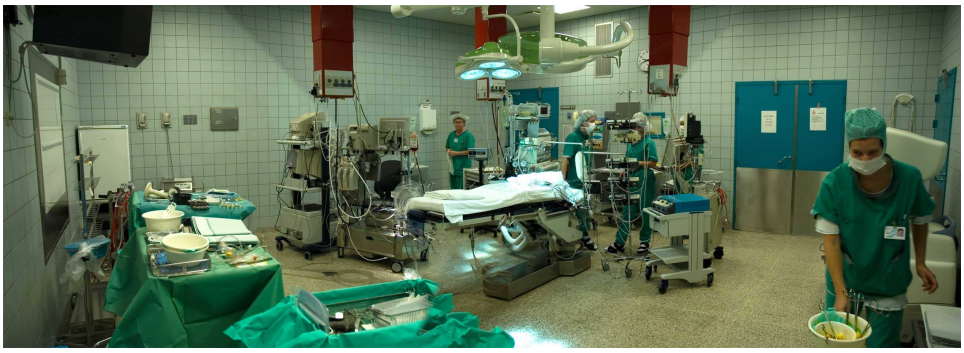


Figure 2.3: The operating theatre. Photograph by Doctor Ole Kromann Hansen

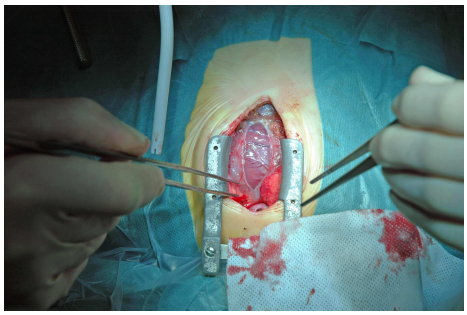
surgery and assist Hjortdal. If problems arose, Kromann would be called in to help Hjortdal and her apprentice.

2.3 Heart Defects

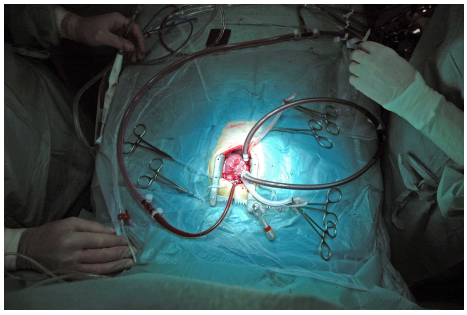
The following section gives an overview of three selected heart defects and the associated surgical strategies. This overview is by no means exhaustive, as there are over 3000 different types of diagnosises of congenital heart defects. For practical reasons, the key-terminology is introduced so that the morphology can later be discussed in detail with respect to the congenital malformations of actual patients in chapter 12. Presenting heart defects and surgical strategies is also important to understand the premises of creating a surgical simulator for this type of surgery, and to further underline the fact that malformed hearts have a very complex spatial arrangement.

2.3.1 Open-Heart Surgery

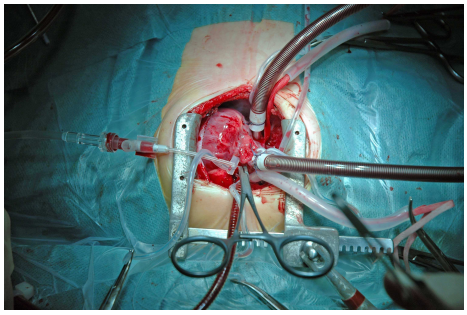
The treatment for heart defects considered in this thesis is through surgery, and more specifically through open-heart surgery. In open heart-surgery the chest is opened, the child placed in a heart-lung bypass machine, and then finally the heart is opened. When the child has been sedated, the child is taken to the operating theater where the surgical procedure is going to take place, see figure 2.3. After the initial covering up and cleaning of the surgical area, the surgeon is ready to begin. He initially makes an incision in the chest to expose the sternum, a bone in the middle of the chest. The sternum is then cut open with an electrical saw that does not damage soft tissue. The chest is then forced open by a clamp, see figure 2.4 a). This creates the working environment in the chest-cavity for the surgeon. While the surgeon is opening the chest he will inevitable destroy small blood vessels. To minimize the bleeding, the surgeon often uses an



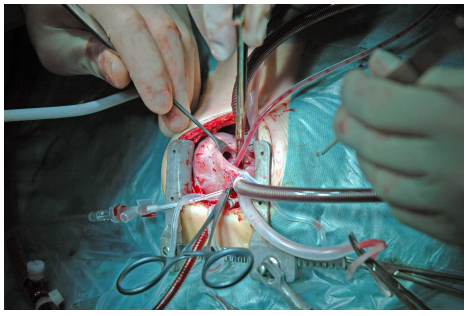
(a) Opening of the pericardium



(b) Overview with surgeons on either sides



(c) Heart lung machine attached, ready to take over the circulatory function



(d) Opening of the heart

Figure 2.4: A series of photographs taken by Doctor Ole Kromann Hansen from the initial phase of a surgical procedure

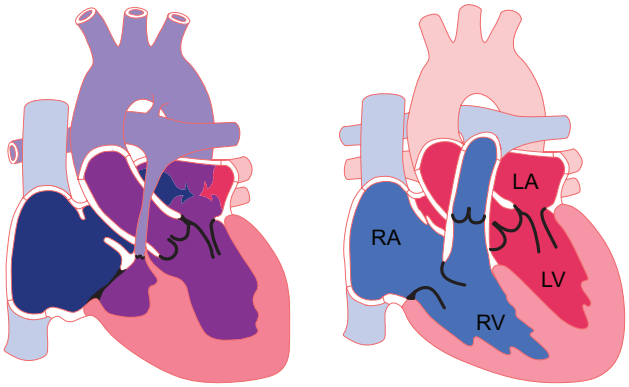
electro-surgical instrument to make incisions in the tissue and at the same time close potential blood vessels. The pericardium is opened next (again see figure 2.4 a)), and stitched to the edges of the previously opened sternum. This effectively raises the heart from within the chest, allowing for an easier access to the heart. Until now, the heart has been beating, but for the next step in the surgical procedure, where the heart is opened, it needs to be stopped and emptied of blood. The next step is consequently to connect the patient to a heart-lung machine which will take over the circulation and oxygenation of the blood. Tubes are connected to the major arteries and veins and the connections from the heart to these vessels are temporarily closed, essentially bypassing the heart, see figure 2.4 b) and c). The heart is stopped by injecting a very cold solution which hinders the electric signals for beating and contains some nutrition for the bloodless heart, see figure 2.4 d). Notice how pale the heart is in figure 2.4 d) compared to figure 2.4 c). The heart itself can now be opened safely, and the surgeon can work in a bloodless environment and on a stationary heart.

2.3.2 Ventricular/Atrium Septal Defect

The first category of defects we are going to consider is relatively simple. Remember, the heart has a right side and a left side divided by the septum. Each part again consists of two chambers, the atrium and the ventricle. When there is a hole through the septum, connecting the two atria or the two ventricles, the diagnosis is an *atrium septal defect (ASD)* or a *ventricular septal defect (VSD)* respectively. These defects arise because the wall between either the atria or ventricle does not finish forming in the development of the baby and thereby leaves a hole. In some cases the ASD or VSD will close by itself through the first couple of years, but if the hole has not closed by the age of three, there is little chance of it closing by itself. The symptoms of an ASD or VSD can be shortness of breath, easy fatigue, and poor growth. Due to pressure differences between the two sides of the heart, a VSD can cause severe problems sustaining the body with blood. Both ASDs and VSDs are most commonly closed by an open-heart surgical procedure where the hole is either closed directly with sutures or, if the size and shape of the hole demands it, is closed through suturing of a patch to the hole. This patch is made of gore-tex and will eventually be covered by the patient's own tissue.

2.3.3 Double Outlet Right Ventricle

Double Outlet Right Ventricle (DORV)[152] is our first case of a *complex* heart defect. In the case of a DORV diagnosis, both the aorta and the pulmonary artery originate from the right ventricle - from there the term "double outlet". Since the patient is still alive there is circulation; i.e. there



(a) The uni-ventricular heart (b) The normal heart

Figure 2.5: Single ventricular anomalies. In b) RA: Right Atrium, LA: Left Atrium, RV: Right Ventricle, LV: Left Ventricle. Image from [46]

must be a connection from the right side of the heart to the left side of the heart. This connection is often a Ventricular Septum Defect (VSD) as described in the previous section. The goal in a corrective surgical procedure is to do a *bi-ventricular repair*, where the left ventricle is (re-)connected to the aorta and the right ventricle reconnected to the main pulmonary artery. There are a number of different possible surgical strategies, which are often first selected during the surgical procedure as they depend on numerous morphological details of the heart. In the context of this thesis, I will only discuss a strategy using an *intra ventricular baffle*. Through a small tunnel (in this case a so-called "baffle") of artificial material, the hole (the VSD) is connected directly to the aorta. The aorta and left ventricle have thereby been re-connected as is seen in a normal heart.

The patient case we will investigate in chapter 12 is diagnosed DORV. More specifically the heart has a *subpulmonary* VSD. That is, the VSD is located below the pulmonary as seen from the right ventricle of the heart. The aorta and pulmonary artery are rotated in a side-by-side configuration as seen from the apex of the heart, instead of above each other as is normal. The aorta is furthermore to the right. That is, furthest away from the left-ventricle. The intra-ventricular baffle which has a leak is therefore difficult to repair, since it must transport blood quite a long way inside the heart.

2.3.4 Single Ventricular Anomalies

Our second complicated case is a *uni-ventricular heart*, see figure 2.5. That is, a heart where one ventricle is severely underdeveloped - and the majority of the circulatory function of the heart must be taken over by the other ventricle (from that the term "uni"). We will consider the case where the

right ventricle is the underdeveloped one, and the left ventricle has to take over the circulatory function - this defect is called *Hypoplastic right ventricle* [100]. This specific defect arises because the tricuspid valve between right ventricle and right atrium did not open in the early embryonic weeks of the baby's life. During the growth of the baby, the closed ventricular valve means that no blood is coming through to the right ventricle to force it to grow. The result is a small (hypoplastic) and weak right ventricle. By similar reasoning, a hypoplastic right ventricle also often results in a *pulmonary artery stenosis*. That is, a narrowing of parts of the pulmonary artery.

In the uni-ventricular heart we have used in the simulator an additional defect has arisen, namely a so-called *discordant ventricular-arterial connection* [144], meaning that the pulmonary artery and the aorta have switched places. Instead of the aorta rising from the left ventricle it comes from the (underdeveloped) right ventricle. Notice that this does not match the general drawing of a uni-ventricular heart in figure 2.5. In our case the patient has survived because there is a VSD connecting the underdeveloped right ventricle to the left ventricle, thereby allowing blood to go through the aorta to the body. In a previous surgical procedure this patient has had his pulmonary vein banded (clamped smaller) so that the pressure of blood in the circulation going to the lungs does not get dangerously high.

A uni-ventricular heart will generally undergo a surgical procedure to maximize the effective utilization of the single ventricle to pump blood from the lungs to the body and let the blood flow passively back from the body to the lungs - bypassing the heart. Such a circulation is called a *fontan circulation*. In the case of our patient, a specific type of fontan has been done, namely a total cavopulmonary connection (TCPC) where a tunnel is created in the right atrium and connected to the pulmonary artery.

2.4 Narrowing Down What to Simulate

In order to define precisely which parts of the surgical procedure to simulate, we have to narrow down which aspects of the real phenomena we need to represent. One perspective on this, with respect to surgical simulation, are the three levels of simulation as presented by Richard Satava [169]. Satava originally designates the three levels with the term *generation*, but I will instead use the term *level* to emphasize the *choice* between levels, depending on the problem domain at hand instead of a pure hierarchical ordering according to the evolution of surgical simulation. The term generation is important though, since the three levels also reflect the overall advance in technical development.

At the first level, only the morphological aspects are of interest as a geometrical shape. Information used to re-construct a geometric representation of the morphology is often retrieved by medical image modalities [163] such

as Computed Tomography (CT) or Magnetic Resonance Images (MRI). Although the data is often retrieved in-vivo from patients or volunteers it can also be retrieved ex-vivo based on plastified hearts which are often preserved because of interesting features. Given our definition that a simulation models a of behavior in a system, a representation on Satava's level one is to be regarded as a pure visualization. The important concepts with regards to our problem domain of a complex morphology for learning and pre-operative planning, is the navigation and immersion in the 3D anatomical datasets as opposed to looking at 2D medical images. In [174] Sørensen et al. showed that the exploration of a static 3D model is a potential valuable tool for pre-operative planning in the case of congenital cardiac defects.

The second level encompasses the geometrical shape and related concepts from level one, but includes major physical properties such as tissue deformation, bone carving, and the interaction with surgical instruments. At this level, the scenario of use can be extended to include training, rehearsal and experimentation. A system on this level is clearly a true simulation.

The third level deals with the additional *functionality* of tissue and organs, e.g. blood flow, electrical signals and even cellular mechanisms. One example of a level three simulation is the system discussed in [97] regarding the simulation of a beating heart. The idea is to simulate the tissue down to cellular size, computing the electrical signals of every single cell comprising the pulsation of the heart. The electrical signals are computed by a dozen differential equations and there are hundreds of thousands of coupled cells in a complete heart. Such a simulation could e.g. be used to test for new drugs to prevent arrhythmia since the functionality of the heart is simulated to a degree allowing for prediction of the real phenomena. The point is that knowledge exists that can explain the behavior of tissue exactly. An implementation of a mathematical model of the heart mechanics outside the scope of surgical simulation has been presented in [141]. A complex finite element based model is used as a simulation of a part of the heart beat. In reference to the fact that the research community is still working hard on real-time aspects of level two surgical simulators, a real-time level three simulation for surgery is out of the scope of this thesis in specific and currently out of the scope of contemporary research in general.

The three levels are not to be thought of as a hierarchy of better or worse tools for learning and for information retrieval - but as different perspectives on *what* to learn and what is *necessary* to learn. As mentioned earlier, one of the strengths of simulation is that we can focus on the parts of reality that are important, and simplify or leave out other parts.

The surgical procedures presented in this thesis deal with a heart that has been stopped. The main functionality of the heart is non-functional because the heart-lung machine has taken over the circulation of blood. The goal of the surgical procedure is naturally a functional one; i.e. the heart is to support the body with oxygenated blood in a more efficient manner. The

reconstruction in itself though, is very much concerned with morphological features of the heart. During planning, the surgeon will discuss questions such as "how does the interior of the heart look as seen from a specific incision?" and "can I reach certain features?". The most important part of the simulation should therefore be the soft tissue deformation in response to interactions and cutting. In perspective of the three levels this thesis deals with a level two simulation.

2.5 Categories of Usage

A surgical simulator is a tool developed and used for a specific category of use. In this thesis I distinguish between pre-operative planning and generalized training for pediatric cardiac surgery. Basically, the distinction is whether or not the simulator is based on patient-specific models (mostly for experienced surgeons) or whether it is based on generalized pedagogical models (mostly for novice surgeons). Within both categories of use, the user should be allowed to train, rehearse, experiment, and explore to aid both the users understanding of the morphology and related surgical procedures. In our preliminary evaluation of the surgical simulator in chapter 12 these two categories are revisited with specific heart models for each scenario. Although the discussion in this section is valid for a range of surgical procedures, I will focus on congenital heart defects.

2.5.1 Pre-Operative Planning

The initial need, as described by the pediatric surgeons of Aarhus University Hospital, was a tool for them to do pre-operative planning. In this case the simulator would be based on the image data of a patient-specific heart, and the surgeon would rehearse the procedure or explore the morphology of the heart.

Pre-operational planning of a surgical procedure is normally done based on ultrasound and catheterization guided by x-ray. In cases of complex defects it can be necessary for surgeons to acquire 2D image data through MRI. The strategy to be used for a specific surgical procedure is tightly coupled to the morphology of the heart, and the surgeon must therefore translate the 2D images to a mental model of a three-dimensional heart. The surgeons at Skejby are quite far ahead in this area, because they already use an explicit representation of the 3D morphology through a volume rendering of 3D MRI [173] or through a three-dimensional geometric model [174] acquired by segmentation of the MRI. This tool alone gives the surgeon a better understanding of the morphology. What is still missing, is the relation of the heart shape to the surgical strategy in which the heart is actually manipulated. When the surgeons are planning for a surgical procedure today, the surgeons must mentally go through the procedure. A surgical simulator is

a tool to more explicitly go through the procedure and explore the heart. When the surgeons analyze information from 2D or even 3D images of the heart, one important aspect is missing; the deformation and interaction he normally experiences when analyzing the situation in an actual surgical procedure. The surgeon is not used to looking at pure geometrical models - the surgeon looks at models that deform in real-time while he investigates them and changes topology based on incisions. He is simply more experienced at making decisions based on what he sees in actual surgery, which is an open heart manipulated with surgical instruments. E.g. when a surgeon looks at an atrial septum defect (ASD) in actual surgery, he decides what to do based on the location of the hole in the atrium septum and the incision he has made into the myocardium. The surgeon might also deform the heart to get a clear view of the septum. A surgical simulator should ultimately be able to present the surgeon with the same image of the open heart as real surgical procedures, and based on this he could make the same decision pre-operatively as in the actual surgical situation and thereby be better prepared. A static geometry of a closed heart does not in that sense resemble what the surgeon experiences.

The simulator can also be used to ensure that a given procedure can be executed as planned by giving the surgeon information about e.g. the level of stress that the tissue is exposed to or whether a given piece of tissue can cover a hole or be reconstructed and fitted into a given shape. These last issues are out of scope of this thesis though.

2.5.2 Education and Training

Our second scenario is for education and training in a more general setting of both novices and experts, although novice surgeons are of most interest initially. In recent years surgical simulation has begun to gain clinical respect and is predicted to be an integrated part of the training to become a surgeon [126].

In the pre-operative case the simulation is per definition based on real patient data and as such is limited by the information that we are able to retrieve from the range of medical images and the time available to build virtual models. In a general educational scenario we are not limited in the same sense and can more aggressively acquire accurate models, do more time-consuming post-processing and do manual changes to enhance and simplify as necessary in relation to the educational case. I deal with the construction of virtual models in relation to our simulator in chapter 11. In some cases the educational scenario might not look anything like the real surgical procedure but still teach the basic elements of the surgical procedure. As an example, the Mist system [83, 179] can use pure geometrical figures to teach hand eye coordination. The training scenarios must be targeted to the aspects of the surgical procedures that are to be trained. It is

not necessarily the ultimate goal to just simulate reality. E.g. in [167, 166] a risk reducing training is set up. Risk estimates are used to avoid damage of important tissue. In the training scenario the student can "feel" the risk areas through haptic feedback. According to [166] brain and cardiac surgery are areas with well-defined areas of risk.

In section 2.2 current teaching of surgical procedures was presented. A simulator could support this learning process with training in virtual reality. A simulator could be used as a supplementary tool in medical curriculum and as a tool to transfer knowledge from master to student. The teachers might use the simulator to present a procedure or technique, and the students could later try out the procedure for themselves. Used as such, a surgical simulator has the potential to be a benefit within the following areas:

An alternative to cadavers and animals

A surgical simulator can be used as an alternative to surgical training on cadavers or animals. In [104] a surgical simulation is presented as an alternative to animals and cadavers in a course in advanced trauma life support. The problem using animals is that they do not have the same anatomy as humans. Cadavers have the correct anatomy although tissue properties change when the tissue is dead, the cadavers are expensive, and can be difficult to acquire. Neither cadavers nor animals are reusable and represent a single case of disease (if any). Using cadavers and animals is also a source of serious ethical issues.

A tool for sharing of knowledge

New or rare surgical procedures can be recorded by experienced surgeons, thereby sharing their knowledge with other experienced surgeons or students. Whole libraries of knowledge could be constructed. The user can watch the procedure from any angle and can take over control of the simulator at any time. In this case, a patient-specific heart with rare conditions would be used.

Minimizing risks to patients

By allowing the young surgeon to train on a surgical simulator and generalize the acquired knowledge to the real surgical procedure, he can reach a higher level of proficiency before coming into contact with real surgeons. Prior to a surgical procedure he can furthermore rehearse the procedure on a patient-specific heart.

Represent arbitrary defects or anatomies

As mentioned earlier there are more than 3000 different diagnosises in congenital cardiovascular defects, this can make it very difficult to reach a level of proficiency within any area, apart from the most common diagnosises. The fact that defects often have an aspect that makes them almost unique, means that it can be very difficult to reach a certain level of proficiency with a specific type of defect. Since the simulator is based on virtual models, unique defects captured in a medical scan can be circulated and trained on an infinite number of times. The virtual models can furthermore be altered through post-processing, thereby generating any desired defect. A simulator would consequently enable surgeons to train on any defect imaginable, thereby reaching the required level of routine.

Be allowed to make mistakes during training

When the student engages in a master-apprentice relationship working on real patients he must always listen to the instructions of the master-surgeon. Any failure, or sum of small mistakes, can in the end be fatal to the patient. The student must always trust the experience of the master-surgeon. In a simulator, the student would be allowed to learn from his own mistakes, thereby getting a better feel for the thin line between "possible" and "impossible" in a given surgical setting. The simulator could allow the student to try out something, rewind and try again - something which is unfortunately not a feature of the real world.

The user is not pressed for time

In real surgery the surgeon must work under a strict time schedule. The heart in a heart-lung machine does not receive oxygen and other nutrient through the blood while it is operated on - although it does receive nutrients through the liquid injected into the coronary arteries. This fact is a great constraint on the time allowed in a surgical procedure. In general, a prolonged time in anesthesia and open surgery results in a increased risk of complications, both during surgery and recovery. Working on cadavers or animals the student is also pressed for time by the other students. The simulator allows the student to train as long as he needs, and potentially at his own personal computer at home.

Skill assessment

Especially in the US skill assessment using surgical simulators has been proposed as a way of grading people or selecting people for surgical career [170, 96]. The continued "grading" of people could be used to make sure that surgeons stay at a certain proficiency level.

Part II

Surgical Simulation

Chapter 3

An Overview of the Research Field

This chapter presents the different themes of interest when we consider the technical aspects of a surgical simulation. At TATRICS 3rd annual presentation Dr. Kevin Montgomery derived some of the common themes in surgery simulation research based on 24 different groups working within the field [132]. The survey by Liu et al. [120] fits well together with Montgomery's perspective and also gives a good overall presentation of these themes. The themes selected for presentation in this thesis are: *Datasets and segmentation, representation, simulation engine, visualization and display,* and *interaction and haptics*. The following sections will serve as an overview of the technical themes involved in the remaining thesis and will narrow down the focus of the thesis compared to the previous chapter.

3.1 Datasets and Segmentation

The type of simulators dealt with in this thesis is related to the shape and physical characteristics of organs and tissue. To build realistic virtual models we need to base them on real human anatomy. Standard medical image acquisition techniques [163] include Magnetic Resonance Imaging (MRI) based on magnetism and Computed Tomography (CT) based on x-rays. Since CT is based on x-rays, an image acquisition in this modality means that the patient is subject to a radiation-dose. MRI on the other hand is risk-free. CT often gives images with better quality and resolution - especially the difference between bone and tissue is evident, although MRI is better at distinguishing between different types of soft-tissue.

Often the datasets are divided into two categories according to the scenario of use; patient-specific or general datasets. Working with acquisition of real patient datasets means working with real patients and therefore a number of constraints apply. First, the time we can use to acquire the datasets

and the variety of modalities we can acquire are limited due to patient-comfort and safety. Secondly, the total resources used for a large number of patient-specific datasets, limits the amount of manual work we can put into each dataset. Very long post-processing times can often not be accepted either, since this could delay the treatment process of each individual patient. General datasets can be acquired from volunteers, in which case time and resources are not a serious problem. The most aggressive acquisition of general datasets can be performed when deceased donate their bodies to medical research. In that case both long post-processing times, high-dose radiation, and destructive acquisition can be allowed. The prime example of this is the Visible Human Project [14] where a donated male body has been acquired with CT, MR and cryosection (slicing of the body in millimeter thick slices and photographing each slice). A number of projects following the Visible Human Project have arisen; the Chinese Visible Human [199], the Korean Visible Human [150] and the Visible Ear [172] to name a few.

Segmentation [163, section 5.3] is the process of dividing the image data into regions according to some measure. In the case of medical image segmentation we seek to identify the different parts of anatomy. The result of segmentation is a volume of voxels from which we can find a surface of triangles through e.g. the marching cubes algorithm [121].

In the case of children with congenital heart defects, CT is most often not used since the radiation-dose involved would incur a significant risk of radiation-related disease later in life. The beating of the heart is an issue since this introduces severe motion artifacts. Therefore MR acquisition must be triggered to a specific window of the heart-cycle. A smaller window results in less motion artifacts, but a larger amount of noise necessitating a prolonged scan-time. The scan-time is important since the children are anesthetized to lie still in the scanner - and this in itself should not be prolonged any more than is strictly necessary. The breathing of the children is another problem, since breathing shifts the entire chest up and down. These artifacts can be corrected for by a so-called navigator based scanning. Our data-acquisition and post-processing is presented in chapter 11.

3.2 Representation

The *representation* of tissue as a data-structure in the simulation is naturally deeply connected to the simulation engine, which I will cover in the next section. Essentially it is a conversion of the output from the segmentation to an efficient data-structure. Both simulation and visualization use a representation of the tissue shape as a basis for calculation. The involved data-structures must consequently be judged flexible and efficient in that perspective. Since the requirements for simulation and visualization may differ, so does the optimal representation of each modality. This fact must

be related to the the tight connection between visualization and simulation. More details on our representation is found in chapter 8 on simulation and 9 on visualization.

3.3 Simulation Engine

The category of simulation engines I consider deal with the modeling of biomechanical properties of tissue, and the numerical methods used to calculate the tissue deformation. Often a trade-off must be made between various aspects of *geometrical detail*, *computational speed*, and finally *realism and precision* within the perspective that computational power is limited. The simulation engine must also take into account various constraints opposed by interaction and visualization. If interaction includes cutting for example, it is in many cases a considerable constraint on the choice of method for calculating tissue deformation. This issue will be discussed more in chapter 5 in relation to specific models of deformation. The main focus of this thesis is on real-time interaction with very complex morphology and a consistent and fast deformation. Hence our priorities are - in no specific order:

- Geometry (model complex morphology)
- Speed (convergence and update rate)
- Robustness (consistency, stability and realism)
- Visual result (realism and graphics)

I expand on the above bullets in the following paragraph. It is essential to this project that the *geometry* of the virtual heart can accurately model the degree of detail needed by the surgeons. If we cannot represent that, nothing else matters. In the case of the heart, it requires a relative large amount of detail to represent the complex morphology. The *speed* of the algorithms is essential to the real-time aspect. If the algorithm cannot deliver some result within one-twentieth of a second, the user will experience too poor a frame-rate to obtain the illusion of smooth animation. *Robustness* covers aspects such as consistency of results, stability and realism of the deformations displayed. It is intentional that realism is only part of the robustness demand. If we had unlimited computational power, absolute realism would be equal to robustness - but because computational power is at a shortage and realism can only be approximated, other terms are important too. A simulation should be realistic enough for it to be useful for *the selected* scenario of use as discussed in 2.5. We can relax the degree of realism to a *believable* deformation within a given context; including consistency and stability. It is essential that the range of deformations is consistent so that we can have clear notion of what kind of accuracy to expect from the simulator. To quote Bro Nielsen:

> "It doesn't really matter whether the deformation that the surgeons see in the virtual environment is accurate as long as it seems realistic! Just as important is that the model is robust and shows a consistent and predictable behavior over time" [29]

Lastly I deal with the issue of *visualization*. Since representation of complex morphology is essential to our problem-domain, the visual representation of this morphology becomes important too. An important point here is that the visualization and simulation need not be of the same level of detail, but must each be judged accordingly to the scenario of use. In our case, we require the visualization to include details and landmarks that are important to the exploration and spatial understanding of the heart, but not necessary in the simulation of the heart. In a pre-operative scenario, the visualization must be based on acquired patient-specific data, but in a more general case the "suspension of disbelief" becomes important to the educational process. In our case we have carefully constructed a realistic visualization including an operating environment and instruments.

In the survey by Sarah Frisken from 1997 [81], a range of different models for the computation of deformable models are presented. In general, very different models exist, both pure *geometrically* and *physically based*. In Liu et al. [120] the geometrically based models are called *kinematic based models* but still refer to the same category. In the survey by Meier et al. [128] they refer to the same category now as *heuristic models*. While both Gibson [81] and Liu [120] categorize the *Spring-Mass model* (which we will look at in the next chapter) as a physically based model, Meier [128] categorizes it as a heuristic model. The geometrical models are most clearly defined by their lack of an explicit representation of mass, force, or other physical properties. These deformations are often fast - but have no justification in real physics. Examples include splines, patches and free form deformation. This thesis deals with the other category, physically based models. A line can not be drawn clearly, but we can order the methods as to how well they approximate the real physical phenomenon, and to what degree they are merely geometrical heuristics. The chain-mail algorithm [80] by Gibson is an alternative model of deformation not directly based on physics. The algorithm favors size of geometry and speed of computation, but the deformations represent realistic tissue deformation to a lesser degree than physics based models. Amongst other things the chain-mail algorithm was used for the simulation of knee surgery in [79]. In chapter 5 I will introduce two of the most used models of deformation and the mathematics used to solve the involved equation; finite element models and Spring-Mass models. In part III, and specifically chapter 8 our simulation engine on graphics hardware is presented.

3.4 Visualization and Display

The subject of visualization and display deals with the strategies used for visualization and the technologies of display hardware. We narrow down the field to deal with real-time visualization of the calculated tissue-deformation. Visualization in this context is concerned with sub-themes such as visualizing the surface through a 3D graphics API, shading and special effects. Surgical simulators have, as most other interactive 3D applications, used the contemporary 3D API such as Direct3D of DirectX [129] and OpenGL [195]. In most cases these APIs are used as intended, with animated vertex positions streamed from the CPU to the graphics card. Most basic shading effects such as phong based lighting give a basic impression of the shape of a 3D object. In cases where an additional "suspension of disbelief" is wanted, additional shading effects can be employed, such as texture mapping, more elaborate lighting models, and shadowing. In some cases additional special effects have been used to create more realism. In [15] blood and water was included as special effects. In [28] visual effects were used to cover an endoscope with attached tissue and generate spray of local anesthetizer. In [200] a Navier-Stokes based simulation of blood is used to occlude the endoscope. In [75] a soft tissue simulation is improved visually through bump-mapping and shadows cast by the instruments.

Our surgical simulator utilizes the OpenGL 3D graphics API for visualization (but also for the simulation). We will return to this subject in great depth in part III, so in this section we will simply state that the visualization features both texture-mapping, normal-mapping [66] and shadow maps [63].

3.5 Interaction and Haptics

The field of interaction covers the instrumentation of tissue; e.g. probing, grasping, piercing and suturing. This covers both the physical devices, collision detection, and response of soft tissue [56]. Two aspects have been given special interest by the research community over the years; cutting and haptic feedback.

3.5.1 Cutting

Cutting is an essential task in many aspects of surgery. Basically, cutting into tissue allows the surgeon to enter the human body in open surgery. In surgery on a heart with a congenital defect, cutting is an inherent part of the surgical procedure since making incisions is often part of the basic strategy to re-shape the heart.

Two aspects of cutting with respect to simulation are important; the simulation engine must support real-time update of the simulation geometry, and the updated geometry must accurately reflect the desired incisions.

Figure 3.1: Phantom Omni haptic-interaction device from Sensable (image from sensable.com)

Our simulation engine supports arbitrary topological changes in both simulation and visualization and although the methods implemented for modifying the topology are rather simple, more elaborate methods exists. As stated in [146] the main issue in cutting is that e.g. tetrahedral and triangle meshes are not closed under the cutting operation. A triangle that is cut along a sweep does not simply result in another set of triangles. The mesh of primitives must be adapted to make the cut appear where the user made the cut. The simplest method is to remove the basic elements intersected by the cut-sweep (e.g. used in [70, 57]). This is clearly not very realistic, but can be repaired somewhat if nodes along removed elements are moved to the cut-sweep. In the other end of the scale is a complete subdivision of the basic elements, in [20] a general subdivision of a tetrahedral is presented. Problems here are that more elements are created with a slower performance of the calculation of tissue as a result. Furthermore issues of degenerate elements become evident. These can consequently be repaired through post-processing [74]. Finally Stappen and Nienhuys [146] have proposed to split meshes along faces (in a mesh of tetrahedrons) or edges (in a mesh of triangles) and snap nodes to the original cut-sweep.

In relation to our contribution of decoupling simulation and visualization in chapter 9 these issues get another dimensionality as visualization and simulation can have different cutting schemes (as is the case in our simulator). This decoupling would enable us to use the most optimal cutting scheme for each representation, e.g. the Delaunay based on-line cutting-scheme by Stappen and Nienhuys [145] for the triangle-base mesh of the surface, since this method is not generalizable to higher order elements which the simulation consists of.

3.5.2 Haptics Rendering

In the introductory text on haptics-interaction by Salisbury et al. [168] it is stated that haptic[1]-interaction is a very important component in VR applications, in line with the importance of visual and auditory components. Whereas visual and auditory media are one-directional from the media to user, the haptic device is bi-directional in nature. The haptic device both receives interaction input and responds with forces. According to [168] this is often referred to as the single most important feature of the haptic interaction modality.

A given haptic device can be characterized by its *interface points* to the human body, the *grounding force* from which forces are applied to the interface points and the *degrees of freedom* in force feedback and positional sensing. The haptics device used in this thesis and many other surgical simulators, the Phantom Omni (See figure 3.1), has one interface point which resembles a pencil, is grounded on the table (and can therefore reproduce weight of objects), has 6DOF in positional sensing but lacks pitch, roll and yaw in force feedback. The Phantom Omni consequently delivers 3DOF force-feedback to the user. Another example, the CyberGrasp Exoskeleton by Immersion, attaches to each finger an actuator with 1DOF of motion and force. Each actuator is grounded on the back of the hand. The system architecture of a haptics rendering module means that the simulation engine must support effective collision detection between a virtual instrument representing the interaction device in VR and the simulated tissue. Following collision detection comes a force response, often calculated on the basis of interpenetration between instrument and tissue. In many areas of surgery the feedback from tissue is essential. This provides an additional challenge as force feedback must be provided at least at 500 Hz to feel smooth. Comparing this to the 30hz needed for the human visual system to accept a series of images as an animation shows how sensitive the sense of touch is. To achieve such an update rate from a simulation running at a much lower frequency, interpolation schemes have been developed, e.g. [125, 155].

The various Phantom haptic devices are often used in surgical simulation [27, 190]. One problem with a physical attachment of the "instruments" is that certain moves can be constrained by the physical arm itself. With this issue in mind [98] has proposed and developed a visual tracking system with haptic feedback through magnetism.

Haptics rendering is potentially very important in surgical simulation. Ro et al. [162] discuss their finding that a group of experienced laparoscopists performed worse than a group of novices on a commercially available laparoscopic trainer. Their conclusion is that the experienced group (who where not experts) rely on the haptic feedback to successfully execute the procedure, which was not available in that particular commercial

[1] From the Greek *haptesthai*, meaning "to touch".

laparoscopic trainer.

In the case of our simulation, the actual forces felt in real surgery on the very small heart of a child are actually quite limited. The heart of a small child simply cannot withstand very large amounts of forces. The surgeons must be very careful in every movement. We have implemented force-feedback nonetheless, and actually exaggerated the forces. Our reasons for doing this, in the context of heart surgery, is that force-feedback can also be used to constrain the amount of deformation users are allowed to affect the heart with, thereby limiting the amount of instability introduced through the use of excessive forces. We will deal specifically with enabling haptics in the GPU based simulator in chapter 10.

Chapter 4

Surgical Simulators

Having a basic understanding of the components of simulator from the previous chapter, we will now look into the categories of surgical simulators developed and the issues in each of those categories. In the survey by Liu et al. [120] three categories of simulators are identified; *needle-based procedures*, *minimally invasive,* and *open surgery.* This categorization reflects the evolution of surgical simulation as a field, although every category is still very much open to research and technical innovation. In the next three sections 4.1, 4.2 and 4.3 we present important surgical simulators within each of the three categories. In section 4.4 we present open source and open framework initiatives for surgical simulation. Finally, in section 4.5 we look shortly at validation studies in relation to specific surgical simulators.

4.1 Needle-Based Procedures

Needle-based procedures deal with the task of controlling the insertion of some needle-type instrument into the human body. Examples include vascular access, catherization, biopsy and anesthesia. This kind of simulation might at first glance seem simple; the degree of freedom in interaction seems very limited and the demands for visualization are in many cases quite limited. One area of specific interest for these simulations is the haptic feedback, which in many needle-based procedures is key to a correct execution. Once the needle is inserted into the patient the interaction is often restricted to 1DOF in the direction of the guidance of the needle. In some cases special-purpose haptic equipment has been built taking into account this limited interaction, but in many cases general 6DOF equipment has also been used.

Some of the first prototypes of needle-based procedures focused on the haptics and to a lesser degree visualization and general simulation of tissue. In [26] a simulator for surgical needles was haptics only, without any additional visualization. Later work such as the CathSim simulator [185] has both very realistic graphics and custom haptic, see figure 4.1. In the Cath-

Figure 4.1: CathSim system for needle insertion. Image from *hpsimcenter.com*

Sim the users can feel a realistic "pop" from the puncture of a vein inside the skin. The visualization supports different ages, skin color and health condition. This project was commercialized as the CathSim AccuTouch System by Immersion Medical.

The work by Stephane Cotin et al. [52] presents a recent high-fidelity simulator for interventional radiology, in which a guide-wire catheter is inserted into the artery network of a patient, monitored through live x-ray images. The simulator includes real-time deformable models (FEM) and handles the many concurrent collision points between guide-wire and vessel. The visualization simulates x-rays through the deformed tissue resulting in realistic x-ray images.

In [119] a general framework for haptic based needle simulation is introduced based on a finite automata representation of states. In [118] this framework is used specifically for a simulator for an emergency procedure where intra-abdominal bleeding is suspected.

4.2 Minimally Invasive Surgery

Minimally invasive surgery with endoscopic[1] equipment is a specific branch of surgery in which a minimum amount of incisions into the patient is sought after. Instruments are inserted through small incisions in the skin of the patient and the instruments can thereby access organs and internal tissue. These surgical instruments are special-purpose in the sense that they are built for minimally invasive surgery. The instruments are often long and cylindrical with a small active part in the end that can either cut or grab.

[1] Of Greek origin meaning "looking inside" or "in-sight"

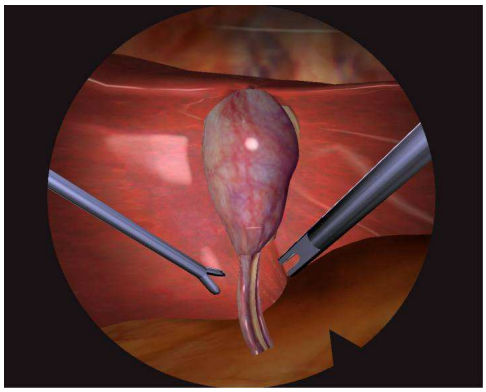


Figure 4.2: The Karlsruhe endoscopic simulator. Image from *www-kismet.iai.fzk.de*

By nature these instruments have 4DOF. The visual feedback is provided by another instrument-shaped camera-scope. The camera-scope is inserted into the patients and a video-feed can be monitored on a nearby screen. The difficult issue in these kind of surgical procedures is that the instruments have a natural pivot point around the insertion into the patient. The surgeon must consequently move the instruments in the opposite direction of the intended movement inside the patient. Furthermore, the instruments have a very limited and strictly confined work-area. Mapping such procedures to a computer simulation is relatively well defined. Interaction with all real organs is mediated through the special instruments and no direct interaction is possible, consequently the real instruments (or replica) can simply be augmented with sensors and actuators. The visual feedback from a simulation can simply be output to the existing screen in the operating theater. A large number of minimally invasive surgery simulators have been developed, in this section I present only the most well known.

The Karlsruhe endoscopic simulator [107, 113] (based on the commercially available kismet[2] environment) is a complete environment for minimally invasive surgery. The interaction device is a custom built artificial cavity with endoscopic instruments as well as a standard monitor. All internal organs are simulated through a Spring-Mass system or a FEM based model with the additional feature that arteries have a pulse that can be felt through the haptic enabled endoscopic instruments. Arteries can also be ruptured, leading to arterial bleeding. The system has been used for both abdominal procedures as well as gynecology. This system has been commercialized through the Select IT as the "Vest System One".

The LASSO project [180] developed a simulator for laparoscopic gyne-

[2]Kinematic Simulation, Monitoring and Off-Line Programming Environment for Telerobotics

cology. The basic data was built on the Visible Human Female data [14]. The project implemented a number of algorithms for deformable modeling, amongst others the Spring-Mass, FEM and chain-mail. The visualization, including generation of textures, vessels, pathological variations and lighting, has been given special emphasis resulting in very realistic appearance of the surgical simulation. Their initial system used the Phantom force feedback devices although the project goal is to use custom endoscopic instruments.

The PreOp Endoscopic Simulator by Bro-Nielsen [28] was used for a range of endoscopic procedures such as bronchoscopy. The system was commercialized by HT Medical Systems (later acquired by Immersion) and was reported to be one of the first "complete" systems in the sense that a range of supporting multimedia content for education was available.

The MIST-VR system by Mentice [83, 179] is used to teach general skills of minimally invasive surgery. In a realistic environment the user can learn how to suture and tie knots. The previous sequence of minimally invasive surgery simulators represents a gradual increase in realism. The MIST-VR system takes another approach through the "core-skills" set of exercises. These exercises use the interaction with simple geometrical shapes to teach basic techniques of inverted interaction and hand-eye coordination. This system has been validated in a number of studies [171, 151] showing that these "simple" exercises can generalize to operating room proficiency.

4.3 Open Surgery

According to the survey by Liu et al. [120]:

> "Open surgery remains the holy grail of surgical simulation. Considerable advances in haptics, real-time deformation, organ and tissue modeling, and visual rendering must be made before open surgery can be simulated realistically" Liu et al. 2003.

Generally, the degree of difficulty in the challenge of open surgery simulation depends on the chosen surgical procedures to simulate. The move from minimally invasive surgery to open surgery does not in itself transfer to a more advanced simulation as such. Open surgery has more potential freedom in interaction and visualization though, often transferring to more encompassing models of human anatomy and more general models of interaction. One severe constraint in a general open surgery simulator is the goal of complete visual and haptic immersion. There is no clear interface to model as in the case of minimally invasive surgery. Ideally, the surgeon should be able to feel the tissue between his fingers looking at the patient in a completely natural way. This goal has not been reached by any project to date though.

Although our contributions are within open surgery we work under the assumption that the surgeon can manipulate tissue to a sufficient degree

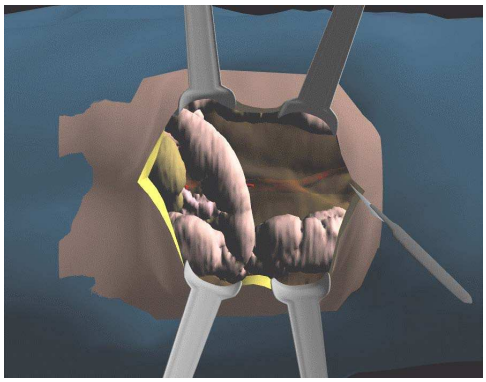


Figure 4.3: HT Abdominal Trauma Simulator. Image from [27]

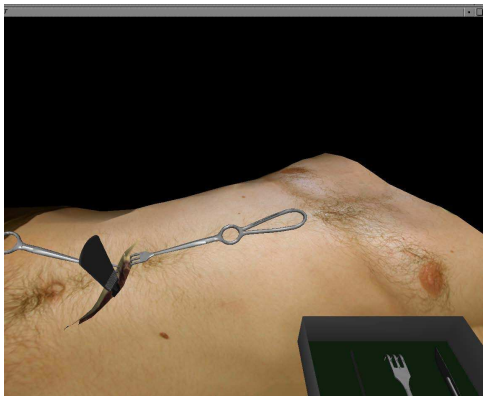


Figure 4.4: Open Surgery Simulator. Image from [19]

through instruments. This fact has been confirmed by our surgeons as a valid approach to their branch of open surgery. Direct manipulation of tissue with fingers is a natural option in surgery, and also used in heart surgery on children - but not as often as one should think, since the heart is actually so small that the fingers and hands can only do rather crude manipulations. The heart might be rotated slightly with the hand or the surgeon might feel with a finger through an incision in the heart to determine tissue thickness or accessibility inside the heart.

Often open surgery is advocated in papers presenting general techniques such as incisions, suturing or handling tissue. This corresponds with the idea that open surgery allows more freedom in interaction and consequently demands more general methods to deal correctly with any interaction.

In 1998 Bro-Nielsen et al. [27] described the HT Abdominal Trauma Simulator for open surgery, see figure 4.3. This simulator is a PC based soft tissue simulation with instrumental interaction of tissue through Phantom haptic feedback devices. Basically the system supports doing incisions

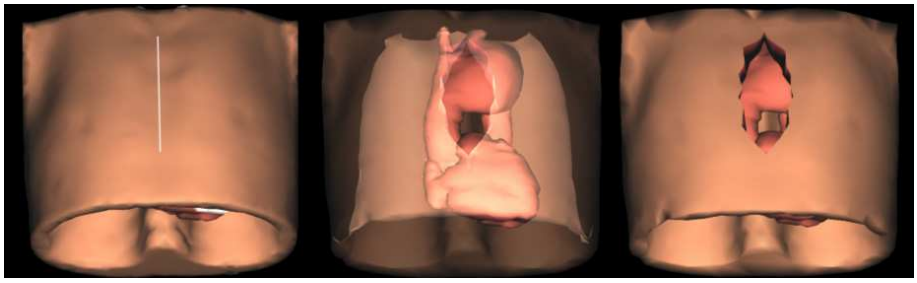


Figure 4.5: ActiveHeart by Nakao simulating the initial incision into the chest. Image from [140].

into the skin and using instruments to pull back the skin to get access to the abdominal region. The simulation is supported by multimedia content since Bro Nielsen et al. realized that not all aspects of open surgery can be simulated. In 2001 Webster et al. [190] presented a PC based suturing simulator for wounds. The system supports haptic feedback through Phantom Omni devices. In 2002 Bielser et al. [19] presented what they called an "Open Surgery Simulator", see figure 4.4. The system could basically simulate interaction between skin and surgical hooks or surgical scalpels, but without any abdominal content as in [27]. In [75] a system for hernia repair is presented, supporting knives and clamps. The system uses a non-linear Spring-Mass system to accurately create the basis for haptic interaction. The system uses Phantom interaction devices.

In the PhD thesis by Nakao [140] he presents his ActiveHeart system where one can practice the initial incision into the chest and palpation of the aorta in conjunction with a pure visualization of the heart, see figure 4.5. Our work fits nicely into this previous research as we investigate the actual surgical procedure after the opening of the chest.

4.4 Medical Simulation Frameworks

It should be evident by now that a huge amount of different simulators for surgical procedures have been created. In many cases there is a considerable amount of re-implementation by many different research teams. Especially work on basic FEM and Spring-Mass based cutting and haptics feedback (on the CPU). Accordingly a number of research projects have tried to establish a common framework with medical simulation to build on.

The SPRING Surgical Simulator [133] by Dr. Kevin Montgomery is a general open-source platform specifically targeting areas of limited access surgery. The SPRING platform can load data in a custom format consisting of mesh-geometry, graphical attributes and tissue-material attributes. The simulation engine itself is based on a Spring-Mass model coupling nodes

to vertices and springs to edges. Interaction and sensor support has been given special focus in this platform [135, 134] with a wide range of different categories of virtual instruments coupled with a wide support of actual interaction devices. The SPRING platform has been used in a number of internal projects but is openly advocated as a prototyping platform for other researchers.

The GiPSi [45] (General Physical Simulation Interface) is an open source and open architecture framework meant to facilitate the open exchange of models and algorithms within organ level surgical simulations. Furthermore this common framework is meant to further the interoperability of medical simulators in general since they will be built on the same framework. Of specific concern is the interface between different dynamic models in a general way.

The SOFA project [53] (Simulation Open Framework Architecture) is an open platform for medical simulation systems in much the same spirit as the GiPSi project, enabling component sharing and optimizing development time. One specific interesting aspect about SOFA is the multimodal representation of behavior model (simulation), collision model, haptic model and visual model respectively. These modalities of representation are connected with mappings that allow e.g. the collision model to deform based on the simulation but still be defined separately.

The OpenTissue Library [147] is an API for physics based animation in general. A range of different algorithms have been implemented within the categories of multi-body dynamics, particle systems, water simulation, level set methods, deformable objects, and collision detection. In comparison with the previous three projects this library does not present itself as a complete framework for surgical simulation projects specifically, but as a general physics based animation library that could probably be used in any of the previous frameworks.

4.5 Evaluation Studies

It has been recognized [123] that the next big step in surgical simulation for education scenarios is a formal verification of the usefulness of training with surgical simulators. In [170] Richard Satava presented the progress in the Metrics for Objective Assessment of Surgical Skills Workshop. The PhD thesis by Holbrey [95] presents a good overview and many good references for validation studies both within training and skill assessment. In the report on the metrics for objective assessment of surgical skills by Satava [170] a number of criteria for validity of surgical simulators are set up. Gallagher et al. [72] and Seymour et al. [171] have proved one important type of validity, construct validity, for the MIST system previously mentioned. That is, that experts and novices can be distinguished by their score on the system.

Pearson et al. [151] showed that the complex task of tying a knot inside a human could be learned from the MIST system and transferred to the real situation. [115] showed how a VR based training on their custom simulator was comparable to conventional video based training. A basic discussion is naturally the metrics used. E.g. [115] use execution time as the basic score, while [69] used a metric of "error reduction".

Apart from a validation of the whole simulator as such, the basic technical elements of the simulators can be validated individually. In [178] user's sensitivity of haptics is investigated to show to what degree users can distinguish and identify a given soft-tissue parameter. The deformable models themselves have also been a target for much investigation. Two aspects are of importance; Instruments for measuring real tissue response (e.g. [38]) and validation of real-time calculation of the simulated soft-tissue response. As an example the Truth Cube [106] is a publicly available dataset of a series of deformations of a material with inlaid markers. These markers are tracked in 3D and thereby represent a volumetric displacement field that can be compared to deformable models. [105] experimentally verified that Spring-Mass and finite element models behaved alike for small deformations within the domain of craniofacial surgery.

In chapter 12 we present a preliminary evaluation of our surgical simulator as a training and pre-operative tool.

Chapter 5

Calculating Deformation

The contributions of this thesis are derived from open heart surgery in which deformation of tissue is one of the most important single elements to simulation as explained in section 2.4. In this chapter we look at the existing theory on the calculation of deformation as well as within which categories of surgical simulators it has been used. The survey by Sarah F. Gibson [81] and later Montanga [131] deal in general with the calculation of deformation. Meier et al. [128] have recently done a survey in many of the same topics but with special emphasis in surgical simulation.

I will give a short introduction to two popular categories of methods, finite element based methods in section 5.1 and Spring-Mass based methods in section 5.2. A comparison of the two methods is presented in section 5.3 which leads up to a general discussion of major issues in surgical simulation specifically with regards to a simulator for congenital heart defects in section 5.4.

5.1 Finite Element Method

Using continuous models of physics for computer animation was introduced by Terzopoulos [182]. In surgical simulation the Finite Element Method (FEM) is often applied to linear approximation of certain material properties, to quickly find tissue deformation based on a continuous model of elasticity. In the context of this thesis we will use the abbreviation FEM or the term Finite Element Method to refer to the specific analysis presented below, although the method is general in nature. The use of finite element analysis in surgical simulation was initiated by Bro Nielsen in [30] and has later been used in numerous applications and numerous variations.

Continuous equations that govern the behavior of soft body dynamics can be constructed but are not easily solved. Analytical solutions can be found for simple cases, but for complex cases we must use numerical methods to discretize and solve the problem. Finite element analysis is a method to

solve these kinds of equations. In the rest of this thesis we look only at the finite element model as a tool to calculate deformations in soft tissue, although a range of other applications are possible.

Finite element analysis can be used to solve the involved equations by decomposing the original three dimensional problem domain (the organ) into a finite set of basic geometric elements, often tetrahedrons. Based on these elements the involved differential equations can be solved approximately.

The main advantage of FEM (compared to the later Spring-Mass e.g.) is that we calculate approximate solutions to the actual equations of deformation in theory of elasticity. The main problem with FEM in the scope of real-time simulation is to make it run fast enough. Several techniques exist and will be presented in the subsequent chapters.

The subsequent sections on FEM are arranged as follows; in 5.1.1 the Theory of Elasticity, which is the basis of the equations to be solved by FEM, is presented. The energy function is presented in section 5.1.2 and discretization into finite elements is presented in section 5.1.3. Numerical techniques solving the resulting system of linear equations is presented in 5.1.4. For more-in depth presentation I refer to the authors master's thesis [1] and source material for finite element analysis [30, 81, 99]. Lastly in section 5.1.6 a number of surgical simulators based on FEM are referred.

5.1.1 Theory of Elasticity

The theory of elasticity [161] is a part of continuum mechanics that deals with the prediction and calculation of the effect of applying an external load on some body with elastic physical characteristics. That is, materials that returns to their original configuration when the external load is released. When studying the relationship between forces and deformation, some of the concepts we need to define are *stress, strain, equilibrium,* and *displacement* [197]. Stress is the strength of the force from interaction such as stretching, squeezing or twisting. Often stress is characterized as "force per unit area". *Strain* is the resulting deformation in the material. The stress-strain relationship defines how tissue deforms under a given force. When forces are applied to the tissue it instantly deforms to a configuration in which the internal "energy" of the tissue is in *equilibrium* with the external energy. The information about the tissue we would like to know for computer animation is the *displacement* of the nodes in equilibrium given some external force.

One of the simplest models of elasticity is the static reversible elastic deformation with a linear stress-strain relationship - often just called the linear elastic model [56]. This model is most often used in surgical simulation. The behavior of real tissue can be represented faithfully by a linear model if the displacement is relatively small. We will consider this subject again in the discussion in section 5.3.

The material properties considered in this thesis are restricted to homogeneous, isotropic, linear elastic materials. Other more complex materials behavior has been described in the literature, such as plasticity (where strain does not return to zero after a certain stress amount), and viscous material (where the deformation depends on the history of the stress on the material). Also more advanced models including non-linear stress-strain and incompressible volumes can be formulated, but cannot be solved in real time with support for topological changes and the amount of detail necessary for surgical simulation. An overview of some of these material properties is presented in [56].

Combined with linear elasticity, FEM elegantly leads to a system of linear equations. The next two chapters will give a short overview of how the set of linear equations are derived from an energy measure and how the set of linear equations can be solved with standard methods.

5.1.2 Energy Function

First some notation; an organ Ω consists of nodes with the initial position $\mathbf{x}_i = [x, y, z]^T$ where $\mathbf{x}_i \in \Omega$. A node is also related to a displacement $\mathbf{u}_i(t) = [u, v, w]^T$. A node can be either free or fixed. The nodal position of a free node i at time t is $\mathbf{x}_i + \mathbf{u}_i(t)$, a fixed is always in the position x_i

The potential energy [30, 81, 99] of a system is defined as

$$\Pi = E_{strain} - W$$

Where E_{strain} is the strain energy and W is the work done by external forces. The potential energy Π reaches a minimum when the derivative $\dot{\Pi}$ is zero, this is the equilibrium that we seek. The work W is defined as:

$$W = \int_{\Omega} \mathbf{f}^T u \, dx$$

The strain energy of the linear elastic body Ω is defined as:

$$E_{strain} = \frac{1}{2} \int_{\Omega} \varepsilon^T \sigma dx$$

where ε is the stress vector and σ is the strain vector. The stress vector ε, indicating stress displacement relationship, is defined as $\varepsilon = B\mathbf{u}$ where

$$B = \begin{bmatrix} \frac{\delta}{\delta x} & 0 & 0 \\ 0 & \frac{\delta}{\delta y} & 0 \\ 0 & 0 & \frac{\delta}{\delta z} \\ \frac{\delta}{\delta y} & \frac{\delta}{\delta x} & 0 \\ \frac{\delta}{\delta z} & 0 & \frac{\delta}{\delta x} \\ 0 & \frac{\delta}{\delta z} & \frac{\delta}{\delta y} \end{bmatrix}$$

The strain vector σ is defined in relation to the stress vector ε through Hooke's law:

$$\sigma = C\varepsilon$$

That is, we have defined a linear stress/strain relationship. C is the material matrix. Assuming a homogeneous and isotropic material, the matrix is defined by the two Lamé material parameters λ and μ:

$$C = \begin{bmatrix} \lambda+2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix}$$

Often the material parameters are expressed in terms of Young's modulus E and the Poissons ratio σ connected to the Lamé parameters through:

$$\begin{aligned} \lambda &= \frac{\sigma E}{(1+\sigma)(1-2\sigma)} \\ \mu &= \frac{E}{2(1+\sigma)} \end{aligned}$$

Intuitively Young's modulus represent the stiffness of the material and Poissons ratio the compressibility. The closer σ is to 0.5 the more incompressible the material is. The energy function we use is then:

$$E(\mathbf{u}) = \frac{1}{2}\int_\Omega \mathbf{u}^T B^T C B \mathbf{u}\, dx - \int_\Omega f^T \mathbf{u}\, dx$$

5.1.3 Finite Element Solution

The Finite Element Method can be used to discretize and solve the above described energy function [30, 81, 99]. To actually find the equilibrium of energy on a computer, we will discretize the continuum into elements joined at node points, see figure 5.1. We choose an element type and an interpolation function of the nodes of the elements.

The Finite Elements most often used is the tetrahedral element with linear interpolation of the displacement fields of the four corner nodes. Through meshing, the shape Ω has been discretized into a number interconnected tetrahedrons. see figure 5.1 for an example in 2D, meshing into triangles. Inside a tetrahedron we can estimate the displacement by a weighted average of the displacement of the four nodes in the tetrahedron expressed in

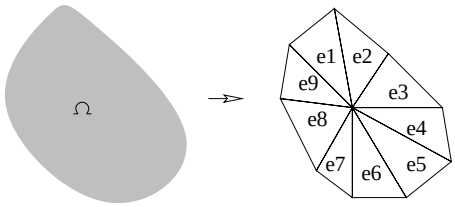


Figure 5.1: Discretization of the shape Ω into triangle elements e1 to e9.

the natural coordinate system of the tetrahedra. Through a series of calculations (see the authors master's thesis [1]or [30]) we can approximate the energy measure with:

$$\tilde{E}(\mathbf{u}) \quad = \quad \frac{1}{2}\mathbf{u}^T K\mathbf{u} - f \cdot \mathbf{u}$$

$K^e$is a so-called *stiffness matrix* and depends on tissue parameters as well as element connectivity. A static FEM formulation seeks to find the minimal energy configuration of all nodes, which amounts to solving:

$$K\mathbf{u} = f \tag{5.1}$$

This is a system of $3n$ unknown displacements, where n is the number of nodes. The matrix K is sparse since the entrances in the matrix K related to a given node are only non-zero where the nodes indicated by the row and column index have a connection to the original node. Standard systems for solving linear systems of equations can be chosen to solve this system.

5.1.4 Solving the Linear System of Equations

A range of standard methods exist to solve the system of linear equations $K\mathbf{u} = f$. First we review direct methods, meaning algorithms that seek to solve the entire set of linear equations through one application of the algorithm. Standard text-book solutions to a system of linear equation includes direct methods such as Gaussian elimination or Cholesky Factorization [109]. An alternative technique for solving the system is to explicitly invert the matrix K as proposed by Bro Nielsen in [30].

$$K\mathbf{u} = f \Leftrightarrow \mathbf{u} = K^{-1}\mathbf{f}$$

In Bro Nielsen's setup the cost was $O(n^3)$ for inversion of K, giving a long pre-computation but interactive update rates [1]. K^{-1} is a dense matrix, but a selective matrix vector multiplication with a sparse force vector can give

[1]Bro Nielsen reports interactive rates for up to 250 nodes in 1996, the typical heart model we use has 30.000 nodes

interactive rates. This method requires storage of the dense matrix K^{-1}, can introduce numerical errors in the inversion, and the pre-computation time exceeds what is normally used in Gaussian elimination or Cholesky Factorization. What saves this application is that the pre-processing is only done once, and then the matrix-vector multiplication is done for each frame. Unfortunately this solving method is not compatible with changes in topology. K^{-1} cannot easily be updated when K changes. Comparing the matrix-vector multiplication to conventional methods such as Gaussian elimination, Cholesky Factorization or Conjugate Gradient (which we shall see in the next paragraph) Bro-Nielsen reports in [30] that the matrix-vector multiplication is ten times faster than the conventional methods. This result is somewhat questionable since Bro-Nielsen does not report on actual performance numbers or on the sparsity of the force vector used in the multiplication - but the overall hierarchy of performance should be trustworthy. A method called condensation exists to make K smaller, substituting boundary conditions in K, thereby only solving for forces on the surface of the organ [30]. Again conventional methods can be used to solve this system. A condensed system would require even further pre-processing time when topology changes.

The iterative methods, in contrast to direct methods, compute a sequence of approximations, $\{\mathbf{x}^1, \mathbf{x}^2 \ldots\}$ to the solution $\mathbf{x}$, converging to the real solution of the system. Specifically large sparse systems, as is our case, are often solved through iterative methods such as the Conjugate Gradient. In [146] Conjugate Gradient is proposed as a well suited method for solving the linear equations of FEM. Since Conjugate Gradient does not rely upon pre-processing or even the explicit construction of the entire matrix K it is well suited for real-time alterations of the original topology.

5.1.5 Alternative FEM Formulations

Two different alternative uses of a FEM based solution to simulation of deformable material will be mentioned in this thesis. First, the simple formulation in equation (5.1) for a static linear system can be changed to a time-dynamic system [29] by adding mass and damping to the system (in much the same style as the Spring-Mass model we shall investigate closer in section 5.2). The equation for the time-dynamic linear FEM is:

$$M\ddot{\mathbf{u}} + D\dot{\mathbf{u}} + K\mathbf{u} = \mathbf{f}$$

Where D is a diagonal matrix with damping coefficients and M is a diagonal matrix with mass coefficients. Using a finite difference approximation this system can be solved either as a large set of linear equations (as in the static FEM) or with explicit methods comparable to the Spring-Mass model in section 5.2. For a reference, non-linear stress-strain based deformation has also been sought solved for surgical simulation, e.g.. [154], although this is naturally a further stress on any real-time constraint in an actual system.

5.1.6 Examples of Use

FEM has been used in craniofacial surgery [105] to simulate tissue response due to movement of bone and bone-parts in the face. The liver has been effectively simulated in [50]. In [142] the tissue of the arm is simulated as a three layer mode with distinct physical characteristics. Brain surgery has been simulated in [88] using FEM to represent nerves and blood vessel with high precision.

5.2 Spring-Mass Models

An often used alternative to the Finite Element Model in surgical simulation is the Spring-Mass Model. The major difference compared to the FEM is that the Spring-Mass Model is a discrete model in its basic definition. Both models are of course discrete when we actually compute the resulting deformation, but the FEM is an approximation of a continuum where the Spring-Mass model is discrete from the beginning.

As with the FEM we represent the organ Ω with a finite number of nodes. The Spring-Mass model is actually a particular type of particle system. A general particle system consists of a number of particles (or nodes) moving in space under the influence of external forces such as gravity, repelling forces, attractive forces, or collision response. Particle systems have often been used to simulate natural phenomena such as smoke or fire. The Spring-Mass model is essentially a particle system with a fixed topology connecting neighboring particles with springs that introduce repelling and attractive forces into the system to constrain the shape.

The first use of the Spring-Mass model for surgical simulation was by Cove et al. in 1993 [54] for laparoscopic gall-bladder surgery. A year later the Kismet group presented Spring-Mass based surface models [107].

The entire abdominal region has been simulated in [27], and specific simulations have been made of the liver [55] and gall-bladder [54]. Hysteroscopy has been evaluated as a case study in [136] with very positive subjective results.

5.2.1 Spring-Mass Formulation

The Spring-Mass model [81] introduces two concepts to model the elasticity of an organ: Springs and Particles. An organ Ω is defined as a number of particles $x_i \in \mathbb{R}^3$ where $x_i \in \Omega$. The particles represent mass and inertia but have no volume. The spring forces are connections between two particles that affect the particles with forces based on their distance.

In this chapter the general notion of node will be interchangeable with the notion of a particle, indicating a physical particle. The notion of an edge will be interchangeable with a spring, indicating transfer of energy.

The position of a particle in space is governed by Newtons second law of motion:

$$\mathbf{f} = m\mathbf{a} \tag{5.2}$$

where f is force, m is the mass and a is acceleration, that is the second derivative of position x.

A spring connects two particles and adds force to these particles based on their distance. Often linear springs following Hook's law are used. A Hookean spring gives a linear relationship between forces exhibited on the particles and the difference between the resting distance and the actual distance of the particles.

To simulate such forces as air resistance and loss of energy in the system, the concept of damping is introduced. Another use for the damping factor is to help ensure convergence of the numerical solutions. We assume that we have n particles that approximate the shape of the organ Ω and $i, j \in \{1, 2...n\}$. With damping the behavior of the Spring-Mass system is governed by the following equation:

$$m_i\ddot{\mathbf{x}}_i = -y_i\dot{\mathbf{x}}_i + \sum_j g_{ij} + \mathbf{f}_i \tag{5.3}$$

This second order differential equation controls the position $\mathbf{x}_i \in \mathbb{R}^3$ of a particle i with mass m_i. A velocity dependent damping is introduced to the system via the y_i constant; the faster the particle goes the more energy the system loses due to damping. Often Spring-Mass systems are damped beyond a realistic amount to increase stability of the system.

Two different categories of forces act on the particle, external and internal. External forces are forces that are external to the organ, e.g. user interaction and gravity, $\mathbf{f}_i$ represent the total external force on the particle i. Internal forces originate from within the organ, in the Spring-Mass simulation they are represented by the springs. $\mathbf{g}_{ij}$ represents the internal forces as described by the spring between i and j. In an actual implementation the spring is not present when $\mathbf{g}_{ij} = 0$. For a linear spring g_{ij} is defined as:

$$g_{ij} = k_{ij}\left(l_{ij} - ||\mathbf{x}_i - \mathbf{x}_j||\right)\frac{\mathbf{x}_i - \mathbf{x}_j}{||\mathbf{x}_i - \mathbf{x}_j||} \tag{5.4}$$

That is, $\mathbf{g}_{ij}$ is the vector between the rest and actual configuration of the spring multiplied by the spring-stiffness multiplied by the spring stiffness k_{ij} between nodes i and node j, l_{ij} is the original length of the spring between i and j. Figure 5.2 shows $\mathbf{g}_{ij}$ without the k_{ij} factor in compressed and stretched states. Intuitively, the spring adds attractive forces to the particles if they are further away than nominal distance and repulsive forces if they are closer than nominal distance.

The Spring-Mass model is said to be local because each particle can only react in response to the behavior of the particles that it is connected to

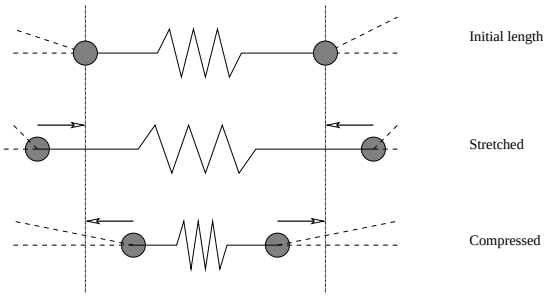


Figure 5.2: Spring response

through springs. For the dynamic system this means that forces propagate through the organ along the springs, and can only propagate one spring each discrete time-step.

The basic spring-model can be extended in a number of ways, such as additional forces to allow for a faster convergence to the original state with "home forces" [54], with explicit volume preservation [181] or by non-linear force-models [113].

5.2.2 Solving the Second Order Differential Equation

From equation 5.3 we will determine the position of the particles in the Spring-Mass system to create an animation of the behavior of the system. If we have a particle $\mathbf{x}_i$ at time t we would like to know the position of $\mathbf{x}_i$ at time $t + \Delta$ assuming we know what forces f act on the particle in that period of time. To solve the second order differential equation governing the position of the nodes in time, one often expresses the equation as two first order differential equations solved with standard methods such as Euler integration or Runga Kutta. Another method, Verlet integration on the other hand is based directly on the second order differential equation. The initial values of the position x are assumed to be given in the rest of this section.

With the introduction of a velocity variable $\mathbf{v} = \dot{\mathbf{x}}$ the equation 5.2

$$\ddot{\mathbf{x}} = \mathbf{f}/m$$

is rewritten to:

$$\begin{aligned} \dot{\mathbf{v}} &= \mathbf{f}/m \\ \dot{\mathbf{x}} &= \mathbf{v} \end{aligned}$$

The choice of integration method is a trade-off between computation time and precision of the integration.

Explicit Euler Integration

First let us write the Taylor series expansion of $\mathbf{x}(t+\Delta)$:

$$\mathbf{x}(t+\Delta) = \mathbf{x}(t) + \dot{\mathbf{x}}(t)\Delta + \frac{1}{2}\mathbf{x}^{(2)}(t)\Delta^2 + \frac{1}{6}\mathbf{x}^{(3)}(t)\Delta^3 + O(\Delta^4)$$

The most basic integration formula is the explicit Euler formulation [109]. Euler integration of the ordinary differential equation $\dot{\mathbf{x}(t)} = \mathbf{f}(t)$ is simply a Taylor-series of order 1:

$$\mathbf{x}(t+\Delta) \quad = \quad \mathbf{x}(t) + \Delta \cdot \mathbf{f}(t)$$

With respect to the second order differential equation 5.2 the solution is:

$$\begin{aligned} \mathbf{x}(t+\Delta) &= \mathbf{x}(t) + \Delta \cdot \mathbf{v}(t) \\ \mathbf{v}(t+\Delta) &= \mathbf{v}(t) + \Delta \cdot (\mathbf{f}/m) \end{aligned}$$

Explicit Euler integration is very simple to compute, but is inherently unstable.

Runga Kutta Integration

Runga Kutta is mentioned here as a more precise and stable, but also slower, alternative to Euler integration. Runga Kutta 4 is an often used [20] variant of Runga Kutta which reproduces the terms of the Taylor series up to a term involving Δ^4. The error is of size $O(\Delta^5)$. Another variant, Runga Kutta 2, has been used in e.g. [181]. One advantage of the Runga Kutta family of integration is that they support a change of step-size to increase accuracy of the integration on parts of the function. This feature is not immediately useful in a real-time surgical simulator because we will need a stable flow of frames at all times.

Verlet Integration

Verlet integration was originally introduced in the field of molecular dynamics and is more rarely used in surgical simulation than the standard methods of Runga Kutta and Euler integration. Verlet integration was used in [101] for advanced character physics[2] and [137] for surgical simulation. The Verlet integration is based on two third-order Taylor expansions of the positions $\mathbf{x}(t)$, one backward and one forward:

[2] Thomas Jakobsen is very often cited in various fields for popularizing Verlet integration - which in his case was used for a very popular computer game.

$$\begin{aligned}\mathbf{x}(t+\Delta) &= \mathbf{x}(t) + \dot{\mathbf{x}}(t)\Delta + \frac{1}{2}\mathbf{x}^{(2)}(t)\Delta^2 + \frac{1}{6}\mathbf{x}^{(3)}(t)\Delta^3 + O(\Delta^4)\\ \mathbf{x}(t-\Delta) &= \mathbf{x}(t) - \dot{\mathbf{x}}(t)\Delta + \frac{1}{2}\mathbf{x}^{(2)}(t)\Delta^2 - \frac{1}{6}\mathbf{x}^{(3)}(t)\Delta^3 + O(\Delta^4)\end{aligned}$$

Adding the equations and isolating for $x(t+h)$ gives us:

$$\mathbf{x}(t+\Delta) = 2\mathbf{x}(t) - \mathbf{x}(t-\Delta) + \ddot{\mathbf{x}}(t)\Delta^2 + O(\Delta^4) \tag{5.5}$$

Since we seek to integrate Newton's equations, $\ddot{\mathbf{x}}$ is known directly as $\frac{\mathbf{f}}{m}$ from equation 5.2. The Verlet method is reasonably fast to evaluate and is very stable.

The damping of the Spring-Mass system was introduced as a linear function of the velocity. In the standard Verlet method the velocity is not expressed directly, and we will therefore use an ad-hoc method of weighting the old and new positions to create a dampening effect:

$$\mathbf{x}(t+\Delta) = \mathbf{x}(t-\Delta)(1-\lambda) + \mathbf{x}(t)\lambda \tag{5.6}$$

5.2.3 Stability

The integration methods used to solve the differential equations of the Spring-Mass system have a trade-off between numerical accuracy and speed of calculation. We need a real-time solution, and the numerical accuracy is therefore of lesser importance. Most important is that the stability of the solution.

In [137] Euler, Runga Kutta 4 and different Verlet[3] methods were compared with respect to their ability to deliver stable real-time results. It was tested how large the time step of the individual methods could be set while remaining stable. Taking into account the calculation time of the integration, the Verlet method is superior to Euler and Runga Kutta 4. The Verlet method is therefore chosen as a standard in this thesis, although other methods could be supported.

Because stability is most important, some measures can be taken to increase it. One method is to dampen the system beyond a realistic level. Some kind of iterative constraining of distances is another possibility. In [158] the phenomenon of elongated springs was identified in simulation of cloth. Elongated springs are identified as a system in which some springs are stretched unrealistically in relation to others. We would like the springs to be of approximately equal length. One solution is to use a technique known as relaxation, which iteratively constraints all lengths [101, 158]. In section

[3]Basic Verlet is not checked, but is equivalent to velocity Verlet when the velocity is not needed

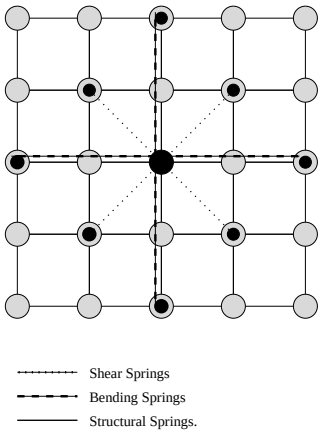


Figure 5.3: 2D regular grid with additional springs

8.7.1 we use a specific constraint determined by the volume of the elements in the simulation.

5.2.4 Spring Topology Issues

The springs represent constraints and flow of energy in the Spring-Mass system, and the topology of these connections as well as the relative position of particles determine the global behavior of the simulated organ. The logic behind the connection of particles also determines the possible visualizations and interaction.

An under-constrained system might have several resting positions and the system might easily end up in configurations of nodal positions which is allowed by the system, but is not realistic. Parts of an under-constrained system can collapse because part of the volume might flip into itself. When a grid structure is used as basis for spring connections the system is under-constrained because the boxes or cubes are in themselves under-constrained. In a 2D grid the missing forces have been identified as missing resistance to shear. The solution is to connect springs across the diagonal [158], see figure 5.3. In a 3D grid composed of boxes additional springs connects corners of the box to resist a collapse [181].

If the system is over-constrained it will exhibit less elasticity and more rigid behavior than we indicated through the spring stiffness. Another problem with an over-constrained system is numerical stability (these kind of stability problems are closely related to the problem of stability and spring-stiffness, see section 5.2.3). Because of these difficulties, systems have often been arranged in regular structures guaranteeing a homogeneous behavior all over the organ. Often regular lattices or prisms with triangular base have been proposed.

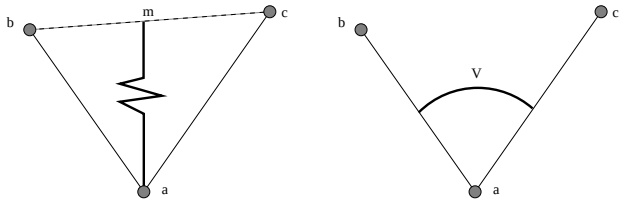


Figure 5.4: Spring to maintain curvature

The strategy behind the connection of nodes through springs is important for the visualization and interaction with the organ. The strategies for connecting the springs are often divided into two categories; surface and volume representations. The surface representation simply defines only a surface with no explicit volumetric elements, the surface model may nonetheless be used to approximate volumetric behavior. The volume representation has elements to ensure a behavior that takes the volume of the model into account. Often the particles of a volumetric representation are arranged in a set of connected tetrahedrons, hexahedrons, or other volumetric geometries.

The choice of surface, volumetric, or other hybrids, can be regarded as a hierarchy of approximations in which true volumetric models are more precise than surface models. A surface model representing the surface of some volume will be faster to compute than a full simulation of volume. The trade-off again is efficiency of computation versus physical accuracy [56].

Because the topology of springs is so important for the behavior of the system and the speed of computation, the characteristics of the simulated organ can be taken into consideration when planning the strategy for connectivity. In [55] the behavior of a liver was simulated by two different geometrical components; A 2D elastic surface to simulate the membrane of the liver (with torsion springs to simulate curvature of the surface) and a 3D mesh to simulate the interior of the liver (as a quasi-viscous material).

Surface Models

While the surgical simulator presented in the third part of this thesis is based on a volumetric representation of soft tissue, it is valuable to investigate the initial experiments in the field of surgical simulation, using simpler representation of morphology. A simple *surface model* is typically represented by a two-dimensional grid with springs to resist shear and bending [158], see figure 5.3. Delingette et al. [56] used a surface model to approximate a thin vessel while Cover et al. [54] used the surface model to approximate the volumetric behavior of a gall-bladder. When a basic surface model is used to approximate the behavior of a volumetric organ, there are of course no explicit forces that react to changes in volume. This can easily lead to

unrealistic results where the model self-intersects. To get *some* volumetric behavior additional springs are sometimes introduced into the surface based models. The basic approach is to introduce springs to guarantee some curvature of the organ. If the surface consists of a simple 2d grid, bending springs can be introduced to resist bending of the surface, see figure 5.3. In [143] the animation of a muscle along an action line is simulated. The muscle geometry is organized in a grid along the surface of the muscle. [143] therefore introduces angular springs (see figure 5.4 (a)) to preserve the volume and the overall shape of the muscle. In [55] torsion springs (see figure 5.4 (b)) are added to maintain the curvature of the surface. All these different springs reduce to a computation over two of three springs arranged to maintain the curvature of the surface.

Volumetric Models

A volumetric model of some tissue often represents the entire organ with smaller atomic parts of tetrahedra or hexahedral elements. The tetrahedral mesh gives a great flexibility for an efficient representation of any morphology [20], but constructing tetrahedral meshes is an active research topic in itself [148]. In the Spring-Mass model, edges of elements are often translated directly to springs. In that case the tetrahedral elements are structurally stable, but hexahedral elements are under-constrained and additional springs are often inserted across the elements [181]. The simple volumetric model can be extended to encompass more elaborate biological features. In [105, 181] a three layer model of the skin tissue is used because the three layers have different physical characteristics.

Connected Surfaces

In the cross between surface and volume models I will present a specific pseudo-volumetric model, the so-called *connected surfaces.* This very simple idea was presented in the authors masters thesis [1] and was later generalized in [62]. The motivation for showing the result here is to emphasize that the difference between surface and volume is not clear-cut in the case of Spring-Mass systems. The model was developed with the heart model in mind and requires no special meshing - it behaves as if it was a true volume representation. It is an alternative to the true volumetric tetrahedron or hexahedral mesh of the heart. From the segmentation we can retrieve surfaces representing the outer and inner surfaces of the heart muscle. Data from the segmentation is given as a surface consisting of triangles. The idea of Connected Surfaces is to use both the inner and outer surfaces, and set up a relationship between them so that a volumetric behavior is exhibited - the relationship is defined as additional forces. These additional forces, called connecting forces, would propagate force between the two layers to

resist bending and constraining the distance between the inner and outer surface. The forces are represented as additional springs connected from each particle to particles in the surface on the opposite side.

Best Topological Scheme

If computational power was not a problem, we might ask ourselves what the best topology scheme would be. There is no simple answer to this question. Surely, volumetric behavior is more realistic than surface behavior because a simple surface model cannot preserve volume to the same degree as a volume model. But the Spring-Mass system might not exhibit a more realistic behavior even when the resolution and connectivity of springs resemble some model of reality more closely. If the resolution or detail of the model is increased the behavior cannot be guaranteed to be the same as in the low resolution model. This basically has to do with the fact that a Spring-Mass system is defined locally; there is no global differential equation that we solve for a minimal energy configuration, this is implicit in the Spring-Mass system.

When the resolution is increased the propagation of forces is slower because forces only propagate along one spring each time step. In a higher resolution model the mass also has to be divided in some way, but because of differences in the connectivity this might not be simple.

The point is that we must validate the models we build, either by comparing them to previous models, formally validating them against real data, or getting expert opinions [71].

5.3 Comparing FEM and Spring-Mass

In this section we compare some of the properties of Spring-Mass and FEM as well as the methods used to solve the involved equations.

In [51] Spring-Mass and a tensor-mass model is compared. The tensor-mass is derived from time dynamic FEM formulation but is solved through explicit integration. Since the tensor-mass is based on the finite element approximation in computation of stiffness, it is independent of any geometric properties. The Spring-Mass system model, on the other hand, is defined from the beginning as a discrete model. The behavior of the Spring-Mass system depends on the connectivity of springs and nodes. In [51] they furthermore argue that since the tensor-mass is derived from continuum mechanics, material parameters can be directly applied from measured parameters. For the Spring-Mass model there is a basic problem of finding parameters, since the topology determines part of the behavior. In the authors early work, one such methods was introduced [3] searching for optimal parameters for a Spring-Mass model in comparison to a FEM model with

the additional argument that the parameters should be optimal when the two models behave alike over time.

In [154] and the comparison in [51] a serious problem with the linear FEM based models is presented, namely that FEM is only valid for small deformations. One issue is naturally that the linear approximations used in FEM have their natural limits; FEM is often cited to be valid only for deformations in the range of 10% [57]. More severe though is the fact that FEM is not invariant to rigid transformations. As an example, when a rigid rotation is applied to the set of nodes, the elastic energy increases - essentially blowing up the shape [154]. The same problem occurs if only part of a model is subject to a relatively large rotation. Spring-Mass models do not have this basic problem, and are invariant to rotation. In [105] craniofacial surgery was simulated with Spring-Mass and FEM and there was some indication that they behave identically for these small deformations.

We now compare properties of the solutions methods; explicit integration (from section 5.2.2), iterative methods, and direct methods of solving the set of linear equations (from section 5.1.4). A specific property of the explicit integration is that it takes a number of iterations proportional to the number of edges (or springs) for forces to travel along the segment described by these edges or springs. That is, forces travel rather slowly through the volume, and the nature of movement of forces through the mesh affect the deformation. A direct solution or iterative solution to the globally defined system of linear equations will (given that the requirements of the method are fulfilled) solve the globally defined measure of energy. The local information available in an explicit numerical integration of FEM or Spring-Mass is furthermore an issue since a local equilibrium of energy does not necessarily correspond to a global minimum. Specifically in the case of Spring-Mass models with no explicit notion of volume this can be problematic since the basic elements constituting the total organ-volume can invert and be kept in that position by the external connections going out from this basic element, i.e. the element is in a local minimum of energy. The FEM model has a basic notion of volume and therefore introduces forces to "flip" the element back in the case of explicit integration of the time dynamic formulation. There is no guarantee though that these volume-governing forces are large enough to escape the local minimum of energy. Another important aspect is naturally the computation time, already dealt with specifically for FEM in section 5.1.4. As mentioned, a direct solution takes a considerable amount of time to solve. An iterative solver or an explicit integration scheme has the possibility of receiving updated interaction, visualize, or otherwise do computation at each intermediate step - of which each naturally demands less computation than a complete direct solution. Following the discussion of support for topological changes started in section 5.1.4, it should be clear that the Spring-Mass model solved with the explicit integration scheme is very well suited to topological changes since no pre-computation is neces-

sary, in contrast to methods of inverting the stiffness matrix or using direct methods.

Many papers present FEM as the accurate realistic method while Spring-Mass is the fast but approximate method - although there is some evidence suggesting this, we must ask ourselves how this theoretical comparison transfers to real data of human tissue. Specifically in the case of cardiac malformations, the tissue-parameters can vary quite much, and are not known prior to surgery. In that case Spring-Mass might be just as "precise" as FEM. More investigation is necessary in this area, to quantify precisely how different tissue-models relate to what tasks can be trusted in simulation, compared to the real procedure.

5.4 Real-Time and Complex Morphology

Based on the previous discussion it should be evident that speed of computation and the inherent constraint in numerical methods used are important aspects of surgical simulation. A general tendency in the research contributions (as presented in previous sections of this chapter) has been to develop faster algorithms for the computation of tissue-deformation. Primarily these attempts have not been about creating entirely new methods to calculate the deformation, but about accelerating existing models in computer animation and mechanical engineering through various assumptions and constraints. The tendency of simulators to increase in complexity and generality, with the end-goal of open surgery, as presented in chapter 4, is in direct contradiction to some of the technical contributions that constrain the amount of flexibility of the available methods. E.g. some of the pre-processing techniques of FEM (section 5.1.4) do not support cutting. This constraint can be eliminated somewhat if we can define a region-of-interest [51, 88] on the organ, within which a flexible tissue-model can be used. Other methods such as [31] assume a point based interaction to speed up convergence of a Spring-Mass model.

In section 2.4 we argued that a simulation is very much a specific representation of reality within very tight boundaries. Developing methods of surgical simulation should always be done with a clear idea of what type of surgical procedure to simulate, and in cooperation with surgeons. We now turn to the specific issue of simulating a heart with congenital defects. From section 2.3 explaining heart defects, it should be evident that we require a large degree of detail to accurately model congenital heart defects than previous surgical simulators (see chapter 4)

The author's early research on a CPU-based Spring-Mass model for faster point based interaction was presented as the LR Spring-Mass model [2], which is a concrete example of an alternative Spring-Mass model meant to react faster to point-based interaction. The LR Spring-Mass model is based

on *local interaction* and *iterative relaxation*. Local interaction is based on an assumption that there is a distinguishable region-of-interest in which the surgeon is manipulating tissue. In this region an iterative relaxation of constraints in the Spring-Mass system is executed from the point of interaction and out. Such as system is clearly less flexible than a general implementation of the Spring-Mass system. The research strategy in this PhD project has been to use the existing method of Spring-Mass as it is without constraining or assuming anything, i.e. this is generally applicable to any organ or interaction modality. We allow the surgeons to do anything anywhere on the heart. The method to achieve this acceleration of computation of tissue-deformation for more accurate simulation of complex morphology, is by using more efficient computational architectures readily available to us. As we shall see the modern graphics processing unit turns out to be such a platform.

Part III

The GPU Accelerated Surgical Simulation

Chapter 6

General Purpose Computation on the GPU

> "And now for something completely different" (John Cleese, 1971)

To understand and motivate the use of graphics hardware to accelerate the computations involved in our simulator, I present programmable graphics hardware and the general use of such in this chapter. In the next chapter we will again take up the subject of surgical simulation in combination with the programmable graphics hardware.

The graphics processing unit (GPU) [193] today is a specialized chip found in modern personal computers and game-consoles. The role of the GPU is to accelerate drawing of 2D and 3D graphics by implementing the required functionality in dedicated hardware. The modern GPU is a further development of the blitter-chips of the 70's and 80's supporting fast drawing of 2D sprites. The Commodore Amiga was the first consumer-level computer with dedicated graphics chip with blitter support. In 1996 a company named 3Dfx released the Voodoo Graphics chip, which was one of the first and most famous 3D graphics acceleration cards. In 1999 the Nvidia GeForce 256 [192] was the first graphics card released with integrated hardware support for transformation and lighting. Nvidia coined the term GPU (Graphics Processing Unit) to describe the feature-set of the GeForce 256. In 2001 the next generation, GeForce 3, was released by Nvidia. This GPU was the first consumer-level card to include programmable vertex and pixel shader. *General purpose computation on the GPU* [89] or *GPGPU* in short refers to the discipline of using the programmable GPU for general problems not necessarily dealing with 3D graphics or not directly suited for the normal rendering pipeline. This chapter will deal with the programmability of graphics hardware and how in general we can use the computational power for something else than graphics.

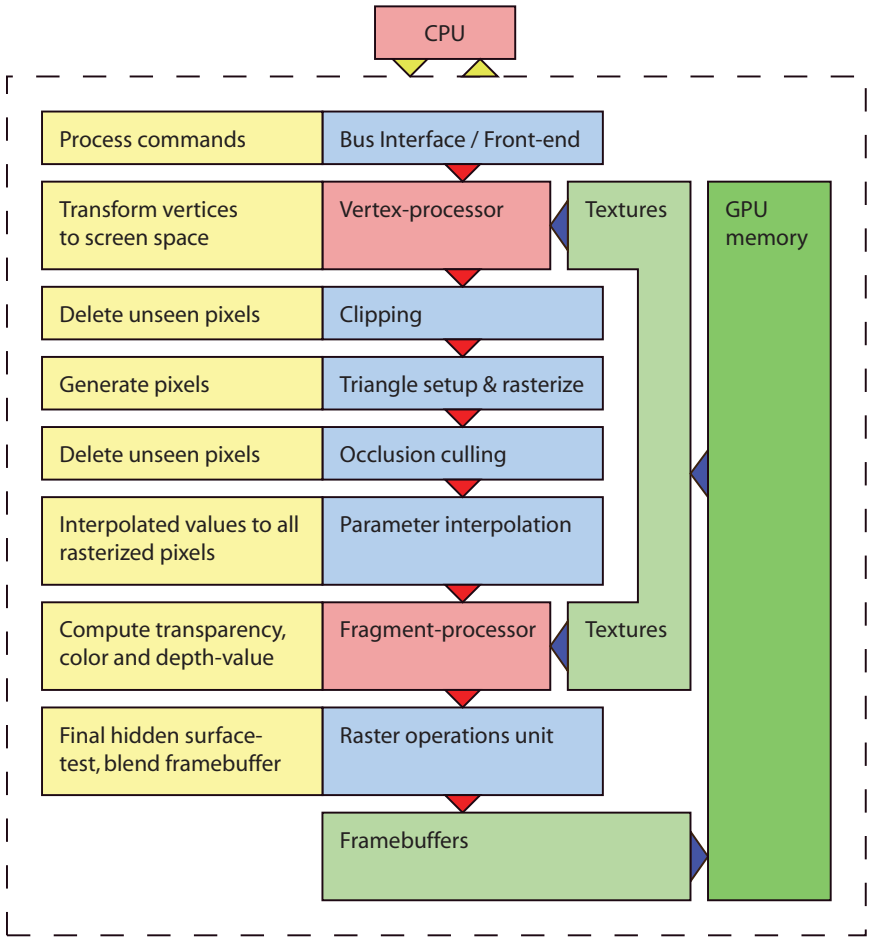


Figure 6.1: A simplified drawing of the graphics pipeline. The red boxes represent programmable parts of the pipeline; The CPU application, GPU vertex processor and GPU fragment processor.

6.1 Graphics Pipeline

Within the context of this thesis, the graphics pipeline [65, 153] refers to the pipeline of elements responsible for a rasterization-based rendering of 3D geometry. Of special interest are the parts of the pipeline implemented in hardware on the current generation GPU's. The graphics pipeline is basically responsible for the generation of a pixelated 2D image based on a 3D scene, including geometry, texture, shaders, light etc. To use the GPU efficiently it is very important to be aware of the graphics pipeline, possible performance bottlenecks and optimizations (see [65, chapter 28]).

In figure 6.1 a simple illustration of the graphics pipeline can be seen. The following is a short description of the normal use of the graphics pipeline. This description also covers what is known as the fixed function pipeline.

That is, the default behavior of the graphics pipeline if the programmability is not used or not available. The CPU has transferred textures and shader-programs to the GPU in a step not shown on the illustration. At the beginning of the shown pipeline, the CPU based application has sent geometry in the form of vertices, arranged in triangular faces, to the GPU. Vertex processing converts the vertices from coordinates in a three-dimensional world space to screen-space through transformation with the Model-View and Projection matrices. Through the triangle setup and rasterization it is determined which fragments should be generated for later fragment-processing, i.e. one for each pixel on the screen covered by a triangle in screen-space, possibly overlapping previously computed fragments. Next is occlusion culling, where fragments can be excluded from fragment processing based on e.g. the stencil buffer and depth buffer. Vertex attributes are now interpolated across the surface of the triangle to create per fragment input-values. Finally the fragment shading determines the final pixel color based on texture-lookups, some mathematical calculations, and the interpolated per vertex attributes. In the fixed function pipeline the actual calculations depend on certain OpenGL states. In the last step of the pipeline the color values can be blended with the current framebuffer.

In the next section we will look into the fact that two parts of this pipeline have recently become programmable, namely the fragment processing and the vertex processing.

6.2 Vertex and Fragment Programs

The graphics pipeline as described in the previous section has traditionally been a fixed function pipeline, in the sense that each step of the pipeline executes a specific functionality to which parameters can be given. Specific OpenGL extensions[1] have been created to allow for other functionality than the default fixed function pipeline. As examples of alternative fragment processing released as extensions are texture environments for more flexible combination of textures and incoming fragments, per-fragment lighting, and dot product based normal mapping. According to the OpenGL extensions for fragment [117] and vertex processing [33] this inflexibility of fragment and vertex processing was in contrast to the actual underlying floating point engines on the graphics cards. Any requests by the community of 3D graphics developers for specific features of fragment or vertex processing had to go through laborious routines to be exposed in the drivers as extensions. By exposing this programmability to the public, the developers of 3D graphics applications are now able to customize vertex and fragment processing to a degree not previously possible.

[1] Vendor specific extensions or ARB extensions to a core OpenGL functionality

In OpenGL the functionality of vertex and fragment programs (or shaders)[2], has been exposed through OpenGL extensions corresponding to the functionality of each new generation of GPU hardware. Of specific interest in this thesis is the fragment programmability and some of the newest functionality in the vertex shader. The fragment or vertex program is uploaded from the CPU to the GPU, delivered to OpenGL in a textual format. The uploaded program is then executed for all consecutive processing of vertices and fragments. This execution is conceptually in parallel for all vertices and for all fragments (i.e. the program is run once for each vertex or fragment) but in reality the program is run in parallel in smaller batches of vertices and fragments. Two of the most recent cards report 48 pixels pipelines (ATI Radeon X1900XT) and 24 pixel pipelines (Nvidia GeForce 7900GTX) respectively.

In the remaining section we discuss specific instances of vertex and fragment programs and the improvements of each new generation. In the year 2000, vertex processing was exposed through the `NV_vertex_program` [108] extension in OpenGL[3]. In 2001 the `NV_fragment_program` [36] extension allowed programmers to access fragment programs in OpenGL. These extensions were developed by Nvidia and were consequently specific to the Nvidia line of graphics hardware. The OpenGL Architecture Review Board (ARB) governs the official OpenGL specification as well as ARB approved extensions. In 2002 the ARB released the `ARB_fragment_program` [117] and `ARB_vertex_program` [33] extensions. The ARB specification for fragment and vertex programmability has been implemented by all major graphics card producers, including Nvidia and ATI. This first generation of programmability in fragment programs allowed for general calculations and texture lookups in fragment programs and general calculations in vertex programs. Initially, both vertex and fragment processors worked in a single instruction multiple data (SIMD) perspective, where the same instructions are executed for all elements, allowing for optimizations of the computational hardware. Through Nvidia-specific extensions the vertex shader was rather quickly allowed to do real branching [35].

The instructions of both vertex and fragment processor operate on four-tuples of data, i.e. (x, y, z, w) in space or (r, g, b, a) as a color. The vertex processor can do computation based on incoming per-vertex attributes (typically a maximum of 16 four-tuples) and per-program constants, resulting in a vertex position, colors, texture coordinates (typically a maximum of 8

[2]The terms pixel and vertex shader comes from the launch of DirectX 8. In OpenGL the same functionality has also been called vertex programs, fragment shader and fragment program - hence the naming is now somewhat ambiguous.

[3]Although register combiners had been exposed earlier and not mentioned specifically in this thesis, the Nvidia line of fragment and vertex programs are mentioned here since later extensions in the form of options were added by Nvidia to the standard `ARB_fragment_program`.

four-tuples), and a few other results. The fragment processor can do computation based on the interpolated incoming vertex attributes in the form of colors and texture coordinates, resulting in a color[4] and possibly a depth value. Each vertex can only write to its "own" vertex attributes and each fragment can only write to its fixed location in the framebuffer. The initial programming of the graphics-cards was rather difficult since one had to work under a couple of severe constraints. The instruction count[5] was limited, the number of texture lookups was limited[6], the number of registers available was limited. Through the evolution of graphics hardware and drivers these limitations are much less obtrusive today than in the beginning of fragment and vertex programmability. Our first attempt at the computation of a Spring-Mass model on the GPU was on an ATI 9800 PRO in the early summer of 2003. Due to the constraints of that time, four passes of fragment programs were necessary for our computations, compared to our current one-pass program which easily fits within the resource limits of today. Other limitations are inherent in the programming model of this generation of cards though;

- No stack or heap - only a constant number of registers
- No integer or bitwise operations
- No branching - instead one must use conditional write of data, i.e. there is no "skipping" computation

In 2004 the Shader Model 3.0[7] was announced. This corresponded with the release of the GeForce 6800 series of GPUs from Nvidia which implemented this functionality as the first graphics hardware producer. Apart from more resources for the fragment and vertex programs SM3.0 included new functionality. Of importance to this thesis, the fragment processor got instructions for doing native true branching [37] and the vertex processor got instructions for doing texture lookups [34]. This functionality was exposed in OpenGL through extensions to the fragment and vertex programs.

The SM4.0 standard and Direct3D 10 has been released [22] and will in large determine the functionality of upcoming graphics hardware. Amongst the features are a stronger requirement of implemented features to avoid graphics-card specific code paths, a geometry shader, and a "common core" virtual machine as the base for the three programmable stages.

[4] One can write to several framebuffers through an extensions known as Multi Render Target though.

[5] Actually the ALU instruction count

[6] and divided into dependent and independent based on whether the address used was based on previous calculations in the fragment program or not

[7] 3.0 since this was the version in Direct-X

6.3 High-Level GPU Programming

The native fragment and vertex programs of GPUs are low level programming mixed with some high-level features. Working under the initial very tight resources it made good sense to work directly in the low-level fragment and vertex programs - rather quickly though high-level languages appeared. let us first look shortly at shader languages in general. Robert Cook is traditionally attributed with laying the grounds for shader languages through his work with shader trees in 1984 [49]. In Cook's shader tree, computation is arranged in a tree of operations, evaluated postorder to result in the final color in the root of the tree. In 1988 the *RenderMan Interface Specifications* was release[156]. The RenderMan language by Pixar was the first successful language, or protocol, describing the interface from modeling to rendering. Of interest to us is the RenderMan shading language. This shading language was targeted for high image quality and defined five different shaders; light, displacement, surface, volume and imager. Like its later successors the language is a C-like syntax.

The Stanford Real-Time Shading Language [157] was the first attempt at a high-level shading language built for the pipeline of modern graphics processors. The motivation was to create a simple abstraction of graphics hardware in the year 1999 with internal support for multi-pass rendering and the initial difficult programmability of GPU's. A cooperation between Microsoft and Nvidia resulted in the two very similar languages *HLSL* (High Level Shading Language) for Direct3D by Microsoft and *Cg* (C for graphics) by Nvidia in 2002. Cg is a compiler and runtime system for a range of platforms and target languages, supporting all the different fragment and vertex languages of OpenGL as well as Direct-X. Most important to this thesis is the fact that Cg delivers a high-level programming language[8]. Cg has been used exclusively as a high-level GPU language in this thesis in conjunction with low-level native fragment and vertex programs. OpenGL Shading Language[9] [164] is an ARB extension to OpenGL[10] that seeks to create a high-level shading language for OpenGL specifically.

6.4 GPGPU and Performance Issues

In this section I will discuss, in general, the resources available on the GPU for general purpose computation. In chapter 7 I will discuss this in more detail with respect to the implementation of surgical simulation. From the previous sections it is evident that the GPU can be programmed in a general purpose way, one element is still missing though; main memory access. The

[8] and a compiler which in the later versions outperforms the earlier hand-optimization of native fragment program code by the author

[9] Also called GLSL or GLslang

[10] and part of the core OpenGL 2.0

programming model of fragments and shader defines no permanent memory so this has traditionally been dealt with in other ways. Textures are the natural choice since fragment programs read from textures. Two additional issues are of importance; the format of the textures and how to write to textures (or render-to-texture). In the initial papers on GPGPU such as [91] only 8-bit textures were available and writing to textures was realized through a copy-operation of the active framebuffer to a texture, from which the data could subsequently be read. The basic method of reading from and writing to textures is still used, but the initial use had two issues; precision of data and a potential bottleneck due to the copy-operation.

8-bit precision is simply not sufficient for scientific calculations. Later extensions to the GPU has allowed floating point precision textures, first through some vendor specific extensions [122, 32] (with various limitations), and later through an ARB approved extension [110]. One thing to notice is that the support for texture-filtering (e.g. bilinear filtering) is potentially not hardware accelerated[11].

To alleviate the bottleneck of a copy-operation on the GPU in conjunction with render-to-texture functionality a number of extensions were introduced in 2000/2001; *pixel buffers* and *render textures*. A pixel buffer is an off-screen framebuffer that can be rendered to, and the render texture allows a color-buffer to be used for both rendering and as a texture. This method of implementing render-to-texture is often just called PBuffer, and will be named so in the rest of this thesis. The problem with PBuffers is that each PBuffer has an OpenGL render context which can take up resources. Changing OpenGL context is also a potential bottleneck. In 2004 a more general handling of framebuffers and textures was proposed in the framebuffer-object (FBO) extension [102] by the ARB. The FBO extension allowed for a much easier implementation of render-to-texture and theoretically for a faster performance, although this has not been the case yet for our use of the GPU.

An important issue in programming of the GPU is that even though we have high-level programmability (as demonstrated in the previous chapter) there is a clear distinction between register-based calculations and reading and writing to "main-memory" on the GPU. Like the CPU, registers are naturally the fastest level in a hierarchy of storage. Consequently the developer of GPU applications is forced to be aware of the memory-hierarchy, unlike programming the CPU in a high-level language where register allocation, including loading and saving to memory, is usually handled by the compiler. High level languages for the GPU as presented in the previous section do a "simple" register allocation for variables in the programming language, but if there are too few registers to express the program the compilation will fail.

[11] at the time of writing the GeForce 6800 series and GeForce 7900 series support bilinear filtering in 16 or 8 bit only

One reason that it might be a good idea to be aware of these issues is naturally the general tendency of memory latency to increase as well as memory bandwidth to decrease relative to speed of calculation [58]. An alternative way of working with computation and communication is formalized in the *stream processing paradigm*, where a stream of data elements (a texture of texels) is manipulated by a kernel of computation (fragment program), that is executed for each element and resulting in a new stream for further processing. Kernels can thereby work on elements as in a pipeline using local chip memory for input/output stream elements to minimize external memory bandwidth (read and write to textures in our case). The computational strategy is to have the memory latency hidden through parallelism, interleaving memory access and computation. This strategy is utilized by the modern GPU making it a cost-efficient parallel processor that has a significantly higher floating-point throughput than a CPU from the same generation of chips. This performance-gap widens with every generation [43]. The fact that the GPGPU processing model fits very nicely into the stream programming paradigm has been recognized in [40]. In most GPGPU research the fragment processor has been used for doing the heavy part of the calculation since there are basically more fragment pipelines than vertex pipelines. The vertex processor can read texture-memory, but this is still a major bottleneck.

The layout of basic computation for a GPGPU program is the following; rendering of a framebuffer filling quad initiates computation on all fragments. The vertices of the quad are given texture-coordinates that map one-to-one to the texture used for data. Results of the computation are written through render-to-texture to another texture and can be used for input in subsequent computations. The texture read from, and the texture written to, are in most cases two different textures since problems of a deterministic read and write would otherwise occur. Let us now look at the addressing available from the vertex processor. Rendering a screen filling quad would interpolate the per-vertex attributes across the entire quad, i.e. a memory addressing based on these per-vertex attributes must fit that interpolation. This gives a minimum of flexibility with respect to memory-addressing, enforcing a specific layout of data, but it is very fast since only four vertices must be processed and addresses are determined by the rasterizer. In the opposite end of a scale of flexibility in memory addressing is the most general case where each individual fragment needs a unique memory-address in the form of texture coordinates not expressed efficiently though linear interpolation. In such a case we could render a point-sized (fragment-sized) geometric-primitive with the unique texture coordinates. In such a use, the vertex processor could very easily become a bottleneck, but this allows for the most flexible addressing from vertex to fragment. Often we must find a middle-way between these two extremes. Another level of addressing to consider is using the values in texture as texture-coordinates to

implement pointers.

6.5 Applications of GPGPU

A number of different applications have been sought implemented on the GPU in the hope of better performance. In chapter 7 we will specifically talk about techniques of use for surgical simulation, including linear algebra[12]. In this section we will therefore look shortly at other selected applications of GPGPU.

In this section I will in short present two different methods of ray-tracing on the GPU. This is of specific interest in GPGPU since ray-tracing in itself is a rendering method, but is not easily expressed in the programmable GPU. Ray-tracing is basically a rendering method in which rays are traced from the eye, bouncing off material or refracting through material, to a light source. Important to our discussion is that bouncing rays around the scene is based on finding intersections with rays and triangles. Two different strategies have been presented; [44] using the GPU for ray-triangle intersections in a CPU based ray-tracer, and [159] implementing[13]. the entire ray-tracer on the GPU.

The strategy in Carr et al. [44] is to use both the CPU and GPU, dividing computation based on what the CPU and GPU does best. This should enable both CPU and GPU to be kept busy with computation, utilizing the combined computational power. In [44] it is recognized that the GPU could effectively do a sequence of ray-triangle intersection tests. Since the CPU is good at anything involving complex branching, sorting, and advanced data-structures, the CPU will setup the ray-triangle intersections for the GPU. One pass of the GPU-algorithm reports intersections between all rays and one triangle. That is, each fragment is responsible for computing intersection between one ray and the current triangle. The information on triangle location is given as per-vertex attributes. For each triangle associated to the current batch of rays a rendering-pass of the algorithm is executed. Each pass checks if any potential intersection with the current triangle is closer than anything else so far. The only thing kept in memory on the GPU is ray origin, ray direction and closest intersection. This all maps directly to the fragment in question and is consequently just saved in several PBuffers of the same size.

The strategy in Purcell et al. [159] is to implement the entire ray-tracer on the GPU. To do this, data structures for triangles, materials and rays must be kept on the GPU. Furthermore to be practically usable, an acceleration structure for ray-triangle intersection test is used. In [44] this accelera-

[12] One reason for this division is that chapter 7 is based on a published paper.

[13] Although it could not actually be implemented at the publication time of the article, the target platform was released a few months later

tion structure would be kept on the CPU. The acceleration structure used is a simple grid-based partitioning of triangles. That is, the GPU-memory will be used for grid cells and references to triangles which again reference material parameters. As in [44] each fragment is responsible for one ray, but now each fragment can be in several states; ray traversing the grid, ray triangle intersection, final shading and finished. At the time of publication Purcell et al. used one pass per state since only conditional write was available. Today the true branching of SM3.0 could be used as well.

The two papers clearly take different approaches to accelerating ray-tracing through a GPU implementation. Carr et al. suffers from potentially slow read-back of intersection results for each batch of tested rays and triangle. There is more flexibility in choosing an acceleration structure though, and assuming a good ray-coherence for each pass the missing acceleration structure for each pass of rays would not be a problem. The large amount of synchronization between GPU and CPU in Carr's ray-tracer is also a potential problem, whereas Purcell et al. can produce the final image from geometry on the GPU. Carr on the other hand utilizes both GPU and CPU, where Purcell et al. can potentially suffer under the lack of efficient flow control on the GPU and an idle CPU. In a SM2.0 implementation many fragments might be idle since they are in different state, and in a SM3.0 the true branching might not be as efficient since close fragments have some probability of being in different states. With this discussion of implementation we should have some idea of the choice we have to make in GPGPU.

Chapter 7

Surgical Simulation on Graphics Cards

In this chapter we return to the subject of surgical simulation, but this time with the knowledge of the programmable graphics cards. This chapter is based on the review paper [10] and is an introduction to surgical simulation implemented on graphics hardware. This chapter has been changed slightly compared to the original article with some additional cross-references and a more detailed comparison of Spring-Mass implementations. For this chapter to stand alone and still resemble the original contribution, some overlap in topics with the subsequent chapters must be anticipated.

7.1 Introduction

General-purpose computation using graphics hardware (or GPGPU) is a research area that has grown rapidly in recent years. By using the modern graphics card (i.e. the GPU) for computations, many computationally heavy algorithms have been accelerated significantly compared to conventional CPU-based algorithms. This includes most of the techniques currently being applied in surgical simulators. Unfortunately, the GPU is difficult to utilize efficiently. A substantial knowledge of its design, programming model, and limitations is necessary for optimal results. This chapter is intended as an introduction to GPGPU aimed specifically for researchers with experience in surgical simulation, who wish to attempt a GPU implementation of their algorithms. We review the literature introducing the most important concepts, and discuss the hardware limitations we must adhere for optimal results.

An overview of the many applications of GPGPU is best obtained by exploring the on-line resource [89] and the books in the GPU Gems series [65, 153]. Moreover these surveys [149, 177] and course material [41, 42] highlight some commonly used algorithms. The survey by Strzodka et al. [177]

has a well-written introduction to scientific computation on the GPU. Based on a general data abstraction model for parallel programming, streams, a compiler and run-time system is available [40]. A few getting-started tutorials are available here [76]. This paper extends these references with a survey and discussion of GPU accelerated techniques aimed specifically towards surgical simulation.

7.2 GPGPU Concepts and Performance

The standard graphics pipeline in OpenGL and DirectX contains fixed functionality vertex and pixel shaders. A basic knowledge of this pipeline is assumed in the remaining chapter, and only a brief review is provided below. More information can be found in e.g. [130]. The vertex shader transforms the geometry (triangles) received from the application from local object space coordinates to window coordinates through a series of transformations. The color and texture coordinates are also computed for each vertex. The geometric primitive is subsequently rasterized, a term that describes the process in which the color of each pixel in the primitive is computed. The pixel shader is responsible for computing these colors. Based on each pixel's spatial position in the geometric primitive, the pixel shader receives the per-pixel interpolated color and texture coordinates as input. The fixed function pixel shader then computes the output color as a function of the input color and the texture colors. The texture colors are found from texture lookups using the per-pixel input texture coordinates.

In the past generations of GPUs the vertex and pixel shaders have gradually become fully programmable. In GPGPU we utilize this to write pixel shaders that no longer compute colors, but instead the scalars and/or vectors involved in a general computation. Each pixel can store a 4-tuple of floating point values in up to 32 bit precision per entry. We store the computed pixels directly in a non-visible GPU memory buffer as it is no longer meaningful to visualize these pixels directly. This buffer is actually a texture that can be used as input for subsequent iterations of our computations. Hence we have established a computational model in which we can both read from and write to GPU memory. A custom vertex- and a custom pixel shader program are uploaded to the GPU and applied in the subsequent processing of primitives in parallel. Due to this parallel nature of the GPU (the Radeon X1900XT has 48 pixel pipes, a GeForce 7900GTX has 24 pixel pipes), a high throughput can be obtained. A CPU-based physical simulation typically stores data in one-, two-, and three-dimensional arrays. On the GPU, data is stored instead in one-, two-, or three-dimensional textures. As the GPU works most efficiently on two-dimensional data structures, we transform both 1D and 3D textures to 2D textures in practice [90]. Naturally, some bookkeeping is necessary to handle this transformation. When

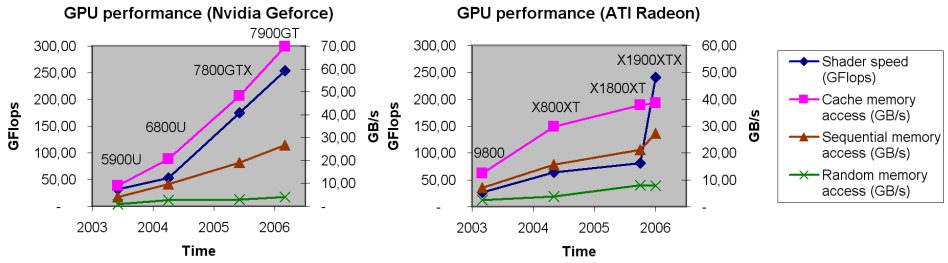


Figure 7.1: Observed performance from the most recent generations of Nvidia and ATI GPUs. Data was obtained using GPUBench [39]. Blue diamonds represent the shader performance measured in GFlops. Cache, sequential, and random memory access measured in GB/s are depicted in the remaining graphs.

a computation is invoked, the pixel shader receives an input texture coordinate that identifies the spatial position of the corresponding pixel in the input textures. If the algorithm requires access to neighboring pixels, this is achieved by offsetting the input texture coordinate before looking up in the respective textures. These offsets can be either global constants or obtained through an additional texture lookup at the current pixel. As will be explained in section 7.3, the type of offset depends on the underlying spatial discretization of the computational domain; whether it is structured or unstructured.

Using the benchmark test suite GPUBench [39] an overview of the performance of a system's GPU can be obtained. Figure 7.1 shows the graphs for the most recent GPUs from Nvidia and ATI. Looking at the number of floating point operations available per second (Flops) it can be observed that the current performance leaders provide roughly 250 GFlops. This number was obtained from the GPUBench test `instrissue`. This test measures the number of MAD instructions that can be executed per second on the present GPU. Since each shader operates on 4-tuples of floating point values and each MAD operation constitutes two floating point operations (a multiplication and an addition), the values reported by `instrissue` are multiplied by 8 for conversion to GFlops. Compared to the theoretical peak performance from a state-of-the art CPU (7.4 GFlops / 3.8 GHz Intel Xeon [59]) it should be clear why a GPU implementation of a surgical simulation can potentially boost performance. Discussing potential performance gains merely based on the shader speed reported in 7.1 does not provide a fulfilling picture however, as a typical GPU implementation of a surgical simulation would not be compute bound. More likely it would be memory bound - meaning that access to GPU memory would be the limiting factor. Consequently, figure 7.1 contains three graphs showing the observed memory bandwidth on the most recent GPUs. They are based on data obtained running GPUBench's

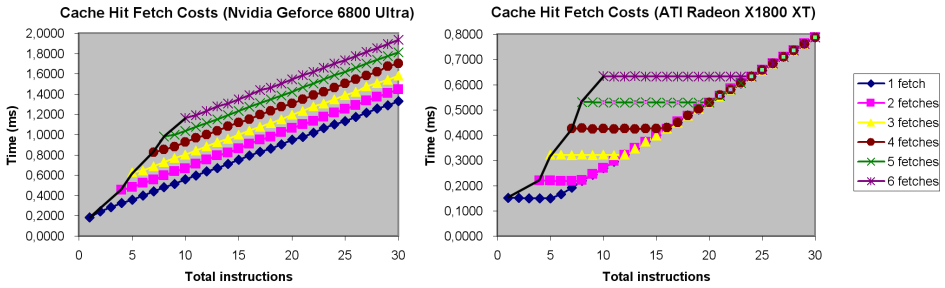


Figure 7.2: Cache hit memory access costs as a function of the number of shader instructions. Data was obtained from the most recent generations of Nvidia (left) and ATI (right) GPUs using GPUBench [39]. The number of texture fetches in each test was varied from 1 to 6.

`floatbandwidth` test on the respective GPUs [39]. It can be seen that cache memory access is significantly faster than sequential and random memory access. The cache memory bandwidth constitutes an upper limit in memory bandwidth, hardly attainable in real-world applications. Depending on the memory coherence of a given application, the growth in memory bandwidth could instead follow the lines depicting the sequential or random memory access bandwidth. Thus, designing memory coherent algorithms is of utmost importance.

To discuss whether a given application is compute or memory bound, the literature (e.g. [153]) defines the *arithmetic intensity* of an application as the amount of work that is performed per memory access. Applications with high arithmetic intensity are most likely compute bound while low arithmetic intensity is an indication of a memory bound algorithm. To discuss this issue in more detail, we once again resort to GPUBench: The test `fetchcosts` shows the execution time of a GPU program as a function of the number of instructions. Figure 7.2 shows the results from this test on two GPUs. Note that each test is comprised of six sub-tests that perform one to six memory cache accesses each. We will discuss below the results obtained on an ATI Radeon GPU. A similar (but not entirely identical) discussion can be made for the Nvidia based GPU, but we leave this discussion for the reader to complete. First notice the horizontal line segments in the rightmost half of figure 7.2 They show that for each memory access, a number of "free" computations can be made without influencing the overall execution time. Only as the non-horizontal (diagonal) part of the graph is reached, there is a cost associated to issuing additional instructions. From the figure we can predict the execution time of an application consisting entirely of memory reads (solid black line). Notice that the slope of this line is much steeper than the slope of the diagonal. The diagonal constitutes the border between a memory bound and a truly compute bound application: An

application with an arithmetic intensity that places it between the leftmost solid line and the diagonal is memory bound, while an application with an arithmetic intensity that places it on (or close to) the diagonal would be compute bound. As we shall see in the subsequent sections, surgical simulation algorithms implemented on the GPU are most likely memory bound, as the complexity in algebraic operations per memory access is limited. Experiments show however, that these GPU-based algorithms still significantly outperform their CPU-based counterparts.

7.3 Surgical simulation on the GPU

Many computational models for deformable surfaces have been proposed in the existing literature. We refer to the surveys [131, 81] and the chapter 5 for a detailed overview. We limit our description of GPU-based techniques to mesh-based deformable models (most often a mesh of triangles, tetrahedrons, or cubes) as this is the preferred approach in real-time surgical simulators that must handle arbitrary incisions and general changes in topology. For the remaining chapter we refer to nodes as the discretized points defining a mesh. We present an overview of the required GPU-based techniques to implement the most common deformable models: finite element models and Spring-Mass models. The reader should subsequently be able to define custom modifications to these general models in GPU terms. The implicit linear elastic finite element models presented in section 5.1 are discussed in section 7.4. The explicit finite element model (tensor-mass model) presented by Cotin et al. in [57] and the explicit Spring-Mass model from section 5.2 are discussed in section 7.5. Szekely et al. used a cluster of processors in [180] to realize a laparoscopic surgery simulator. Many of the general considerations on the design of parallel algorithms for numerical computations on multiple CPUs transfer directly to the parallel processor we introduce in this chapter, namely the GPU.

7.4 Implicit Finite Element Models

Using the notation section 5.1, finding the deformations in the implicit linear elastic finite element model reduces to solving either a static system on the form $K\mathbf{u} = f$ or a dynamic system on the form $M\ddot{\mathbf{u}} + C\dot{\mathbf{u}} + K\mathbf{u} = \mathbf{f}$, where M and C are diagonal mass and damping matrices, K is a symmetric positive definite matrix representing the topology and stiffness of the discretized mass points, and u and f are the deformation and external force vectors respectively. No matter the choice of system, it can be rewritten on the form $\tilde{K}\tilde{\mathbf{u}} = \tilde{f}$ following a finite difference time discretization for the dynamic system [30]. If we let n denote the total number of mass point in the finite element discretization, $\tilde{K}$ has dimensions $3n$ x $3n$ and entry k_{ij}

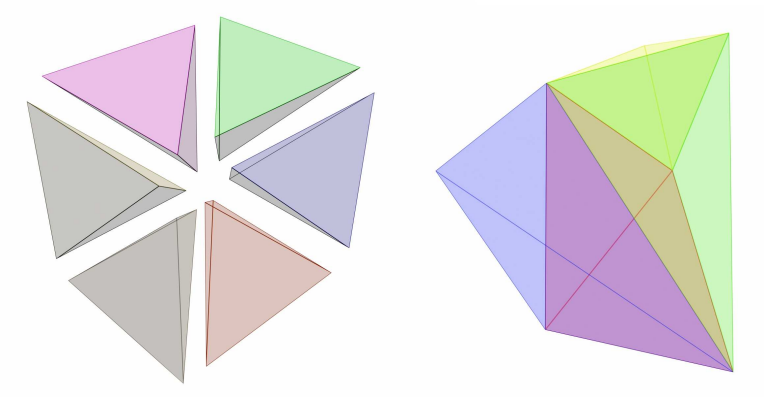


Figure 7.3: Structured (left) and unstructured (right) tetrahedral meshes. The choice of tetrahedralisation influences the layout of the resulting stiffness matrix in a linear elastic finite element model.

encodes the connectivity and stiffness between nodes i and j. In its most elementary form $\tilde{K}$ is sparse having non-zero entries only between connected mass points. Depending on the choice of spatial discretization, it can either be structured (banded) or unstructured. Figure 7.3 (left) illustrates a spatial discretization that leads to a banded matrix. Boxes (possibly consisting of six tetrahedra of fixed topology) are used as the basic spatial building blocks in a regular three-dimensional grid. The rightmost tetrahedralisation in figure 7.3 on the other hand, leads to an unstructured sparse matrix. Finally, using the condensation technique described in [30] and section 5.1.4, $\tilde{K}$ can be transformed to a smaller dense matrix of boundary nodes. We have distinguished between the different layouts of $\tilde{K}$ since they each call for their own distinct representation on the GPU. The three matrix layouts are 1) sparse (banded), 2) sparse (unstructured), and 3) dense. Solving the linear system of equations involves matrix and vector algebra. We discuss common linear algebra operations below for the different representations of $\tilde{K}$. This is followed by a discussion on GPU-based solutions to the linear system.

The first formulation of a general framework for numerical algebra on the GPU was published by Thompson et al. in [183] in a very machine-near language. Inspired by the BLAS and LAPACK libraries [60, 17], Krueger et al. subsequently published their initial work on a GPU-based counterpart [153, 111].

7.4.1 Sparse Banded Matrices

Figure 7.4(top) shows the representation in [111] of sparse banded matrices. Each band in an n x n-dimensional matrix A can be seen as a one-

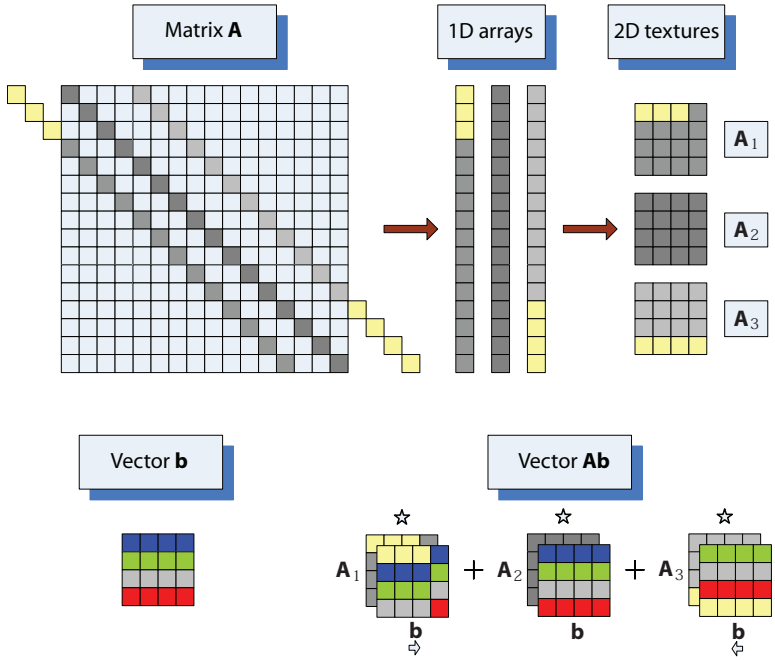


Figure 7.4: GPU representation of a sparse banded matrix A, a vector b, and the corresponding matrix-vector multiplication (adapted from [111]). Top: Each band represents a one-dimensional array which is stored in a 2D texture on the GPU (A_1-A_3). Zeroes are prepended or appended depending on the position of the band in the matrix. Bottom: A vector $\mathbf{b}$ is defined and multiplied to A. Each band in A is multiplied with $\mathbf{b}$ pixelwise. The products are added to form $A\mathbf{b}$. Notice that the texture coordinates used to access $\mathbf{b}$ are offset corresponding to each band.

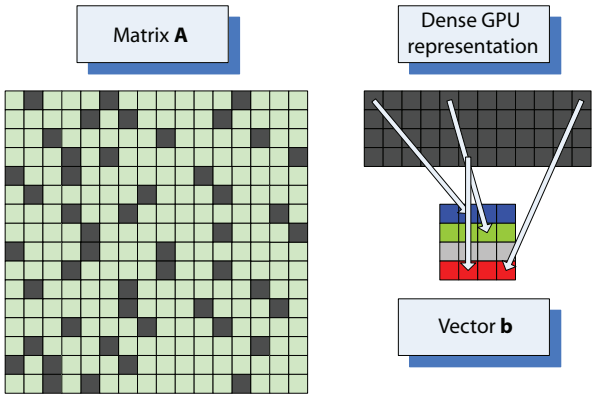


Figure 7.5: Unstructured sparse matrix representation specialized from [23]. We assume exactly three non-zero entries per row in this example (gray). These values are stored in a *dense texture*. For each pixel in this texture a pointer (i.e. texture coordinate) to the corresponding entry in vector b is stored (arrows).

dimensional vector of length n. As explained in section 7.2, we convert this vector to a two-dimensional texture on the GPU. Similarly, vectors $\mathbf{b}$ and $\mathbf{x}$ of dimension n are stored in textures of identical dimensions to the bands of A. In many computations it is necessary to find the matrix-vector multiplication $\mathbf{x} = A\mathbf{b}$. To achieve this we render a quad covering output texture $\mathbf{x}$ in multiple passes; one rendering pass for each band in A. Figure 7.4 (bottom) illustrates the three passes required for a tri-banded matrix-vector multiplication with this representation. In each pass the values in corresponding pixels in textures A and b are multiplied and added to $\mathbf{x}$. In each pass (except the pass using the matrix diagonal) the texture coordinates used to look up pixels in b are shifted to account for the corresponding band position. Several optimizations to this basic scheme is possible and discussed in [153, 111]. Since it is not possible to read from and write to the same texture during a pass, the accumulative writes to texture x must be implemented through two textures, one of which is bound as input and the other for writing in an alternating fashion. It is important to realize the parallel nature of the algorithm: For each of the three passes in the example in figure 7.4, the n entries in the result vector x are computed simultaneously, providing a significant speedup to CPU-based matrix-vector multiplications given a sufficient number of pixel pipes and texture memory bandwidth.

7.4.2 Sparse Unstructured Matrices

Sparse unstructured matrices are handled differently from the banded matrices above. Krueger et al. [153, 111] renders point-based primitives to implement such matrix-vector multiplications. We will instead describe a

different approach using texture pointers, as this relates nicely to this section's subsequent discussion of algorithm design that minimizes memory bandwidth. We illustrate a specialization of the general sparse unstructured matrix-vector multiplication by Bolz et al. [23] to find the product $\mathbf{x} = A\mathbf{b}$. We assume that a constant number of non-zero entries exist in each row of the sparse nxn dimensional matrix A. In figure 7.5 we use only three entries per row to reduce the size of the figure. We create a one-dimensional array of length $2x3xn$ to represent A. $3n$ entries are necessary to store the non-zeroes values in A. Furthermore, for each value we additionally store a pointer to the corresponding entry in textures $\mathbf{b}$. As always, we represent this one-dimensional array as a two-dimensional texture on the GPU. We render a quad covering our output texture x to initiate the parallel computation of $A\mathbf{b}$. For each pixel we look up the three non-zero values in the corresponding row in the texture representation of A from the input texture coordinate. The texture representation of A furthermore provides us with three pointers (texture coordinates) that are used to look up the values in $\mathbf{b}$ corresponding to the non-zero entries in A. The results from the three multiplications are added and stored in x. Again, it is important that the reader recognizes the parallel nature of the algorithm, in which each entry in x is computed in parallel.

With the understanding of sparse matrix representations on the GPU, the reader should have the prerequisites to derive representations of dense matrices as these are more straightforward than those of sparse matrices. We refer to [153, 64, 73] for completeness.

We now return to solving the linear system $\tilde{K}\tilde{\mathbf{u}} = \tilde{f}$ that was defined initially in this section. Since $\tilde{K}$ is symmetric and positive definite, one approach to finding $\tilde{\mathbf{u}}$ is through the conjugate gradient algorithm. Using their respective frameworks for linear algebra, both Krueger et al. and Bolz et al. showed how to implement the conjugate gradient algorithm on the GPU in [111, 23]. An alternative approach guaranteed to converge to the right solution for arbitrary starting configurations is the Gauss-Seidel iterative process (again since $\tilde{K}$ is symmetric and positive definite). Contained in [111] is a short section discussing the implementation of this algorithm on the GPU. Closely related to Gauss-Seidel's method is the Jacobi method. In contrast to Gauss-Seidel's methods, Jacobi's method is ideally suited for a parallel implementation. For this reason it has been used intensely in previous GPGPU publications, e.g. in several chapters in [153, 65], and in [82, 165].

The representation of banded matrices as shown in figure 7.4 results in a minimum number of texture fetches: only the actual values needed for a matrix-vector multiplication are read from texture memory. The unstructured sparse matrix representation on the other hand requires further texture lookups to perform a matrix-vector multiplication, as pointers are stored in textures and the corresponding values only obtained through an

additional texture lookup. Looking back at the discussion related to figure 7.2 it should be clear why the banded representation performs better than the unstructured alternative. It is simply due to the lower number of texture fetches involved. Consider also the limited number of algebraic operations performed per memory access in both approaches (that is the arithmetic intensity is low). Given the memory bandwidth on current GPUs both matrix representations are memory bound, although positioned differently in the graphs in figure 7.2. It was Fatahalian et al. who initially reported that memory access is indeed the limiting factor for dense matrix-matrix multiplications on the GPU [64].

The reader is encouraged to examine the GPU-based surgical simulator by Wu et al. [196]. They used an implicit finite element solver through the conjugate gradient algorithm and obtained a two-fold acceleration. This work was done on an Nvidia GeForce 5950 Ultra however. As is clear from figure 7.1, both the GPU speed and memory bandwidth have increased five- to ten-fold since on the most recent GPU's.

7.5 Explicit Models: Spring-Mass and Tensor-Mass Models

We return to the general equation of Newtonian motion: $M\ddot{\mathbf{u}}+C\dot{\mathbf{u}}+K\mathbf{u} = f$ [30, 57]. We now seek to solve the system of differential equations through an explicit time integration scheme rather than by the implicit method discussed in section 3.1. A particularly well suited explicit integration scheme is Verlet integration [187], a scheme in which the position of each mesh node for the subsequent time step is calculated from its positions in the two previous iterations and from an elastic force vector (acceleration vector). No additional information, e.g. velocities, needs to be stored and calculated. The force vector is calculated locally from each node's connectivity in the mesh. We denote the force vector corresponding to a Spring-Mass system as $\hat{\mathbf{f}}_i$ and the force vector relating to the tensor-mass system as $\tilde{\mathbf{f}}_i$. Using the notation from [57] these forces are then defined as

$$\hat{\mathbf{f}}_i = \sum_{j \in N(\mathbf{x}_i)} k_{ij}(\|\mathbf{x}_i\mathbf{x}_j\| - l_{ij}^0) - \frac{\mathbf{x}_i\mathbf{x}_j}{\|\mathbf{x}_i\mathbf{x}_j\|}$$

and

$$\tilde{\mathbf{f}}_i = K_{ii}\mathbf{x}_i^0\mathbf{x}_j + \sum_{j \in N(P_i)} K_{ij}\mathbf{x}_j^0\mathbf{x}_j$$

where $\mathbf{x}_i$ denotes the position of node i, $\mathbf{x}_i^0$ is the initial (undeformed) position of node i, $\mathbf{x}_i\mathbf{x}_j$ is the vector between nodes i and j, l_{ij} and k_{ij} is the

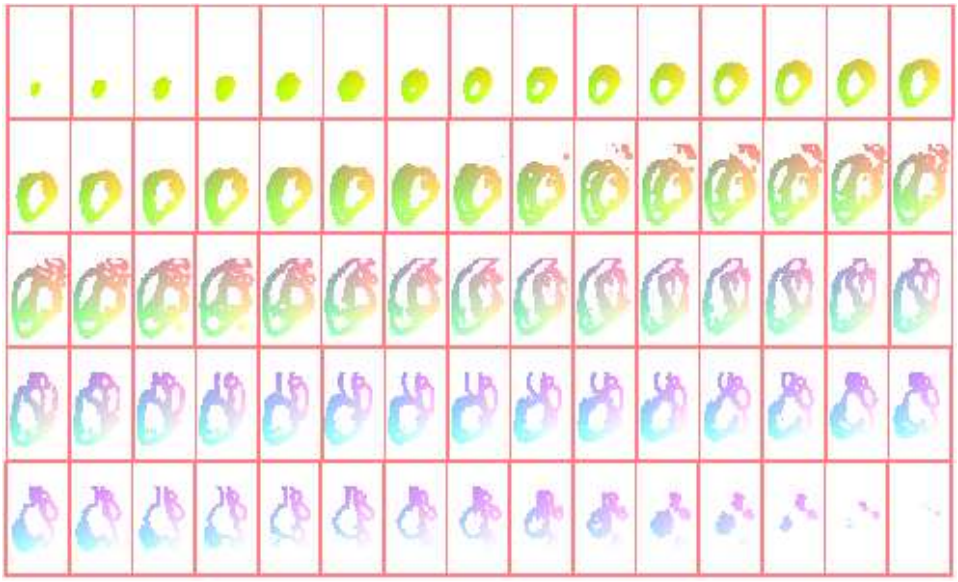


Figure 7.6: Position texture, inspired from [5]. A regular 3D grid of nodes is mapped to a 2D texture. The colors of the individual pixels denote the corresponding particle's position.

spring rest length and stiffness respectively between nodes i and j, and K is the rigidity (or stiffness) matrix of the linear elastic finite element model.

A two-dimensional GPU- and Spring-Mass based cloth simulation using Verlet integration was presented in [86]. It is sufficiently simple to be recommended for inexperienced GPU programmers. The author presented a three-dimensional, volumetric Spring-Mass based surgical simulator implemented on the GPU in [3, 5]. This method is presented in full in the next chapter, chapter 8. [5] compares two Spring-Mass implementations, one in which nodes were confined to a regular three-dimensional grid, and one in which node positions were unrestricted and springs explicitly represented in a connectivity texture. Overall, a twenty-fold acceleration over a similar CPU-based system was achieved for the first method, while the latter achieved a ten-fold acceleration. Our results were obtained on a GeForce 6800 Ultra. It is clear from figure 7.1 that both the shader speed and memory bandwidth have increased significantly since and even better results could be obtained on the most recent generations of GPU's. In the faster of the two methods they use a position texture to store the positions of the nodes in the Spring-Mass system. An example of one such texture is depicted in 7.6. To initialize parallel computation of each time-step, a quad at the size of this texture is rendered in the output buffer. A depth test is used to prevent that calculations are wasted on the white (void) particles. The forces are computed for each pixel (node): The two most recent position textures

are provided as input textures to the pixel shader. By adding to the input texture coordinate the fixed offset to each neighbor, each neighboring node's position can be looked up. As the nodes are restricted to a regular grid, the individual spring's rest lengths are known and need not be looked up. Furthermore, the spring stiffness is also kept constant. Thus, the only texture fetches involved are those used to obtain the connected node's positions. I.e. we use only the minimal number of texture fetches. This is important in the light of the discussion concerning the cost of texture fetches (See figure 7.2). Our alternative approach uses texture lookups to fetch the texture coordinate of each neighbor, essentially working as a list of pointers to nodes. This doubles the overall number of texture fetches, consequently reducing the simulation rate by a factor of two. Replacing the spring induced forces $\hat{\mathbf{f}}_i$ with the tensor-mass forces $\tilde{\mathbf{f}}_i$ would instead solve an explicitly formulated finite element model. For each connected node we need then an additional texture lookup in the stiffness matrix to obtain K_{ij}. Consequently we can expect the tensor-mass model to run at approximately half the speed of the Spring-Mass model. Compared to a CPU-based implementation however, it is still significantly faster. The most recent performance measurements are found in Sørensen et al. [9], who report simulation rates exceeding 1 kHz using a GeForce 7800 GTX on a Spring-Mass system consisting of 20.000 nodes connected with 18 neighbors each in a regular volumetric mesh (grid).

7.5.1 Alternative Spring-Mass Systems

Our methods presented in [3, 5] are simple to implement and run fast, but use a significant amount of texture memory: The position texture approach wastes memory representing void particles, while the connectivity texture approach allocates memory for a constant number of neighbors per node wasting texture memory if the number of neighbors varies significantly throughout the mesh. Later[1] papers [77, 78] and a presentation [198] have presented alternative methods that we will present and analyze in this section.

In [77] Georgii et al. have described the implementation of a Spring-Mass system on the GPU. The general strategy is the same as our explicit method in [4], namely to record a list of pointers to neighboring particles. In [77] the basic structure of the Spring-Mass system is tetrahedrons. Each particle is related to one fragment as in [4]. For each node, information on all adjacent tetrahedra are looked up and used for the force calculation. Each tetrahedra amounts to five texture lookups; two texture-lookups are used to retrieve texture-coordinates to neighbors as well as spring-stiffness and tetrahedra volume, and three dependent texture lookups to retrieve the actual position of the three neighboring particles. Since the number of tetrahedra adja-

[1] Although only by a few months

cent to a given particle is not constant, [77] use a stack of *valence textures* to encode different levels of connectivity, ensuring that fragments only do the number of lookups into tetrahedra that are necessary. This is in contrast to our explicit method in [5] where we do lookups proportional to the maximum number of neighbors (maximum valence). In [77] the same neighboring particle might be subject to a considerable amount of lookups since each tetrahedra potentially shares edges, and each tetrahedron is handled separately. This is a serious bottleneck since all the presented algorithms are most probably memory access bound (See figure 7.2).

In our method in [5] each spring force is calculated exactly twice and in [77] each spring force is calculated at least twice. This led Georgii et al. to develop an edge-centric data structure instead [78], that iterates over springs rather than nodes to calculate the spring-forces. In a first pass a texture-lookup retrieves positions of neighboring particles of the spring, this information is used to retrieve the current position of the particles through two texture-lookups, and finally the spring-force is calculated and written to the spring-force texture. To scatter the force of each spring to the two connected particle, two point primitives are rendered at the position of the connected particles in the texture of particle positions. The actual force-vector is retrieved through binding the texture of spring-forces as the color-component of a vertex-array, and the addition of forces is realized through additive blending of the framebuffer. This reduces the arithmetic intensity of their algorithm, since each spring force is now computed only once, whereas they are computed twice in the previous methods [5, 77]. As all the presented algorithms are most likely memory bound (See figure 7.2) this is not a major advantage however. We are more interested in examining the number of texture fetches used in the edge-centric approach Let us analyze this in more detail. The number of texture lookups $t_1$per particle in the edge-centric method of [78] is:

$$t_1 = \frac{e + 2e}{p} = 3\frac{e}{p}$$

where e is the total amount of edges and p the total amount of particles. In the implicit method of [5] the number of texture-lookups is simply the maximum amount of neighbors, E;

$$t_2 = E$$

for a totally regular mesh, $E = 2\frac{e}{p}$ and consequently have this relation between t_1 and t_2 :

$$t_2 = E = 2\frac{e}{p} < 3\frac{e}{p} = t_1$$

In the regular mesh shown in figure 7.6) there are in average 7.5 edges per particle, and we have a maximum of 18 spring per particle, i.e. $\frac{e}{p} = 7.5$, and

$E = 18$. Considering this example we get:

$$t_1 = 3\frac{e}{p} = 3 \cdot 7.5 = 22.5$$

and

$$t_2 = E = 18$$

The comparison is presented here as one perspective on the resource utilization of different methods. The edge centric method [78] is clearly much more flexible than the implicit method of [5] but at the cost of additional texture-lookups. Some other factors might turn out to be the bottleneck of the edge centric method though. The transfer of texels to the color values of a vertex-array is a function that should be equivalent to a texture-lookup in performance. (adding another 7.5 to the texture lookups in the example). The additive blending and rendering of a large number of point-primitives is using a considerable amount of resources. Blending can furthermore not be executed in 32bit on current GPU's. An additional context switch from spring to particle render-texture is furthermore necessary.

In [198] a cloth simulation based on iterative relaxation (that is, not Hookean-springs) is realized on the GPU. They divide the spring-constraints into independent sets that require a pass of the algorithm each. A cloth with shear and structural springs (as in figure 5.3) would require 8 passes, where each fragment (representing a particle) does two texture-lookups.

7.6 Visualization and Interaction

Depending on the chosen simulation model, the result of each time step is either a texture of node deformations (implicit model) or a texture of node positions (explicit model). In either case a deformed surface triangle-mesh of the modeled organ can be visualized from a static display list of the initial mesh configuration through a dedicated vertex shader [5, 3]. For each vertex in the visualized surface mesh, the application provides the required texture coordinate to look up the corresponding deformation vector or particle position. The vertex shader can thus compute the deformed vertex position for the current time step.

From both section 7.4 and 7.5 it is clear that a structured spatial discretization of the simulated volume results in the fastest algorithms due to a minimum number of texture lookups required in each time step. This can however result in a jagged (stair-like) look of the modeled morphology as illustrated in figure 7.7. To overcome this problem I propose in chapter 9, based on the paper [6], to fully decouple surface visualization from the underlying volumetric simulation. Each vertex on the surface model is represented by an offset from the nearest node in the simulation mesh. Figure

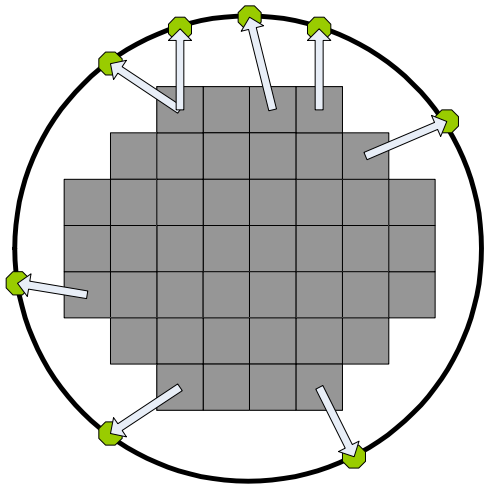


Figure 7.7: Surface visualization (circle) DE-coupled from a volumetric simulation of a sphere discretized to a regular grid (gray). The green circles represent vertices on the surface mesh that can be sampled at any resolution. Each vertex is represented by an offset vector (arrow) from the nearest simulation node.

7.7 shows a smooth surface drawn by this method. The offset vectors are expressed in the tangent-space of the surface, and the surface thus correctly deforms based on the deformation of the associated nodes in the simulation mesh.

Interaction with a GPU-based surgical simulator is the final issue to be discussed in this chapter. The overall question is whether to resolve user interaction on the CPU or on the GPU. If the CPU is chosen one must be careful not to transfer large amounts of data from the GPU to the CPU in each frame, as this would introduce a performance bottleneck. Consequently, interaction that involves computations on the current state of the simulation is probably best implemented on the GPU. Peripheral devices can only be communicated with through the CPU however, so a minimum amount of per-frame data transfer cannot always be avoided. In chapter 10 based on the paper [9] I show how to implement force feedback from a GPU-based simulator with limited performance penalty. Several groups have recently published algorithms for GPU accelerated collision detection [85, 194, 48].

Chapter 8

Spring-Mass on the GPU

The following chapter is a combination of the papers [4, 5] with more details on the implementation and the current state of the simulator. In particular section 8.7.1 on probing has been extended with a technique for restricting the movements of particles into unwanted configurations. I have also included some illustrative source code from the simulator not previously available. The original papers included a discussion of a simple visualization and a range of screen-shots. In this thesis the issue of visualization will be delayed until chapter 9 where the missing screen-shots will also be shown as motivation for a decoupling of simulation and visualization.

8.1 Introduction

Many surgical simulators used in practice are based on Spring-Mass deformable models due to performance reasons. The Spring-Mass model is considered physically based and achieves real-time visualization and fast convergence for geometry of moderate size.

In surgical simulation in general, there is a trade-off between the costs of calculations, how realistic the tissue-deformation is reproduced, and how detailed the morphology being simulated appears. It is our goal to simulate a very high degree of morphological detail in real-time and use a physics-based model. As an example, the cardiac morphology is complex and requires a high degree of geometric detail to be modeled accurately.

The Graphics Processing Unit (GPU) is a programmable parallel processor capable of processing vertices and fragments in parallel. The GPU is designed to perform graphics rendering based on geometric primitives and resulting in pixel coloring. Recently the GPU has become programmable to a degree that makes it useful for general-purpose computation.

We present a surgical simulator based on a Spring-Mass system accelerated by an implementation on the graphics processing unit (GPU). The purpose is to achieve a considerable speedup due to the parallel processing

capabilities of the GPU. This acceleration could be used to increase the accuracy and convergence of the numerical calculations and to increase the complexity of the simulated morphology. At the time of writing the paper on which this chapter is based, only simple Spring-Mass systems had been implemented on the GPU[86] - limited to simple 2D shapes. Other physical based systems have been implemented on the GPU previously. E.g. conjugate gradients for fluid simulation [23], CML systems for the simulation of a range of physical based phenomena [91], and more general linear algebra operators [111]. See chapter 7 for more detail.

Slow data transfer from the GPU to the CPU has been an additional bottleneck when handling interaction and visualization. With the recent generation of GPUs (GeForce 6800, Nvidia, USA) both simulation, interaction, and visualization of a Spring-Mass based surgical simulator can be accelerated on the GPU.

8.2 Parallel Computation of the Spring-Mass System

For reference we rewrite the equation of the Spring-Mass system (5.3) from page 50:

$$m_i \ddot{\mathbf{x}}_i = -y_i \dot{\mathbf{x}}_i + \sum_j \mathbf{g}_{ij} + \mathbf{f}_i \tag{8.1}$$

Likewise we rewrite the equation for spring forces (5.4) from page 50:

$$\mathbf{g}_{ij} = k_{ij} \left(l_{ij} - ||x_i - x_j|| \right) \frac{\mathbf{x}_i - \mathbf{x}_j}{||\mathbf{x}_i - \mathbf{x}_j||} \tag{8.2}$$

Lastly we rewrite the equation for Verlet integration (5.5) from page 53:

$$\mathbf{x}(t+h) = 2\mathbf{x}(t) - \mathbf{x}(t-h) \tag{8.3}$$

The GPU can in many ways be regarded as a shared memory parallel architecture, although with many limitations. We will begin by some considerations on the design of a simple parallel algorithm to solve a Spring-Mass system on a shared memory parallel architecture. Fortunately the solution to (8.1) is straightforward to parallelize and solve iteratively on a shared memory parallel architecture. Processor P_i is responsible for updating information regarding particle $\mathbf{x}_i$. At each time-step and for each particle the following must be done:

1) Calculate spring forces based on the distance to neighboring particles

2) Numerically integrate to calculate the new position of particle $\mathbf{x}_i$.

Through the shared memory architecture, processor P_i can retrieve the current position of all particles for force calculation. In the next section we will review the GPU as a parallel processor with shared memory. In section 8.4 we will discuss the implementation of integration of particle positions on the GPU. In section 8.5 we will represent two methods of representing and calculating spring forces on the GPU.

8.3 GPU Pipeline

The focus of this chapter is to express the calculation of the Spring-Mass system effectively in terms of the hardware accelerated features of the GPU. Recently, the vertex processor and fragment processor have become programmable. Both processors are parallel processors with a number of pipelines working simultaneously. Vertex and fragment computation can depend on previous iterations through texture lookups and render-to-texture functionality exposed through Pixel Buffers (PBuffers). A PBuffer can be bound either as the rendering target or as a texture. Reading from textures and writing to the rendering target implements shared memory in fragment programs. Throughout this section we will refer to the PBuffer as a texture or as the rendering target interchangeably depending on the context. Using floating point texture extensions we can do computation on IEEE 32 bit floating point numbers. These features enable general purpose computation on the GPU.

When rendering a geometric primitive the vertex information is processed by a programmable vertex shader. Vertices are transformed and per vertex information is interpolated and transferred to the programmable pixel shader. The pixel shader processes this information as well as additional input in the form of textures to compute the final coloring of a fragment (a generalized pixel).

If we regard each fragment as a representation of a particle position, we can design fragment programs to solve equation (8.1). An off-screen rendering buffer (PBuffer) is used to store the calculated positions. Each position in the PBuffer maps uniquely to the position of one particle. Each component of each particle position $\mathbf{x}_i = (x, y, z)$ is represented as a 32-bit floating point value in the red, green and blue part of a pixel through OpenGL float-buffer related extensions. The PBuffer containing positions will in the remaining thesis be referred to as the *position texture*.

8.4 Integration Loop on the GPU

The fragment processor was chosen for our implementation as there are generally more fragment pipelines available than vertex pipelines. Equally important, texture lookups are more efficient in fragment programs.

If we assume for the moment that we have calculated the forces affecting a particle, we must then design a loop that can integrate the particle position. A time-step of the simulation is computed by activating a dedicated fragment program by rendering a single quad covering the entire PBuffer. In this program, the supplied texture coordinates refer to particle positions in the PBuffer. Through a texture-lookup the actual positions are retrieved. The texture coordinates are given at the vertices of the quad and automatically interpolated throughout all fragments. A one-to-one mapping of the PBuffer between the input texture and output buffer is then established.

The choice of the numerical integration method is influenced by the properties of the fragment program's input and output. Verlet integration (equation 8.3) is well suited for our GPU implementation since calculations of future positions depend exclusively on the previous two positions. The two previous particle positions can be provided as input through textures. These textures are available from previously written PBuffers. The result of Verlet integration is the position; this conserves bandwidth compared to numerical integration resulting in more than one vector.

8.5 Spring Connections and Force Computation

In this section we explain in detail the spring connections and force computations of two alternative Spring-Mass implementations for the GPU. The first method represents spring connections explicitly, while particles in the second method are connected implicitly based on their location in a three dimensional grid. The explicit method allows the most freedom in representing spring connections and particle locations at the cost of some simulation speed compared to the implicit alternative.

8.5.1 A Spring-Mass System with Explicit Connections

In a general Spring-Mass model every particle can be arbitrarily connected to other particles with no special order assumed. We encode this spring connectivity in a separate floating point texture, in the following referred to as the connectivity texture. The usage of the connectivity texture is illustrated in figure 8.1. The connectivity texture defines for each particle p_i a list of springs to particles p_j by storing 1) the texture coordinate in the position texture of each particle p_j and 2) the rest length l_{ij} and stiffness k_{ij} The connectivity texture remains constant as long as no changes in particle connectivity are made. From an element of the connectivity texture a

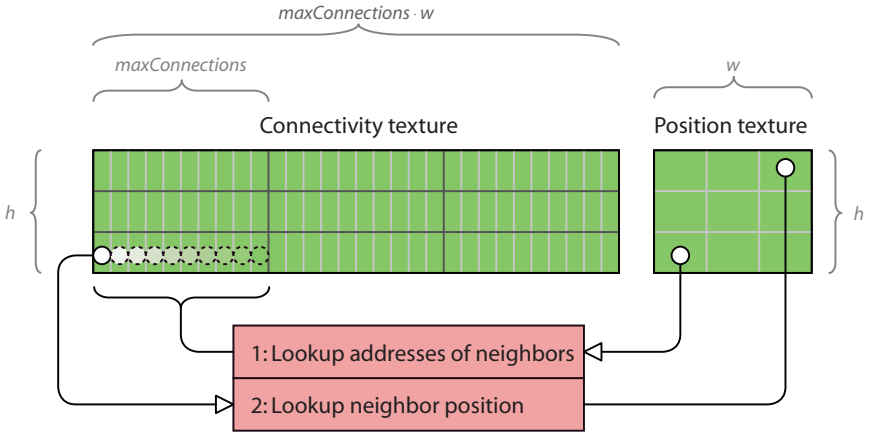


Figure 8.1: Layout of the connectivity texture. For a fragment (x, y) in the position texture, the connectivity texture contains the texture-coordinates of neighbors in coordinates $(x \cdot maxConnections, y)$ to $((x + 1)maxConnections, y)$.

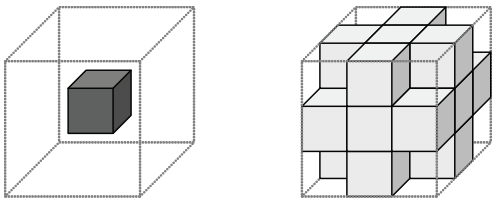


Figure 8.2: Particle connectivity in a 3D grid. Each particle (left) is connected to 18 neighbors (right)

texture lookup provides the neighboring particle position $\mathbf{x}_j$. By iterating over all springs connected to particle p_j it is possible to calculate equation (8.2). If a node has less than the maximum number of neighbors, the spring rest length or stiffness can be set to 0 to indicate that a given element of the connectivity texture should not be considered.

Note that the spring forces will be calculated twice (with different sign), from each particle connected to it. We have chosen not to take advantage of the fact that $\mathbf{g}_{ij} = -\mathbf{g}_{ij}$ from equation (8.2), as it would require a second pass. Additional information would also have to be given to each fragment to indicate the sign of the force of a given spring

8.5.2 A Spring-Mass System with Implicit Connections

A potentially limiting factor of the approach of explicit connections is the intense use of texture lookups to retrieve the positions of neighbor particles. What if the texture coordinates of neighbors could be given directly as in-

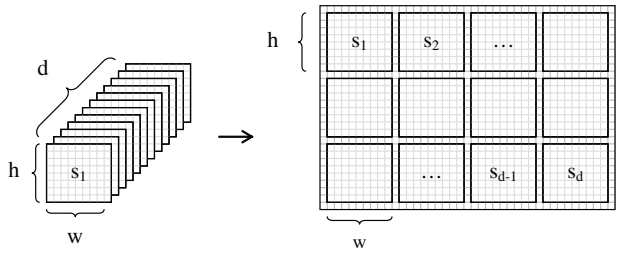


Figure 8.3: The flat 3D-texture approach. The 3D volume of voxels is mapped to a 2D texture by laying out each of the d slices of size $h \cdot w$ in the 2D texture one after another. The slices are padded with elements containing unique alpha values of zero to represent the volume borders.

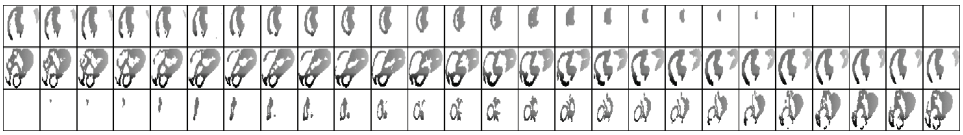


Figure 8.4: The position-texture of a 42.745 particle pig heart. White areas are grid points not associated with particles.

put to the fragment program to enable a single texture lookup to retrieve neighbor positions? In the method presented in this section, the texture coordinates needed to lookup neighboring particles are given directly as input to the fragment program from the output of the vertex program. To avoid that the vertex processor becomes a bottleneck by rendering individual fragments as geometry, we again conceptually invoke the fragment computation with a single quad covering the position-texture. Texture coordinates are specified for each vertex and interpolated automatically by the rasterizer before being received as input in the fragment programs. This means that particles must be connected in such a way that their neighbors can be fetched from per vertex interpolated texture-coordinates. That is, particles should be connected in a fixed pattern. We use a 3D grid as depicted in figure 8.2 to construct a Spring-Mass system with eighteen springs constraining axis aligned changes as well as shearing.

The grid must be mapped to the two-dimensional position-texture to use the proposed approach. This is achieved through a derivation of the flat 3D-texture approach [90], see figure 8.3 and an example in figure 8.4.

The quad rendered to invoke fragment computation is given eighteen texture coordinates that are offset a fixed amount from the texture coordinates identifying the particle, see figure 8.5. Neighbors within a slice are addressed by offsetting the fragment position by ± 1 along the width or height of the texture. Neighbors between slices are addressed by furthermore offsetting with either $w + 1$ or $-(w + 1)$ to reach a neighbor in the previous and next

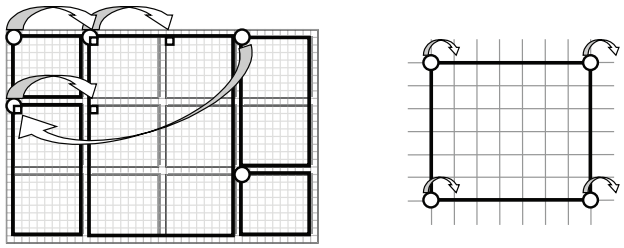


Figure 8.5: Vertex texture coordinates for neighbor positions. For clarity we will only show two of the 18 possible texture coordinates. a) depicts the five quads rendered to ensure correct wrapping. The texture coordinate pointing to the neighbor directly below in depth is shown for the top-left vertex. b) shows the four texture coordinates given with all quads for the right neighbor.

slice respectively. In this way each fragment program knows the texture coordinates and hence the positions of its neighbors by one texture lookup each. Particles on the border of a slice are not intended to be connected to all their fragment-neighbors. To alleviate this problem we pad the slices with inactive particles that we do not seek to process - notice the padding in figure 8.3. We define inactive particle fragments to have an alpha value of zero. This value is detected in the fragment program, and the calculation is skipped for the associated springs. The 18 texture locations of neighbors plus the location of the particle itself are given through 8 texture-coordinates (with $8 \cdot 4 = 32$ values). A simple mapping would only give us 16 texture-coordinates when we need 19 (including a texture coordinate for the particle itself). Since some of the neighbors have identical components (e.g. right neighbor and bottom right neighbor) we can reuse identical components.

Instead of the conceptual model of rendering only one quad to invoke full fragment computation, it is necessary to render five quads with texture-coordinates constructed to take into account the border-cases of the flat 3D-texture approach by wrapping the addresses for neighbors, as illustrated in figure 8.5.

By this implicit connectivity, $\mathbf{g}_{ij}$ in equation (8.2) can be determined using only 1 texture lookup per neighbor instead of 2 per neighbor as was the case in the explicit method. Furthermore we have fixed the possible rest lengths to all neighbors to either 1 or $\sqrt{2}$. We can hereby simplify the force computation somewhat if we assume all stiffness coefficients k_{ij} to be equal

to the constant k:

$$\begin{aligned}\sum_j \mathbf{g}_{ij} &= \sum_j \frac{1}{2} k_{ij} \left(l_{ij} - \|\mathbf{x}_i - \mathbf{x}_j\|\right) \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|} = \\ &\frac{1}{2} k \left\{ \left(\sum_{j \in D_1} \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|} - \sum_{j \in D_1} \mathbf{x}_i - \mathbf{x}_j \right) \right. \\ &\left. + \left(\sqrt{2} \sum_{j \in D_2} \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|} - \sum_{j \in D_2} \mathbf{x}_i - \mathbf{x}_j \right) \right\}\end{aligned} \tag{8.4}$$

$D_1$is the set of neighbor particles with distance 1 and D_2 is the set of neighbor particle with distance $\sqrt{2}$. Using this expression as the basis for the fragment programs saves instructions. Equally important this gives us a sum of unit-vectors, which we can use later to calculate normals after deformation in chapter 9. This expression transfers to the cg code in algorithm 1.

As in [86] the geometry is connected in a regular grid. Unlike [86] however, we operate on a 3D grid. The grid must furthermore approximate an arbitrary geometry. Hence, it is necessary to exclude some of the particles in the grid. Even though we can conditionally skip calculation of these particles in the fragment programs, there is some overhead involved in doing so. To overcome this problem we initially fuse an image of the model in the OpenGL depth buffer once, and set up a depth-buffer test to fully eliminate processing inactive particles. Conceptually we carve out the morphology in the grid of particles. Grid points are active particles in the simulation if they are inside the myocardium or a vessel wall; otherwise they are discarded by the depth-buffer based cull. See figure8.4 for an example.

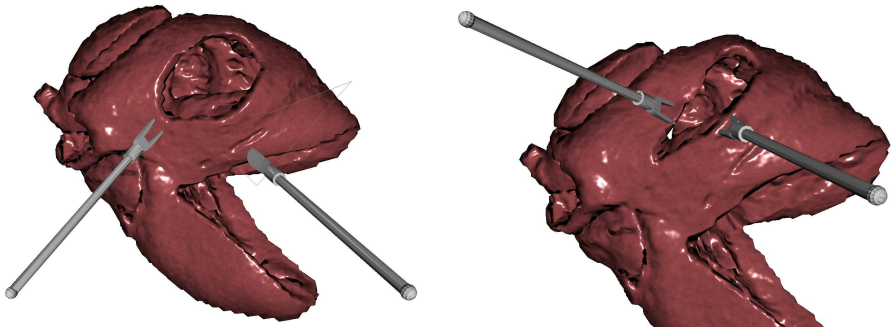
8.6 GPU Spring-Mass Performance Results

This section presents performance measures of the described GPU Spring-Mass systems and compares them to a CPU implementation. The GPU based Spring-Mass systems were implemented in CG, OpenGL arbfp1 (including `fragment_program2` features) as well as Visual Studio C++. The CPU Spring-Mass system was implemented in Visual Studio C++ and compiled with all optimization flags set and optimized for speed. The CPU Spring-Mass system is a direct port of the GPU implementation. The tests were run on a Pentium IV 3GHz machine with a Gainward CoolFX Ultra/2600 (GeForce 6800 Ultra) graphics card. Performance results for different sizes of GPU Spring-Mass systems in comparison with the CPU implementation are reported in 8.7 and 8.1. The original articles [4, 5] included an example dataset of 42.745 particles running in real-time. Although the

Algorithm 1 Partial cg code for force-calculation according to 8.4.

```
float3 calculateAcceleration
    (uniform sampler2D posTex,
     float3 pos,
     float2 texCoord,
     ...
     float4 texCoord7,
     float stiffness,
     out float3 sumOfUnitVectors )
{
    ... initialisation of variables to 0
    // Neighbors with rest length 1
    NPos = tex2D(posTex, float2(texCoord.x,texCoord1.y) );
    if (NPos.w!=0.0)
    {
        float3 d = pos.xyz-NPos.xyz;
        sumOfDifferenceVectors += (d);
        sumOfUnitVectors += normalize(d);
    }
     ... continues for all 6 neighbors of rest-length 1
    float constraintDist = 1;
    acc = acc + stiffness * ( constraintDist *
              sumOfUnitVectors - sumOfDifferenceVectors );
    normal += sumOfUnitVectors;
    // Neighbors with rest length √2
    sumOfUnitVectors = float3(0,0,0);
    sumOfDifferenceVectors = float3(0,0,0);
    NPos = tex2D(posTex, texCoord4.xy);
    if (NPos.w!=0.0)
    {
        float3 d = pos.xyz-NPos.xyz;
        sumOfDifferenceVectors += (d);
        sumOfUnitVectors += normalize(d);
    }
    ... continues for all 12 neighbors of rest-length √2

    normal = normal + sumOfUnitVectors;
    constraintDist = √2
    acc = acc + stiffness*(constraintDist*
              sumOfUnitVectors - sumOfDifferenceVectors );
    return acc;
}
```



(a) (b)

Figure 8.6: A Pig heart consisting of 42.745 particles in a regular grid reconstructed from a CT data set. a) original geometry b) deformed by grabbing. Notice that this is a very early screen-shot from the simulator.

Table 8.1: Performance comparison for the CPU, Explicit Connections GPU and Implicit Connections GPU. In the 2nd , 3rd and 4th column we present iterations per second. GPU/CPU columns present the GPU speedup in comparison to the CPU. The GPU was a GeForce 6800 Ultra and the CPU a 3ghz Pentium IV

	Method				
Nodes	**CPU**	**Implicit GPU**	**Implicit GPU / CPU**	**GPU**	**Explicit GPU / CPU**
10.000	45,8	839,8	18,7	457,1	10,2
20.000	20,2	476,9	23,6	234,4	11,6
40.000	9,9	264,6	26,9	121,0	12,3
50.000	7,8	218,0	28,1	99,0	12,7
100.000	3,3	104,1	31,4	48,5	14,6

visual quality is not representative of the current simulator we show the figure as a reference, see figure 8.6.

8.7 Interaction

To interact with the simulation the system can use either two magnetically tracked Polhemus Fastrak styluses with two buttons each or two Phantom Omni devices with haptic feedback (See figure 8.8). Each stylus provides position and orientation of the instruments in space. We support three modes of interaction; probing, grabbing and cutting. The main issue in all these interaction methods is to avoid communication between the CPU and GPU every frame, as this will introduce a bottleneck in the application.

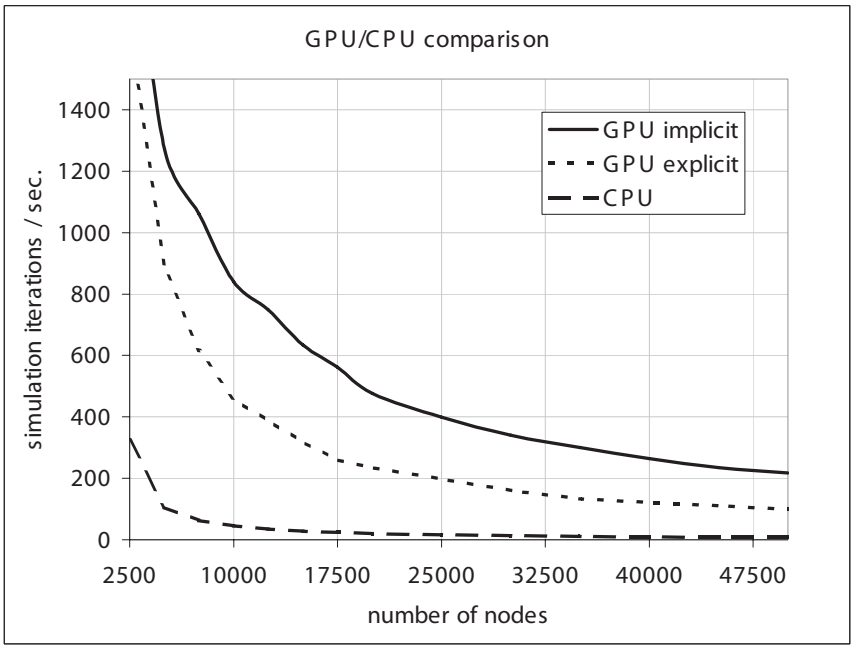


Figure 8.7: The GPU Spring-Mass systems in comparison to a CPU implementation. The GPU was a GeForce 6800 Ultra and the CPU a 3ghz Pentium IV

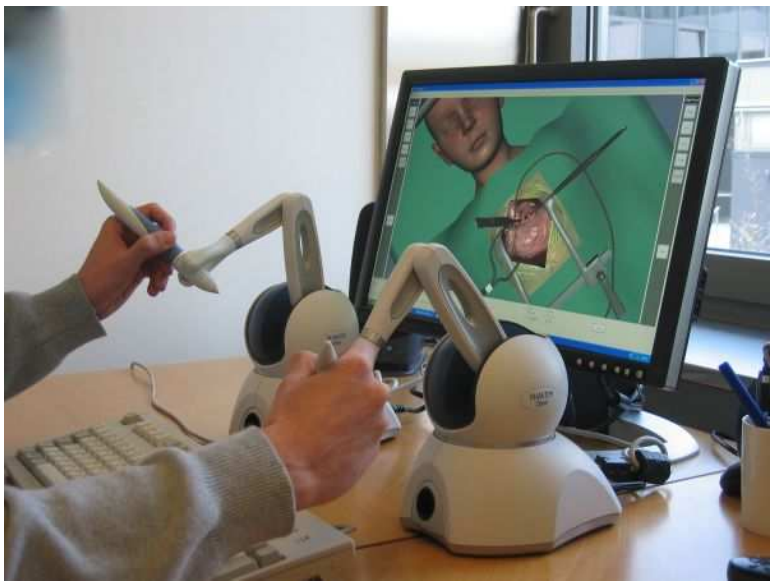


Figure 8.8: This picture shows the current simulation setup with Phantom Omni feedback devices and the virtual environment for training surgical procedure in congenital cardiac surgery.

8.7.1 Probing

When probing, all particles seek to remove themselves from the volume of space covered by the interaction instrument [135]. The fragment program calculates intersection and intersection response. In papers [4, 5] we used the following method; every fragment simply checked if the particle were inside a globally defined set of spheres. If this was the case the particle were projected to the sphere surface. We found that this was a major source of the flip-over problem discussed in section 5.3. A concrete result of the flip-over issue is that the visualization presented in chapter 9 depends on a well defined topology. In short, a group of three nodes span a space in which vertices of the surface are expressed. If a flip-over occurs with these three nodes, the space spanned will also flip and any vectors or vertices expressed in it will flip as well. That is, we will see the back-face instead of the front face resulting in black areas. The problem is that the repeated absolute positioning of particles onto the surface of the sphere could very easily create problems in which the implicit volume of the basic elements of figure 8.2 would be inverted or otherwise invalidated. One issue in the problem is that we do not take the connectivity of springs into account, and that we have no forces depending explicitly on the volume. We could introduce these volume-forces, but propose a more restrictive method to better guarantee that the before-mentioned problems do not occur. Our solution is to restrict the movement of particles that have a risk of invalidating the volume requirement. We define a polyhedron of the closest six neighbors, shrink this polyhedra by a factor and finally restrict the movement of particles to this small polyhedra. See figure 8.9. This restriction of movements is very conservative and we consequently only perform it on nodes within a certain distance to the probing instrument.

8.7.2 Grabbing

All particles have a state indicating whether they are grabbed or not. When the grab-button is pressed we perform an intersection check on all particles using a bounding sphere of the grabbing tool. If a particle is intersected the state of the particle is changed to *grabbed*. When particles are grabbed, their position will be set relative to the center of the grabbing device for as long as the grab-button is active. We have chosen to resolve the grab state of particles on the CPU. When the grab-button is pressed we do a read-back of the position texture *once*. The CPU knows which particles are grabbed and what their positions should be relative to the interaction device. This information is rendered back into the position texture after the integration, overwriting that calculated position, and before cycling of the PBuffers. When the grab-button is released the additional rendering is no longer necessary. Since the position texture is only read back at the

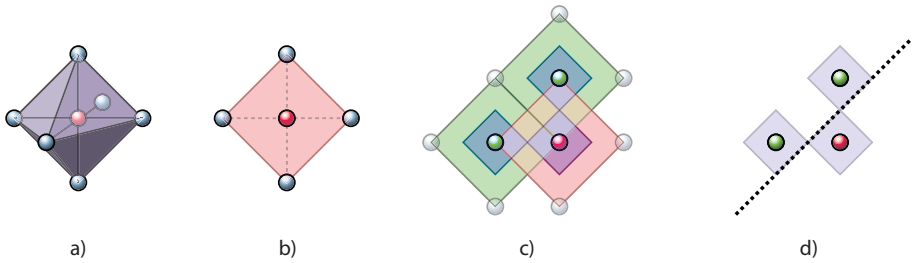


Figure 8.9: Restriction of movement of particles. a) for each particle (red) we check for intersection with each of the triangles spanned by the closest six (blue) neighbors. b) illustrated in 2D, the red particle seen in isolation is allowed to move within the red area defined by the four neighbors (in 2D). c) since all particles are updated simultaneously the green particles connected to the red particle may also move. We consequently restrict the movement of all particles to a more narrow volume (blue) than the original defined by neighboring particles. d) in this simplified 2D case, the original invariant of b) is fulfilled if both green particles and the red particles stay within their blue narrow volume, i.e.. do not cross the dotted line so that the volume is not well defined.

beginning of a grab, there is no noticeable slowdown in the simulation. The final result is that the grabbed particles move relative to the interaction device and neighboring particles will follow due to the springs connecting these particles. It is possible to handle grabbing entirely on the GPU. This leads to more fragment operations per simulation iteration however, and results in lower simulation rates.

For the simulation to behave realistically, a grab needs to include a relatively large set of particles. If a grab is made across an incision, the user will not be able to open up the incision with that grab. To alleviate that issue, we include only particles in the grab that can be accessed by following spring-connections to a depth corresponding to the radius of the grab-sphere.

8.7.3 Cutting

A cutting action defines an incision in the spring-particle connections, removing or altering connections so that there are no structural constraints across the incision [134]. The geometry should furthermore follow the incision as closely as possible. Since the surface faces are not directly represented in the fragment programs for the Spring-Mass computations, updating the surface geometry is done on the CPU with up-to-date nodal positions. Before each cutting procedure the position texture is read back to the CPU once. Collision detection can be done on the CPU or GPU [47] but collision response in the form of structural changes is handled on the CPU. When

Figure 8.10: For the implicit method we proposed a cutting scheme that rendered additional quads to exclude individual springs. This graph shows the relationship between the size of the cut as percentage of the whole geometry of 42.745 particles and the frame rate. The frame-rate is for rendering the pig heart morphology (figure 8.6) with four iterations of simulation per visualization on a GeForce6800 Ultra

using explicit connections, cutting is simply a transfer of CPU based cutting schemes [146] since we represent both particles and springs. Structural changes can be propagated to the GPU by simply writing into the position texture and connectivity texture. When using implicit connections we have no representation of springs and we cannot insert particles between other particles. A simple cutting scheme would erase particles and hereby the springs connecting the particle to its neighbors. This can be accomplished by writing an alpha value of zero into the fragments corresponding to the particles we wish to erase. This will mark the fragments as inactive particles. The incision would then be at least as wide as the length of two relaxed springs. We propose instead a method of cutting the implicit model that improves the granularity; we erase individual springs. To erase a spring we render a special fragment-sized quad at the position of the two associated fragments. The fragment program is unchanged, but we erase springs by giving the value of zero for those texture coordinates (out of the 18) related to erased springs. A zero texture coordinate results in reading the texture padding and consequently indicates that there is no connection. This approach of cutting results in a growing number of quads rendered proportional to the number of erased springs. In most simulations the number of quads will fortunately not grow excessively.

8.7.4 Interaction Results

Performance results for the implicit GPU cutting are reported in figure 8.10 showing the decline in performance when cuts are made in the simulation.

8.7.5 Discussion and Conclusion of Interaction

For this kind of surgical simulation the growing number of quads rendered (figure 8.10) to support incisions is a minor performance issue since using few and small incisions is an important characteristic of surgery.

The granularity of cuts in the implicit method is not a big problem for the incisions in the heart, since these are large compared to the detail of the Spring-Mass system. The interaction through magnetically tracked styluses gives 3D position and rotation but lacks any haptic feedback which would help the surgeons in judging forces used in the interaction. I refer to part II for a more use-specific discussion.

8.8 Discussion and Conclusion of GPU Spring-Mass

As can be seen from figure 8.1 and figure 8.7, the GPU clearly outperforms the CPU in computation of a Spring-Mass system. Comparing the implicit GPU method to the CPU implementation we have a speedup of up to a factor 30. The GPU achieves this computational speedup since we successfully expressed the Spring-Mass algorithm in computational resources available in the modern GPU. One aspect of this is the use of the very specialized fragment processor, but the biggest impact on performance is clearly that the type of computation presented here is very well suited to the cache-scheme of the GPU, as discussed in section 7.2. The GPU method with explicit connections runs at about half the speed of the implicit method, because of the double amount of texture-lookups. There is clearly a trade-off between speed and the generality of connectivity in the Spring-Mass system.

With the increased number of iterations available per second, we can achieve faster convergence when simulating physically based deformable geometry. We can iterate several times in the simulation loop before rendering the result to the screen, still in real-time. Furthermore we can achieve greater stability and precision of the numerical methods by using smaller time-steps than previously. Finally, the added computational power could also be used to process larger geometries with added degree of detail, enabling more elaborate surgical simulations.

If we consider a standard Spring-Mass implementation, there are many improvements that can accelerate the simulation on the CPU. These might, however, not be easily ported to the GPU. Hence, the presented speedup is not to be interpreted as a speedup compared to the fastest CPU implementation available. If we consider the LR-SpringMass model [2] for CPU acceleration presented in section 5.4, that method is not easily implemented on the GPU. In [2] we iterate through particles in a sequence determined by interaction and thereby accelerate the global convergence of the integration. On the GPU we have no control over the sequence of fragment processing. Likewise, iterating over all springs to perform relaxation poses a problem

because each fragment can only output one particle position, and we must work in a stream paradigm where input and output buffers cannot be the same.

In the past decade, GPU performance growth has exceeded that of the CPU [43]. This is expected to extrapolate well into the future. Hence the acceleration factor is expected to grow correspondingly.

8.9 Future Work

Future work in a technical perspective should include studies on how the GPU and CPU can be made to work more efficiently together. In the presented solution, the CPU is not utilized efficiently because the transfer of data from the GPU to the CPU becomes a bottleneck. It is simply too costly to let the CPU and GPU communicate beyond a trivial amount. With the PCI-Express standard for graphics cards the communication speed is expected to grow considerably and should be equally fast in both directions. We must then consider what kind of processing is suitable for the GPU and CPU, and allow them to work jointly on the problems at hand.

In cases where the spring-mass model is not considered adequate, other physically based models of deformation could be ported to the GPU following the principles of this progress report. The explicit FEM model from section 5.1.5 could be implemented as a dynamic simulation in much the same way as the spring mass model, but requiring a larger number of neighbor lookups as well as neighbor dependent stiffness coefficients.

It would also be very interesting to look into the possibilities of automatic level of detail in the spring-mass simulation as well as the visualization using the hardware accelerated linear interpolation of texture lookups. Such a representation would probably include some hierarchical representation of vertices and nodes on the GPU, and could possibly also be used as an acceleration structure for intersection tests without the transfer of data to the CPU.

As a next step in the range of interaction modalities, future research will investigate how the GPU implementation of the spring-mass algorithm can support suturing. The difficulties herein are especially evident in the implicit GPU implementation because connectivity is implicitly defined and suturing seeks to explicitly connect parts of the tissue.

Chapter 9

Decoupling Visualization and Simulation

This chapter presents the visualization techniques developed to show a detailed model of the virtual heart. The chapter is based on the paper [6] with the addition of some source code and better quality figures.

9.1 Introduction

Recently GPUs have become programmable to a degree that makes them useful for general-purpose computation [43]. Specifically, the computation of physically based systems has been successfully implemented [91, 5, 23]. The GPU is designed to work with textures, and the discretization of the physical equations often takes advantage of this by restricting nodes to a regular grid. Consequently, attention must be put into visualizing the results in detail, smoothly and continuously.

In this chapter we focus on the problem of visualizing a deformable surface defined by a Spring-Mass system solved on the GPU. In chapter 8 we presented methods for GPU accelerated computation of Spring-Mass based elastic deformation. In particular, a grid-based arrangement of mass-points gave a speedup of a factor of 20 to 30 compared to a similar CPU implementation.

We present three generally applicable methods for the visualization of a surface deformed by a set of nodal positions. The methods are not directly dependent on the calculation of the physical system and can be applied to other physical systems or methods of animation. First, a simple one-to-one mapping (with approximate normals) is defined between visualized vertices and mass-points. Secondly, we present two methods of deforming and visualizing a detailed surface based on the deformation of a relatively low number of nodes. The two methods both define a dynamically changing orthonormal vector basis for all nodes on the surface of the simulated volu-

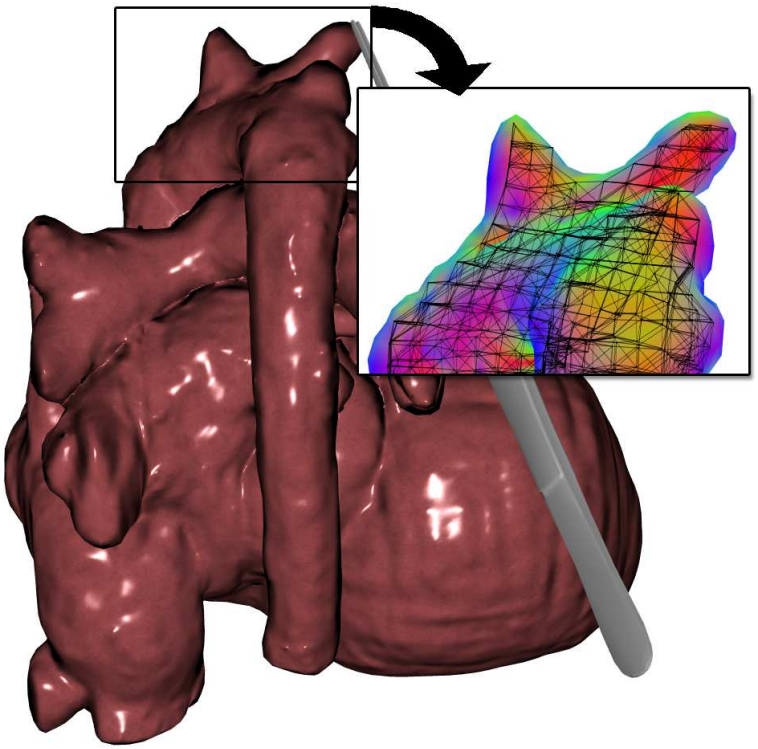
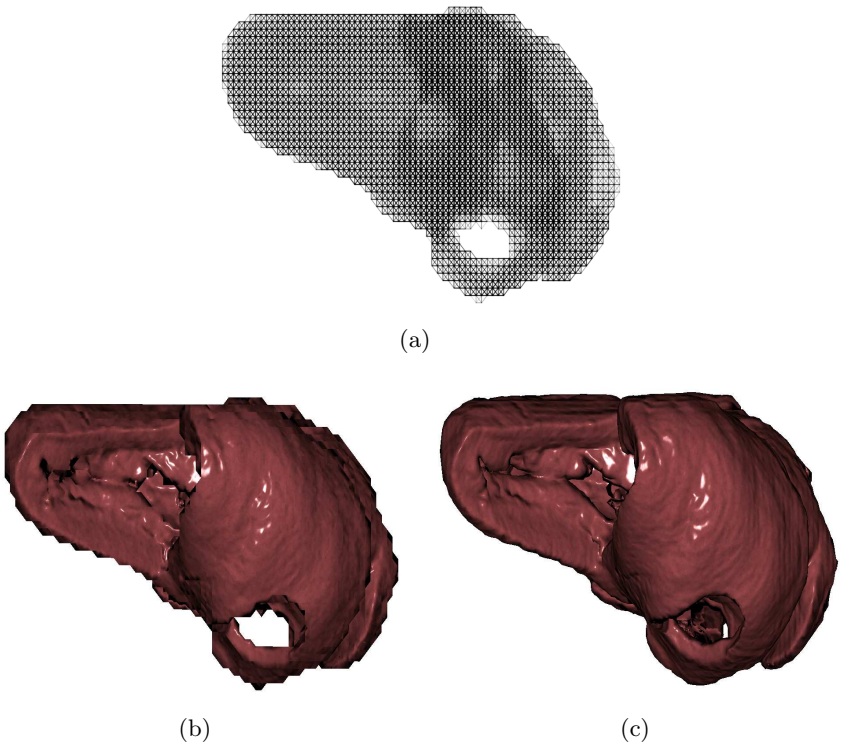


Figure 9.1: A physical simulation on the GPU calculates the deformation of a grid of 20.000 particles representing the shape of a heart. A heart surface with 50.000 faces is mapped to the grid. As a result the highly detailed surface can be deformed in real-time based on the deformation of the simpler grid. In the close-up, the grid is shown on top of shading illustrating the dynamically calculated normals.



(a)

(b) (c)

Figure 9.2: A close-up of the grid-based Spring-Mass simulation showing the springs of the physics system in b). The visualization is done with one-to-one mapping between nodes and vertices in a) and offsets from nodes to vertices in c). An orthogonal projection matrix has been used to clearly show the grid.

metric grid. The surface-vertices to be visualized are expressed in this basis and consequently reflect the deformation of the simulated nodes. Tangent space bases can also be expressed in the dynamically calculated bases and enable normal-mapping of the highly detailed deforming surface geometry. Common to all the methods presented is that we construct a detailed surface mesh and render this mesh through a vertex program that defines the mapping between vertices on the surface and nodes in the simulation.

The overall goal of our project is to simulate surgery on children with congenital heart disease, supporting deformation as well as cutting of heart-tissue. The detailed visualization is needed for the surgeon to accurately recognize features of the heart. At the same time the physical system must have a resolution that allows for real-time deformation.

9.2 Previous Work

Our simulated system can be regarded as a low-resolution geometry, based on which we express and animate a highly-detailed geometry. Classical approaches to this problem on the CPU include bump mapping and displacement mapping. Bump mapping [21] perturbs the surface normals at a per texel level, affecting shading as if the geometry had bumps. Displacement mapping [49] tessellates each triangle and displaces the newly generated vertices along the normal according to an amount read from a height-map. Displacement mapping allows for the animation of a low resolution mesh of control points that deform the high resolution geometry accordingly.

Bump mapping can easily be implemented in fragment programs on the GPU, but does not correct for the jagged appearance along the contour of the mesh. While bump mapping does not in itself solve the problem of the jagged contours, it can be combined with the presented work to visualize small-scale surface details. GPU implementations of displacement-mapping based on textures have been made possible through the texture lookup instruction in vertex programs introduced in shader model 3.0. Simple one-dimensional perturbations along the z axis of a grid of vertices have been used to visualize a dynamic height-map [112], but the method cannot directly do displacement mapping for an arbitrary 3D mesh.

Because of the large amount of fragment processing power available, several authors have introduced the idea of computing per-pixel offsets from actual geometry to a surface described by a height-map [94, 103]. These techniques generally use a height-map with ray marching which is computationally expensive. Complex 3D meshes including curvature have furthermore been problematic because the current ray-marching techniques are not aware of curvature, nor the mapping of height-maps to faces. Techniques such a [189] pre-processing the intersection-tests to simplify the calculations on the GPU and using lookups in the pre-processed data exist, but these

are at the cost of extensive memory usage. For our application the memory usage is too high. The technique furthermore requires the geometry of the simulated system to fully enclose the surface to be visualized. That requirement is too restrictive for a grid based simulation visualizing incisions that are smaller than the grid resolution.

9.3 Methods

In this chapter we use the term *vertex* to describe a vertex in the surface mesh exclusively, and the terms *node* and *nodal* to describe a position in the less detailed set of control points. When we wish to emphasize that a node-position is based on a physical simulation we use the term *particle*. We start out by briefly reviewing our GPU Spring-Mass simulation in section 9.3.1. In section 9.3.2 we present a simple one-to-one mapping between nodes and vertices and in section 9.3.3 we present two more general mappings from a set of vertices to a less detailed set of nodes.

9.3.1 Grid Based Calculation of Deformation on the GPU

We briefly review the grid-based layout of particles (also called the method of implicit connections) for the calculation of deformation on the GPU as presented in chapter 8. The method is based on a layout of particle-positions in a regular three-dimensional grid. Each particle is connected in a fixed pattern to the 18 nearest neighbors. The grid is mapped to a 2D texture for fragment processing, and the texture containing the current set of particle positions is called the position-texture. In fragment programs we calculate the forces influencing the particles and numerically integrate to obtain positions. The basic linear spring force g_i with rest length l_{ij} for a node i with position p_i and neighbor positions D is calculated as:

$$\vec{g_i} = \sum_{j \in D} \frac{1}{2} k_{ij} \left(l_{ij} - \|\mathbf{p}_i - \mathbf{p}_j\| \right) \frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|} \tag{9.1}$$

This expression can be optimized for fragment processing if we assume that the spring coefficients k_{ij} are all equal to the constant k and observe that springs arranged in the grid have rest-lengths 1 and $\sqrt{2}$:

$$\begin{aligned} \vec{g_i} = \frac{1}{2} k \Bigg\{ & \sum_{j \in D_1} \frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|} - \sum_{j \in D_1} (\mathbf{p}_i - \mathbf{p}_j) + \\ & \sqrt{2} \sum_{j \in D_2} \frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|} - \sum_{j \in D_2} (\mathbf{p}_i - \mathbf{p}_j) \Bigg\} \end{aligned} \tag{9.2}$$

D_1 is the set of neighbors with distance 1 and D_2 is the set of neighbors with distance $\sqrt{2}$. Besides saving instructions, this expression includes a sum of unit-vectors, which is used to find approximate normals after deformation.

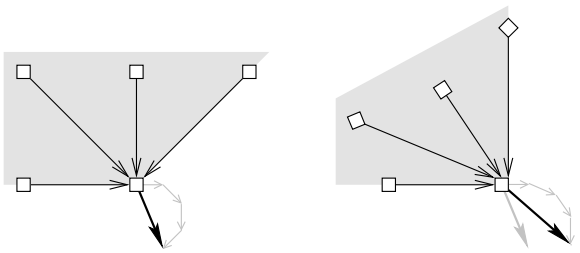


Figure 9.3: A 2D example of a normal calculated on the basis of unit-vectors from neighbors. a) the original geometry. b) the deformed geometry. The normalized vectors (grey) are added to form the approximate normal (black).

To visualize the set of nodes in real-time based on the deformation of nodes we cannot simply read back the node positions to the CPU and render vertex positions accordingly, as this would be a major performance bottleneck. Instead we utilize a new feature in the shader model 3.0 design; texture lookups in vertex programs. We use vertex programs to express the calculations involved in mapping vertices to nodes. Through vertex texture fetches we access only nodes that are part of the surface of the simulation-mesh. The geometry specified to the 3D API to visualize the current simulation-step is a static mesh, allowing caching on the graphics card, while the vertex-program can retrieve up-to-date nodal positions from the position-texture.

9.3.2 One-to-One Mapping and Approximating Normals

A limited visualization method is presented in this subsection to motivate the later generalization. This method was presented in the papers on Spring-Mass computation on the GPU [4, 5] and repeated in the paper on which this chapter is based as a motivation for the decoupling of simulation and visualization. The method is based on a simple one-to-one mapping between the position of a vertex and a surface node, simply transferring the position of the node to the corresponding vertex-position. To render the model we initially set up a display-list, which renders the surface geometry at its original rest position. We pass one texture coordinate per vertex to provide the coordinates of the corresponding node in the position texture. The vertex program then fetches the most recent node position, performs basic transformation and outputs the deformed position for further processing in the graphics pipeline.

An important issue in this visualization is the calculation of surface normals used for shading of the surface. In a conventional CPU application, one would often approximate vertex normals by averaging the adjacent face normals. Face normals are found by reading the positions of nodes making up the face and calculating the normal of the plane. Implemented on the

Algorithm 2 Pack and Unpack Cg programs using the 23 bits mantissa divided into 7,8 and 8 bit for x,y and z components.

```
float pack(float3 a) // pack into 23 bits mantissa 7,8,8
{
    a = normalize(a);
    float first8Bit = floor(((a.x+1)*0.5) * 255);
    float second8Bit = floor(((a.y+1)*0.5) * 255);
    float third7Bit = floor(((a.z+1)*0.5) * 127);
    //2^8 and (2^8 * 2^8)
    return first8Bit + second8Bit*256 + third7Bit*65536;
}
float3 unpack(float a)
{

    float first8Bit; float second8Bit; float third7Bit;
    third7Bit = floor(a/65536.0);
    a = a - third7Bit*65536;
    second8Bit = floor(a/256.0);
    a = a - second8Bit*256; first8Bit = floor(a);
    float3 ret = float3(first8Bit/255,
                                  second8Bit/255,
                                  third7Bit/127);
    return (ret*2.0)-1;
}
```

GPU this would result in an excessive amount of texture lookups. It is not an option either to read back the position-texture each frame due to performance reasons; the calculation of normals must take place on the GPU. We approximate surface normals by the normalized sum of the unit vectors pointing from neighbors to the particle in question, an example is seen in figure 9.3:

$$\vec{N_a}(i) = normalize(\sum_{j \in D} normalize(\mathbf{p}_i - \mathbf{p}_j)) \tag{9.3}$$

Since we already computed the sum of the unit vectors in the force calculations in equation (9.2), we only need to normalize this vector and save it. This approximation gives us well behaving normals almost for free. We pack the 3-tuple normal into the 32 bit alpha channel of the fragment representing the particle-position, see algorithm 2.

The normal is read and unpacked by the visualization vertex program and send to a fragment program for per pixel lightning. The one-to-one

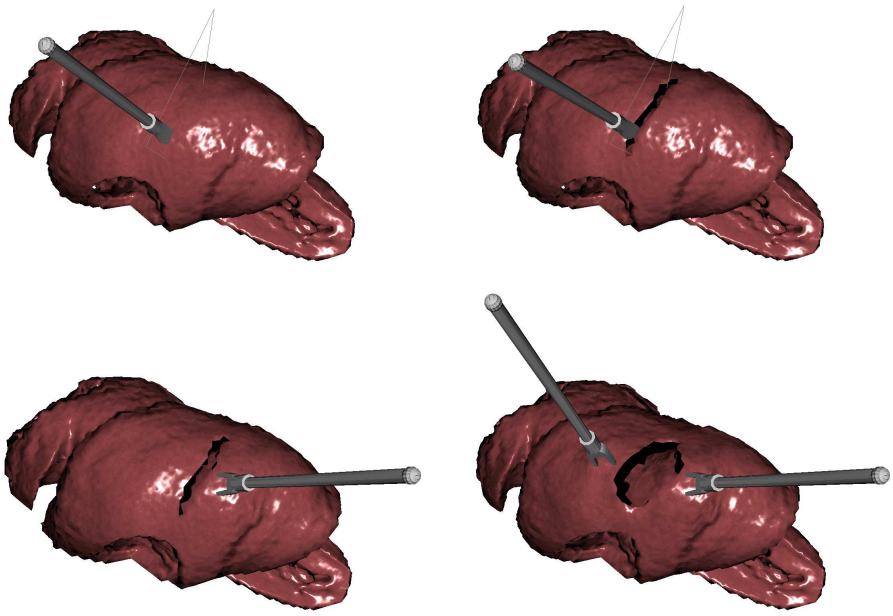


Figure 9.4: A series of screen-shoots showing cutting (section 8.7.3) in combination with the one-to-one mapping. Notice the jagged and spring-wide incisions.

mapping has some severe limitations since it is based directly on the set of simulated nodes. Basically the visualized result has a very jagged look, see figure 9.2 a) and b). Furthermore, cutting the simulation grid as proposed in section 8.7.3 in combination with the one-to-one mapping as proposed in this section results in very large and jagged incisions, see figure 9.4. These limitation are removed by the remaining work in this chapter, see figure 9.1 and figure 9.2 c).

9.3.3 Mapping using the Triangle Basis

In the previous section we presented a one-to-one mapping between a surface vertex and a node with a direct transfer of nodal positions to vertex positions. For shading, an approximate normal was found based on the force calculation. The technique presented in this section enables a "many-to-one" mapping from vertices to nodes through an offset vector. The goal of this technique is to deform highly detailed geometry based on the deformation of less detailed geometry. The less detailed geometry can be as simple as a set of interconnected points.

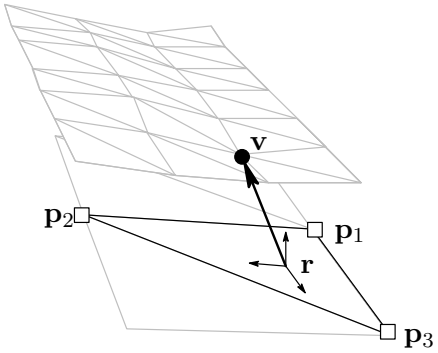


Figure 9.5: a) For a high-resolution set of vertices and a low-resolution set of nodes we define for each triangle $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ and corresponding vertex $\mathbf{v}$ the reference point $\mathbf{r}$ and offset vector $\vec{o} = \mathbf{v} - \mathbf{r}$.

Defining an offset in the triangle basis

For each vertex $\mathbf{v}$ in the highly detailed surface we associate a triangle of three nodal positions $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$, which control the position of the vertex, see figure 9.5. The triangle defines a space within which we will represent vertex positions and vectors from the highly detailed model. We need the positions of the vertices to be based on the location, rotation and scaling of the triangle bases. First we define a reference point $\mathbf{r}$ for each vertex. The reference point is chosen as a projection of the vertex onto the triangle base. The reference point will be represented by weights (w_1, w_2, w_3) of the nodal positions:

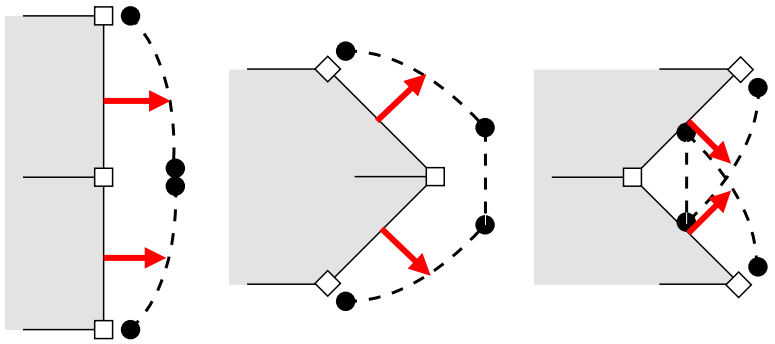
$$\begin{aligned} \sum_{i=1}^{3} \mathbf{p}_i w_i &= \mathbf{r} \\ \sum_{i=1}^{3} w_i &= 1 \end{aligned} \tag{9.4}$$

This definition allows the reference points of the vertices to adjust as the triangle scales.

The vertex $\mathbf{v}$ can be expressed as an offset vector $\vec{o}$ from the reference point $\mathbf{r}$:

$$\mathbf{v} = \mathbf{r} + \vec{o}. \tag{9.5}$$

This vector is expressed in object space. As a next step, we express the offset vector in a vector basis formed by the location of nodes in the associated triangle. Ideally, the offset vector should be affected by both rotation and scaling of the triangle. We simplify the definition of the offset however, taking into account only the rotation of the associated triangle. Depending on the chosen projection of vertices onto triangles, the offset vector will be



a) b) c)

Figure 9.6: Visual artifacts can occur when vertices are rotated based on per triangle information only. For simplicity we exemplify in 2D. a) is the original configuration of nodes (boxes) and vertices (dotted line and spheres). b) illustrates the problem of two vertices that are very close in the original configuration but are disproportionally far away from each other when the deformation occurs. c) illustrates how the vertices can intersect the mesh.

close to the normal of the triangle, in which case scaling in the triangle plane has little or no effect. Per triangle we define the orthonormal basis ($\vec{T}$, $\vec{N}$, $\vec{B}$) based on ($\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$) as the *triangle basis*:

$$\begin{aligned} \vec{B}_t &= normalize(\mathbf{p}_3 - \mathbf{p}_1) \\ \vec{T} &= normalize(\mathbf{p}_2 - \mathbf{p}_1) \\ \vec{N} &= \vec{T} \times \vec{B}_t \\ \vec{B} &= \vec{N} \times \vec{T} \end{aligned} \tag{9.6}$$

The offset vector $\vec{o^t}$ in the triangle basis is calculated as:

$$\vec{o^t} = \left(\vec{o} \cdot \vec{T}, \vec{o} \cdot \vec{N}, \vec{o} \cdot \vec{B}\right) \tag{9.7}$$

Curvature across triangles

In the previous section the triangle basis was defined as being constant across each triangle. Consequently, in some cases deforming a mesh of triangles can result in visual artifacts between vertices that are associated to different triangles, see figure 9.6 and figure 9.8 a). The deformation of vertices associated to different triangles is not in any way inter-dependent even though they might be very close in the original configuration in world space. A better solution is to interpolate the orthonormal bases across the triangles, see figure 9.7. This means that the orthonormal basis should be defined at each node and depend on the orientation of all incident triangles. This approximates the curvature of the simulated shape more closely, not only

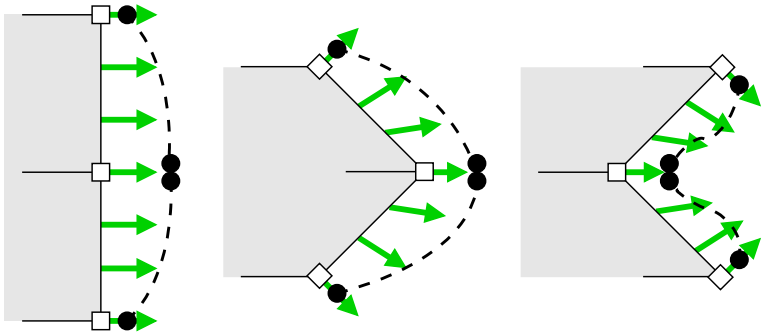


Figure 9.7: A correct curvature when vertices are rotated based on interpolated information. For simplicity we exemplify in 2D. The configuration of nodes are as in figure 9.6.

the orientation of each triangle in isolation. The difference is comparable to the difference between flat and Gouraud shading. A naive implementation would require a large amount of expensive vertex texture fetches, calculating the triangle bases of all surrounding triangles. Instead we address this issue by remembering that we already have approximate normals per node (section 9.3.2). We do not have a tangent and bi-tangent per triangle node though. Our solution is to assume that the approximate normal $\vec{N_a}(i)$ and the normal $\vec{N}$ (equation (9.6)) are close enough that we can use the per-triangle tangent $\vec{T}$ and bi-tangent $\vec{B}$ to construct a per-vertex tangent and bi-tangent based on the approximate normal. To compute the interpolated normal for a vertex associated to a given triangle of nodes $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ we use the weights (w_1, w_2, w_3). The interpolated triangle basis $(\vec{T_c}, \vec{N_c}, \vec{B_c})$ can be expressed as:

$$\begin{aligned} \vec{N_c} &= \textstyle\sum_{i=1}^{3} w_i \vec{N_a}(i) \\ \vec{B_c} &= \vec{N_c} \times \vec{T} \\ \vec{T_c} &= \vec{B_c} \times \vec{N_c} \end{aligned} \tag{9.8}$$

Any vertex on the highly detailed geometry can now be expressed in the interpolated triangle basis as in equation (9.7).

Implementation

The entire highly detailed mesh is sent to the GPU for visualization through OpenGL as vertices arranged in a mesh. For all vertices we pre-calculate the nodal weights (to express the reference point) and the offset in the triangle basis on the CPU *once*. This information is given as vertex attributes to the vertex program. The nodal positions reside in texture memory; consequently we give three texture-coordinates as additional per-vertex attributes

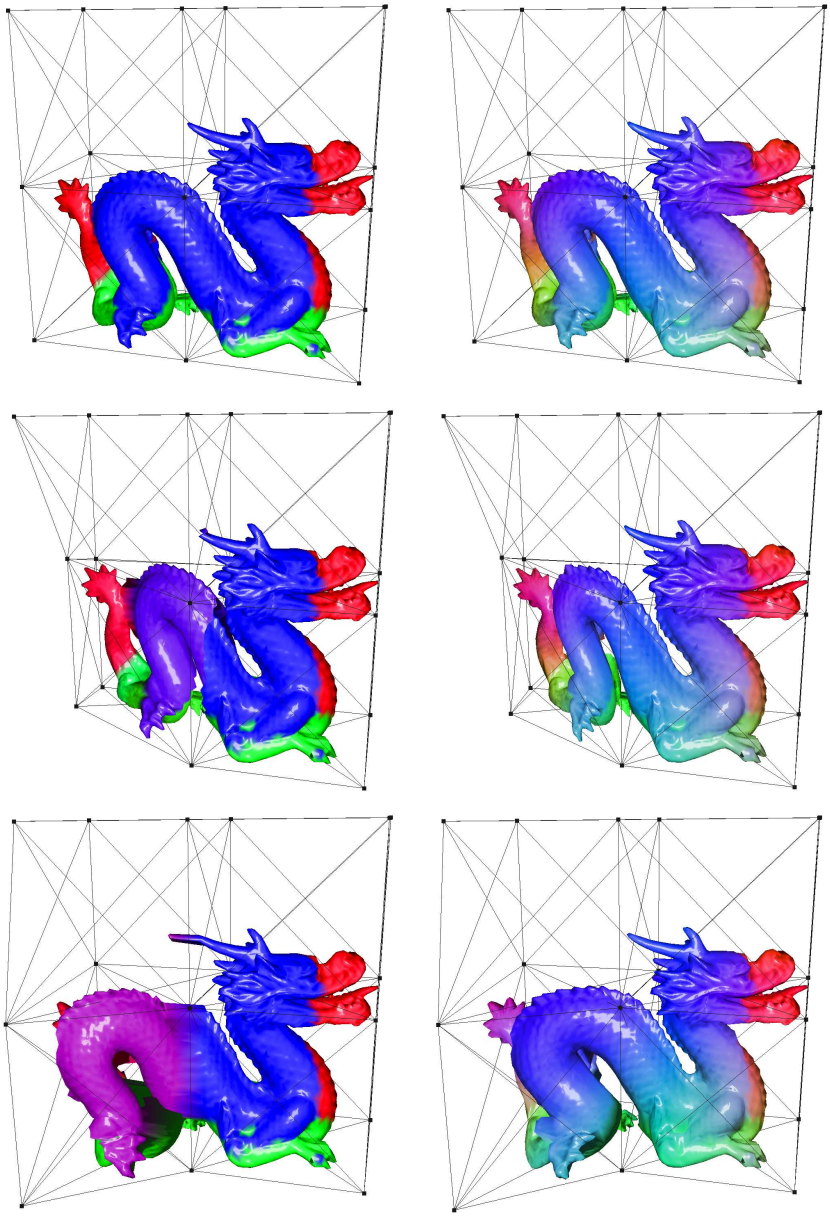


Figure 9.8: Visualization of the deformation of a dragon [175]. The left column of images shows the deformation without correction for curvature and the right column of images shows the deformation with correction for curvature. The top row of images are without any deformation while the middle row and bottom row show a push and pull of a node respectively. The shading of the dragon is a combination of normal-mapping and the color representing the dynamic normal.

to resolve the current positions $(\mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3)$.

In the vertex program three texture lookups are used to retrieve the current nodal positions. Finding the vertex position in the vertex program is equivalent to the construction of weights (equation (9.4)), triangle-basis (equation (9.6) or (9.8)), and offset-vector (equation (9.7)) on the CPU. On the GPU these calculations are done "backwards" compared to the CPU; resulting in the current vertex point $\mathbf{v}'$ based on the configuration of the current positions $(\mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3)$:

$$
\begin{aligned}
&lookup(\mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3) \\
&\mathbf{r}' = \textstyle\sum_i^3 \mathbf{p}'_i w_i \\
\\
&\vec{B}_t = normalize(\mathbf{p}'_3 - \mathbf{p}'_1) \\
&\vec{T}' = normalize(\mathbf{p}'_2 - \mathbf{p}'_1) \\
&\vec{N}' = \vec{T}' \times \vec{B}_t \\
&\vec{B}' = \vec{N}' \times \vec{T}' \\
\\
&unpack(\vec{N'_a}(1), \vec{N'_a}(2), \vec{N'_a}(3)) \\
&\vec{N}'_c = \textstyle\sum_{i=1}^3 w_i \vec{N'_a}(i) \\
&\vec{B}'_c = \vec{N}'_c \times \vec{T}' \\
&\vec{T}'_c = \vec{B}'_c \times \vec{N}'_c \\
\\
&\vec{o'} = \begin{cases} (\vec{o^t_x} \cdot \vec{T}'_c, \vec{o^t_y} \cdot \vec{N}'_c, \vec{o^t_z} \cdot \vec{B}'_c) & \text{if interpolated} \\ (\vec{o^t_x} \cdot \vec{T}', \vec{o^t_y} \cdot \vec{N}', \vec{o^t_z} \cdot \vec{B}') & \text{if constant} \end{cases} \\
\\
&\mathbf{v}' = \vec{o'} + \mathbf{r}'
\end{aligned}
\tag{9.9}
$$

Shading

Simple Gouraud shading of the highly detailed mesh depends on a per vertex normal. An optimal normal for shading is possibly different from the normal of the triangle basis. Consequently a pre-calculated normal for the initial mesh configuration is expressed in the triangle basis hereby reflecting the deformation of the nodes. Tangent-space based shading such as normal mapping or parallax mapping is possible if we express vectors defining the tangent space in the triangle basis. These additional vectors are given as per-vertex attributes.

9.3.4 Results

The fragment and vertex programs have been implemented on a GeForce 6800 Ultra graphics card. For shading of all the illustrated models we have used normal mapping.

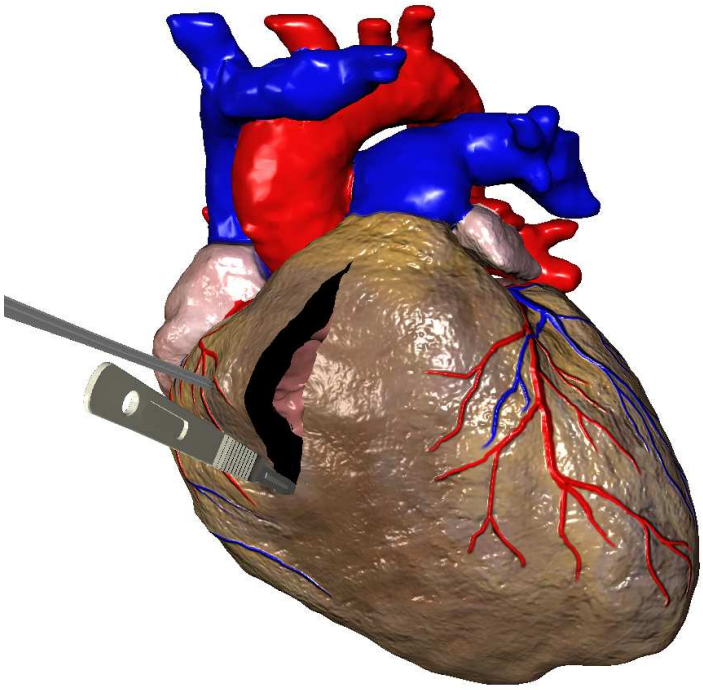


Figure 9.9: This figure shows the result of making an incision into a high-resolution surface controlled by a lower resolution simulation grid.

The heart model in figure 9.1 has approximately 50.000 faces. The simulation grid consists of 20.000 nodes. The triangle-basis based visualization with correction for curvature can be visualized at 150 fps. Including 15 simulation steps per visualized frame results in 25 fps. This amounts to 6.67 milliseconds per visualization step and 2.22 milliseconds per simulation step with our current implementation. Without curvature correction each visualization of a frame takes 5.26 milliseconds (190 fps).

As a test for the mapping of a high resolution surface onto a very low resolution set of nodes we used the dragon model [175] consisting of 9.971 faces controlled by just 18 nodes. In figure 9.8 the dragon is visualized with the two variations of triangle-basis based mapping from section 9.3.3.

The visual difference between the one-to-one mapping of node positions to vertex positions and the triangle-basis based mapping with offsets to vertex positions can be inspected in figure 9.2. Comparing the incisions in figure 9.9 using the triangle-basis based mapping and figure 9.4 using the one-to-one mapping a clear difference in resolution of the cut is evident. The size of the cut into the simulation grid is of comparable size in the two models, but is not visible in figure 9.9.

9.3.5 Discussion and Conclusion

We use the GPU hardware for both visualization and calculation of deformation. It is important to realize that the speed of both simulation *and* the visualization of the Spring-Mass system is faster than a conventional CPU implementation. The GPU implementation of the Spring-Mass system without visualization is 20 to 30 times faster than a similar CPU implementation [4, 6, 5]. Secondly, the GPU can cache the highly detailed surface for visualization in the case of the GPU based simulation because the definition of vertex attributes does not change. A CPU implementation of a Spring-Mass system cannot use this technique because new vertex positions are calculated every frame, and these need to be sent to the GPU.

The visualization methods presented have various tradeoffs that must be considered in choosing which visualization technique to use. Using triangle-basis based mapping, as presented in section 9.3.3, allows us to decouple visualization and simulation. This allows e.g. visualization of higher resolution or smoother appearance than the set of simulation nodes represents, see figure 9.2. If this is not necessary, the simpler one-to-one mapping can be used instead. If tangent-space based shading is to be used, e.g. for normal mapping, the one-to-one mapping does no directly support this since only an approximate normal is available. In such cases the triangle-basis based mapping is useful even if an offset vector is not needed.

The triangle-basis based mapping can be simplified somewhat if we assume that the offset is always along the dynamically calculated normal. In that case we do not need to create the triangle basis to express the offset vector. We still need to perform three vertex texture-lookups to interpolate the approximate normals though. The construction of the triangle basis is still necessary if we wish express other vectors in the deforming space defined by the triangle - e.g. to use normal mapping.

The approximate normal $\vec{N_a}$ (see equation (9.3)) depends on certain properties of the nodes. The method is most effective if the nodes are arranged in a regular grid, since this guarantees that the neighbors are evenly distributed for calculation of the approximate normal. Very important, there must exist a neighbor that adds to the approximate normal in the direction of the desired surface normal. A thin cloth simulation would not provide correct approximate normals since nodes are only connected in two dimensions. There is no additional neighbor in the third dimension to force the approximate normals to point outwards from the cloth. In the surgical simulator this means that we cannot use the current approximate normals at parts of the organ with just one node depth. The triangle-basis based mapping that corrects for curvature through the approximate normals naturally has the same dependence on the property of nodes. The triangle-basis based mapping with a constant triangle basis across triangles can be used instead to visualize the geometry in these cases. The heart model in figure 9.1 is

rendered with the triangle-basis based mapping without correction for curvature because parts of the blood-vessels are simulated as "one node thin" sheets.

If the offset vector in triangle-basis based mapping is sufficiently short compared to the relative resolution of the set of nodes or if the reference point is close to a node, we can leave out correction for curvature without compromising the visual appearance. In the case of the heart presented in figure 9.1 the correction for curvature can be left out, but in the case of the dragon in figure 9.8 the difference is very evident.

9.3.6 Future Work

It would be interesting to look into the possibilities of an automatic level of detail (LOD) in the Spring-Mass simulation coupled with the triangle-basis based visualization. This would enable a seamless change between resolutions of simulation because each level of the LOD can be mapped to the same high-resolution surface. Dynamic LOD could be implemented by exploiting hardware accelerated linear interpolation of texture lookups. Such a representation would probably include some hierarchical representation of vertices and nodes on the GPU, and could possibly also be used as an acceleration structure for intersection tests without the transfer of data to the CPU.

In cases when the triangle-basis based mapping with correction for curvature is the preferred method of visualization but the approximate normal is not well defined for all nodes, the two methods could be combined. In pre-processing on the CPU we could identify the problematic nodes and use the mapping without correction for curvature for these vertices only. The choice of method could be included as a per-vertex attribute to let the vertex program choose between the two methods on a per-vertex basis.

In a normal utilization of the graphics pipeline for drawing geometry, fragment processing is conserved through depth-buffer culling. In future work we will look into removing part of the potentially large amounts of computation done for *vertices* that are hidden by other geometry. In vertex programs we have no equivalent to depth-buffer culling of fragments though, since the information is not available until we have calculated the vertex positions. Through utilization of the low-resolution set of nodes and approximate normals, some information is available though.

In future work of the visualization we are going to look at the possibility of simulating 2d image modalities such as x-ray, CT and MRI. The visualization of such image modalities will allow us to evaluate the surgical simulator by comparing post-operative scans with visualization of the simulated procedure in the same image modality as the scan. The visualization could also be used to train in the field of interventional radiology [52].

Chapter 10

Haptic Feedback

The following chapter is based on the paper [9] and describes our haptic interaction. Compared to the original paper [9], section 10.3 has been added explaining how we create smooth haptic interaction from discontinuous simulation-data.

10.1 Introduction

Surgical simulators have traditionally been implemented on the CPU and many algorithms have been proposed in order to calculate realistically looking soft-tissue deformations in real-time. Haptic feedback is often used to increase the realism of user interactions. This provides an additional challenge as force feedback must be provided at least at 500 Hz to feel smooth. To achieve such an update rate from a simulation running at a much lower frequency, extrapolation schemes have been developed, e.g. [125, 155]. Overall, speed has been a major concern as many time-consuming tasks all had to be handled on the CPU. Motivated by this issue, it was shown in chapter 8 that a twenty to thirty fold acceleration of a Spring-Mass based surgical simulation could be achieved when moving computations from the CPU to the GPU (i.e. the graphics card). This allowed real-time surgical simulation on very complex organs, such as the heart, for the first time. Calculating tissue deformations on the GPU does however expose some previously unaddressed problems on how to resolve haptic interaction. Since communication with the haptic devices must be handled on the CPU, synchronization and data transfer between the GPU and the CPU is necessary. Unfortunately this is a relatively slow operation. Hence, we must carefully design the communication scheme to avoid new performance bottlenecks.

In this chapter we describe and evaluate an efficient method of haptic interaction with the GPU based surgical simulator. Haptic feedback is provided in response to collisions between instruments and tissue. The overall design criterion is to allow efficient, smooth, two-handed haptic interaction

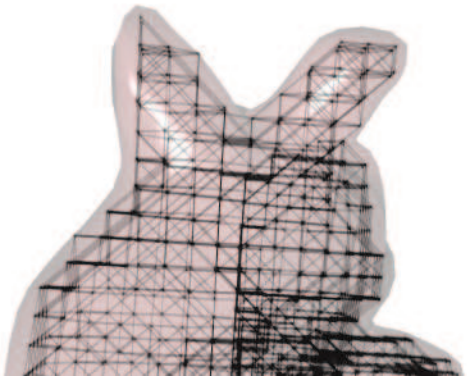


Figure 10.1: Particles in the Spring-Mass system are connected in a regular, three-dimensional grid (black). Each particle is allowed to move, but constrained by the springs to neighboring particles.

and easy balancing of the workload between simulation, visualization, and delivery of force feedback.

10.2 Materials and Methods

10.2.1 Hardware and Software Platforms

The hardware platform used to evaluate the simulation was a personal computer running Windows XP on an AMD FX-55 CPU, and 2 GB of memory. The graphics bus was PCI Express x 16. The proposed algorithms were tested on three different graphics cards, a GeForce 6800 Ultra, a Quadro FX 4400, and a GeForce 7800 GTX, all from Nvidia. Two Phantom Omnis (Sensable Technologies) were used to achieve haptic interaction by low-level access through the accompanying OpenHaptics Toolkit. All programming was done in C++ using OpenGL and Cg with some manual modifications of the compiled vertex and fragment programs. The cardiac model used throughout this chapter was obtained from three-dimensional MRI [173] using the segmentation algorithm described in [174]. The marching cubes algorithm was used for all surface reconstructions. Throughout this chapter we will consider a Spring-Mass simulation of 20.270 particles visualized by a surface of 137.490 faces.

10.2.2 Spring-Mass Simulation on the GPU

This section provides a short description of our GPU based simulator, since some terminology from its implementation is necessary to explain the extension with haptic interaction. First we discretize the volume of the concerned organ, e.g. the heart muscle mass, into a set of particles arranged in a regular

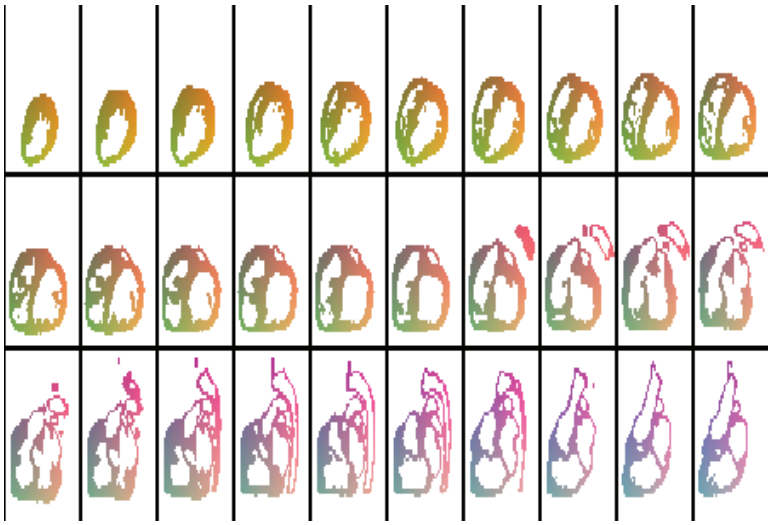


Figure 10.2: Excerpt of the position-texture. Each particle in the three-dimensional grid (Figure 2) is mapped to a unique texel (non-white). White texels correspond to void simulation particles outside the tissue.

three-dimensional grid (See figure 10.1). Each particle is connected in a fixed pattern to it's 18 nearest neighbors. The grid is mapped to a 2D-texture such that each particle is represented by a single texel (See figure 10.2). We name this texture the position-texture. Conceptually, parallel computation of the Spring-Mass system is invoked by rendering a single quad covering the entire position-texture. Processing of the white (void) particles is avoided by a depth test. A fragment program computes the forces that influence each particle due to its spring connections and the Spring-Mass differential equation is solved by numerical integration to obtain updated particle positions. We refer to one pass of these calculations as a simulation-step in the remaining paper. A surface is constructed from the boundary nodes in the Spring-Mass system and used for visualization. During visualization, a vertex program performs texture lookups in the position-texture to obtain the most recent particle positions. This allows us to use a static display list of the initial surface to also render it deformed. Rendering this surface is subsequently referred to as a visualization-step. In chapter 9 a mapping that decouples the visualized mesh from the physical simulation was presented. This allows for arbitrarily detailed surface meshes to be deformed by an underlying physical simulation (Figure 10.1).

10.2.3 Probing and Grabbing

The probing gesture (i.e. touching the tissue with an instrument) was realized entirely on the GPU by extending the fragment program responsible

for the simulation-step. Prior to writing the final position to the position-texture, each fragment determines if the corresponding particle has moved inside the bounding ellipsoid of an instrument. In that case the particle is projected to the boundary of the ellipsoid before updating the position-texture.

The grabbing gesture was designed to make use of both the CPU and the GPU. At the beginning of the gesture we read back the position-texture to the CPU once to identify particles inside the instrument's bounding volume and store pointers to these. For the duration of the grabbing gesture a dedicated fragment program writes the absolute positions of the grabbed particles into the position-texture based on the current position of the instrument.

10.2.4 Haptic Feedback

Communication with the haptic device is handled by the haptic thread on the CPU. We are consequently required to continuously read back data from the GPU to the CPU in order to provide haptic feedback. We could transfer the entire position texture and let the CPU compute the relevant forces from the current simulation state, however, it is much cheaper to compute these forces on the GPU and read back only the result. In the following we describe how to achieve this for the grabbing and probing gestures:

During a grabbing gesture the CPU holds a list of the grabbed particles and corresponding coordinates in the position-texture as described in section 10.2.3. We create an off-screen *force-buffer* the size of this list and for each particle a dedicated fragment program is run. This fragment program looks up the position of all neighbors to the current particle in the position-texture and calculates the force stored in each spring connection. The individual spring forces are summed to find the overall force vector affecting the particle. This vector is stored in the force-buffer at the position corresponding to the processed particle. Finally the force-buffer is read back to the CPU and passed to the haptic thread. If two hands are active both sets of corresponding forces are returned in a single read-back. This is an optimization as each read-back implies a relatively costly synchronization between the GPU and the CPU.

During a probing gesture the situation is more difficult since the set of particles contributing to the force feedback is no longer constant during the gesture, but changes in each frame depending on the position of the instrument. We can re-use the force-buffer approach however, if we extend it with a fast method to pick the particles that collide with the instrument. Note that instruments can only collide with surface particles. As a fast picking algorithm, we propose to render a simulation-surface into an off-screen picking-buffer. A simulation-surface is a mesh that connects the boundary particles in the Spring-Mass simulation. It is not necessarily identical to

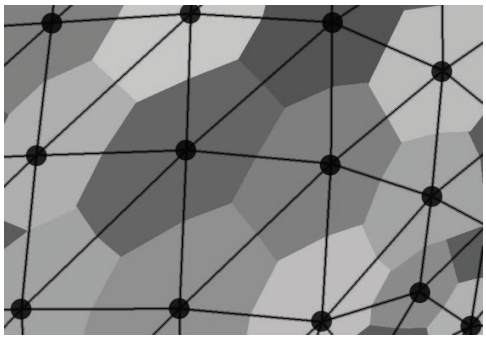


Figure 10.3: Contents of the picking-buffer. The simulation-surface is projected into the picking-buffer as represented by the black mesh. The surface is shaded with a color representing the texture coordinate in the position-texture (Figure 10.2) of the corresponding particle in the Spring-Mass system (Figure 10.1).

the high-resolution mesh used in the visualization-step due to the mapping presented in chapter 9. The simulation-surface is rendered as seen from the base of the associated instrument. It will be shaded with colors that provide texture coordinates to the nearest simulation node in the position-texture. The result is a Voronoi-like diagram as illustrated in figure 10.3. During force-buffer calculations we use this diagram to identify which particles in the Spring-Mass system are potentially touching the instrument. Consider a point on the boundary of the instrument. Transforming this point with the model-view and projection matrices that were used when rendering the picking-buffer, results in a picking-coordinate. Using this picking-coordinate for a texture lookup in the picking-buffer provides the texture coordinate to the corresponding particle in the position-texture due to the shading of the simulation-surface. To determine if the instrument is actually near the picked node, the third color-component of the picking-buffer stores the distance from the picked point on the simulation-surface to the instrument.

The discussion above describes how to use the picking-buffer to find the position-texture coordinates of the particles colliding with the probing instrument. It boils down to a single texture lookup for each sampling point on the boundary of the instrument. In the previously described case of grabbing, these position-texture coordinates were given directly as input to the fragment program which updated the force-buffer. When probing, we extend this fragment program to use instead a texture lookup in the picking-buffer to obtain the desired position-texture coordinate. The calculation of probing forces is initialized from the CPU, which keeps a list of sampling points on each instrument's boundary. One by one, the sampling points are projected onto the picking buffer and the resulting picking-coordinates passed to the force computing fragment program. Each resulting force is

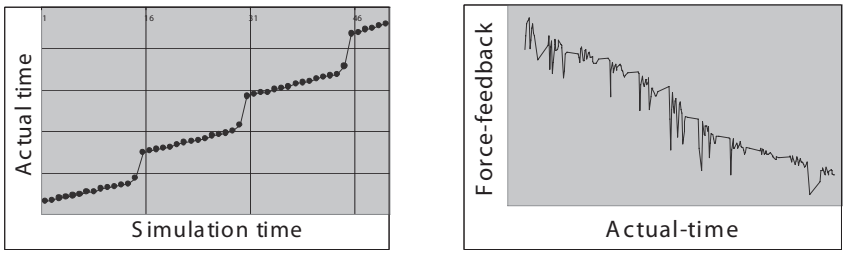


Figure 10.4: In a) is shown a graph of simulation-time vs. actual-time. The delay during visualization can clearly be seen as discontinuity. For correct haptic-interaction this graph should be linear. In b) the resulting effect on force-feedback can be seen clearly as spikes on the graph.

stored in the corresponding entry in the force-buffer, which is finally read back to the haptic thread on the CPU.

10.3 Smooth Haptic Interaction from Discontinuous Simulation Data

In this section we deal with a specific problem arising within the field of haptic-rendering. The problem occurs in the GPU based simulation since it supports the execution of several simulation-steps for each visualization-step. During a visualization step the simulation is suspended but the user can move the interaction-device. When simulation is resumed, the position of the virtual instrument, as seen from the simulation, will appear to have moved in a discontinuous way, see figure (10.4). A naive solution would be to try and "smooth" the uneven forces in figure (10.4) b). This is not a viable solution though, since it does not solve the actual problem. In general terms, the problem is that the execution time for a simulation-step does not correspond to the size of the time-step in the numerical integration (i.e. *simulation-time*), and that the additional delays introduced are not constant. Any interaction-data (in *actual-time*) should consequently be realigned correctly with the actual execution time of the corresponding time-step. The non-uniform distribution of simulation-steps not only arises from the pause during visualization, but also from internal resource distributions on the GPU. That is, even a pure sequence of simulations steps may not arrive at regular intervals. As a result of all these issues, both the positions of grabbed nodes (*from* haptic-device) as well as the read-back forces (*to* haptic-device) are wrong in relation to the simulation. The effect is clearly felt as uneven and noisy force-feedback, see figure (10.4).

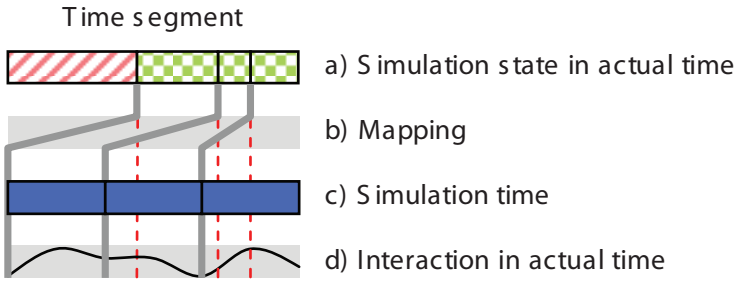


Figure 10.5: Given a time segment of one visualization-step and three simulation-steps. a) shows the actual time of execution for the visualization step (diagonal-pattern) and simulation-steps (checkerboard-pattern). c) shows how the time-segment is divided as seen from simulation-time (i.e. the size of each simulation-step is constant). b) depicts the correct mapping between simulation time and actual time. d) shows the arrival of interaction-data in actual time. Notice how an incorrect direct use of interaction data (dashed lines) would e.g. miss the big chunk of interaction-data collected during the execution of the visualization-step. Through the mapping b) the interaction-data is realigned correctly to the actual execution time of the time-steps (Follow the fat lines down through the figure).

10.3.1 Method

For the highest quality of haptics, force-feedback is collected after each simulation step. We propose to realign interaction-time with simulation-time as follows (see figure 10.5). The average length of a time-segment is maintained and the number of simulation-steps in a time-segment is known. From this we divide the time-segment into even intervals - one for each simulation step, see figure 10.5 c). Assuming a time segment starts with visualization, interaction-data will arrive while no simulation-step can receive it. A buffer of interaction-data is consequently maintained. This buffer allows retrieval of interaction-data corresponding to the simulated time through interpolation of the buffered interaction-data.

10.4 Results

Table 10.1 summarizes the performance measurements obtained from the simulator for each of the tested graphics cards. As expected the newest GPU, the GeForce 7800 GTX, is the fastest for fragment and vertex processing (rows 1-3). It performs significantly better than the other two cards. To calculate and read back the accumulated spring forces the cards perform comparably due to the relatively high synchronization cost of initiating a data transfer (row 4). A simulator was developed to support haptic inter-

Table 10.1: GPU rendering times on selected graphics cards.
[1]One simulation-step in a Spring-Mass system of 20.270 nodes (18 neighbors each).
[2]One visualization-step (90.868 vertices / 137.490 faces).
[3]One rendering step of the off-screen picking-buffer (12.031 vertices / 46.928 faces).
[4]One calculation and subsequent read-back of force feedback from 50 particles.

	GeForce 6800 Ultra	Quadro FX 4400	GeForce 7800 GTX
Simulation-step [1]	2.5 ms	3.5 ms	0.9 ms
Visualization-step [2]	19.9 ms	20.8 ms	11.1 ms
Picking-buffer [3,4]	6.3 ms	5.3 ms	2.6 ms
Force-buffer [4]	0.3 ms	0.7 ms	0.2 ms

action with cardiac models.

The result of applying the proposed method of section 10.3 is that interaction data from the haptic-device arrives at the simulation-step at correct simulation-time, delayed maximally with a visualization-step. Forces from the tissue-dynamics are read-back after each step of a simulation and now correspond correctly to the time of interaction, delayed on visualization step, see figure 10.6.

10.5 Discussion

It is clear from table 10.1 that surface visualization is the single most expensive task in the simulator. In fact, several simulation-steps could be performed for each visualization-step while maintaining an overall frame rate of at least 30 Hz. Here the term overall frame rate covers the accumulated cost of a number of simulation-steps, a visualization-step, rendering of the picking-buffers, and finally calculation and read-back of the force buffers. Each simulation-step should be followed by rendering and read-back of the force-buffer to ensure the highest update frequency of the haptic

Figure 10.6: Applying the proposed method to the interaction from figure 10.4 b) shows a clear removal of noise.

devices. When probing, the additional cost of updating the picking-buffers must also be considered. As seen from the 3rd row in table 10.1, updating the picking buffer after each simulation-step significantly reduces the number of simulation-steps possible per overall frame. We briefly mention two strategies for high-frequency haptic rendering that do not necessitate picking-buffer updates after each simulation-step. The first strategy is to read back the force feedback from the GPU at e.g. 30 Hz and use the algorithms presented in [125, 155] to extrapolate this data to the desired update frequency on the CPU. Another strategy is to allow the use a slightly outdated picking-buffer but proceed with the update and read-back of the force-buffer based on updated instrument positions. We have chosen the latter of these two strategies.

Using the GeForce 7800 GTX as an example, we describe a combination of steps that will lead to an overall frame rate of 30 Hz running the simulator with haptic interaction. An overall frame rate requirement of 30 Hz corresponds to 33 ms available per frame. 11.1 ms are used for surface visualization. Two picking-buffers (one for each hand) will be updated once in each overall frame, leaving 17 ms for simulation and force read-back. During this period we can perform approximately 15 simulation-steps with subsequent rendering and read-back of the force-buffers. This corresponds to a simulation-step frequency of 450 Hz.

Comparing the overall system performance to a similar implementation on the CPU, the GPU implementation is much faster. With the work presented in this article we believe to be one step closer to a fully functional cardiac surgery simulator. As a next step we are planning to extend the simulator with support for suturing of patches to close e.g. septal defects.

Chapter 11

Building Virtual Models of the Heart

In this short chapter we present an overview of the process of building the virtual models of the heart for the GPU accelerated surgical simulator. The methods presented in the following chapter are based on ongoing development with Thomas Sangild Sørensen and Jean Stawiaski.

11.1 Segmentation

We have developed a semi-automated segmentation method based on the Watershed transform [188]. It is applied to 3D cardiac MRI to produce patient-specific models for virtual heart surgery. Our method can be used interactively and efficiently to create specific-models of individual morphology.

11.1.1 Algorithm

The Watershed transform uses an intuitive description of boundary in an image: It considers an image as a topographic surface where the height of each point is directly related to its gray level. The algorithm then simulates a flooding of this surface from a finite set of points. To avoid mixing of water of different sources, a watershed line is constructed where they meet. The watershed line computed on the gradient of an image finds the high gradient points which are related to boundaries in the image. In cardiac MRI this corresponds to e.g. the border between the blood pool and the myocardium. To avoid over-segmentation due to noise in the images, the set of points from which the flooding process starts is defined interactively by the user [18].

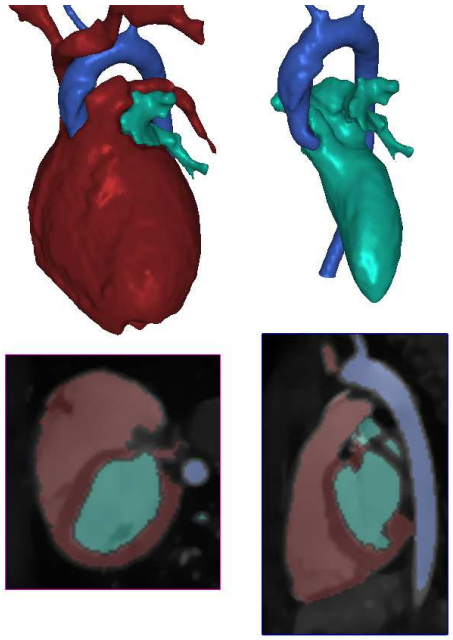


Figure 11.1: Example of a segmented image illustrated by surface rendering and cut planes overlaid on the original data.

11.1.2 Imaging and Segmentation

All models are reconstructed from 3D MRI acquired with isotropic voxels at a resolution of approximately 1.73 mm^3, which forms a dataset of 256x256x100 voxels. The images are first filtered using anisotropic diffusion [191]. The user then specifies the different objects of interest in the image by inserting a few markers from which the watershed transform is invoked. The user can place markers interactively by drawing on 2D slices. This method was used to segment the blood pool and myocardium volume semi-automatically (figure 11.1). After the segmentation step, the user can add or delete markers if it is necessary.

11.1.3 Image Visualization

We have developed software dedicated to heart MRI segmentation and visualization. Our software allows the user to explore a highly detailed view of the dataset for easy interpretation. This is important for validation purposes. The user can also view the segmented image as a set of surfaces, one surface for each region. The marching cubes algorithm was used to extract detailed surfaces of the segmented image. Users can visualize the surfaces and overlay it with the original dataset. It is also possible to view 2D slices of both the segmented and the original image as well as a 3D volume rendering

of the original dataset (figure 11.1).

11.1.4 Software

The user interface of our software was developed with python Tk-inter and visualization of the images is performed using the Visualization Toolkit (VTK). The segmentation tasks are done with our own custom image processing library. The software can be easily extended for other segmentation tasks, e.g. other organs and imaging modalities.

11.1.5 Discussion

We encountered problematic issues with the segmentation of the right ventricular epicardium due to bad contrast of the bordering tissue. The resolution of the images is often too small to distinguish clearly this part of the heart. However we believe the problem can be solved via new methods based on graph-cuts and minimal surfaces [25] .

11.2 Models Suitable for Surgical Simulation

Our aim is to obtain a volumetric segmentation of the heart muscle (the myocardium) and vessel walls and use a Spring-Mass simulation to interactively deform this volume. Surface visualization of the deformed volume is fully decoupled from the physical simulation.

11.2.1 Obtaining the Volumetric Simulation Grid

A typical 3D MRI results in a dataset of 256x256x100 voxels [173]. The number of tissue voxels classified by the segmentation process is in the order of 250.000. Even for an optimized Spring-Mass system resolved on the GPU, simulation and convergence rates are inadequate for such a large system. We consequently down-sample by a factor of 23 to obtain approximately 30.000 simulation nodes in a regular grid.

11.2.2 Surface Visualization Processing

A highly detailed surface is extracted of the endocardium-, epicardium-, and vessel borders at the full resolution of the segmentation by the marching cubes algorithm. The resulting model easily contains 1.000.000 triangles. As described in chapter 9 we represent each vertex by an offset from a simulation grid node. This deforms the surface based on the deformation of the underlying simulation grid, but also requires some per vertex processing. As this is a relatively slow operation we reduce the triangle count in the model to avoid a visualization bottleneck. Normal maps are used to conserve the details of the high-resolution mesh in the simplified model of 50.000-100.000

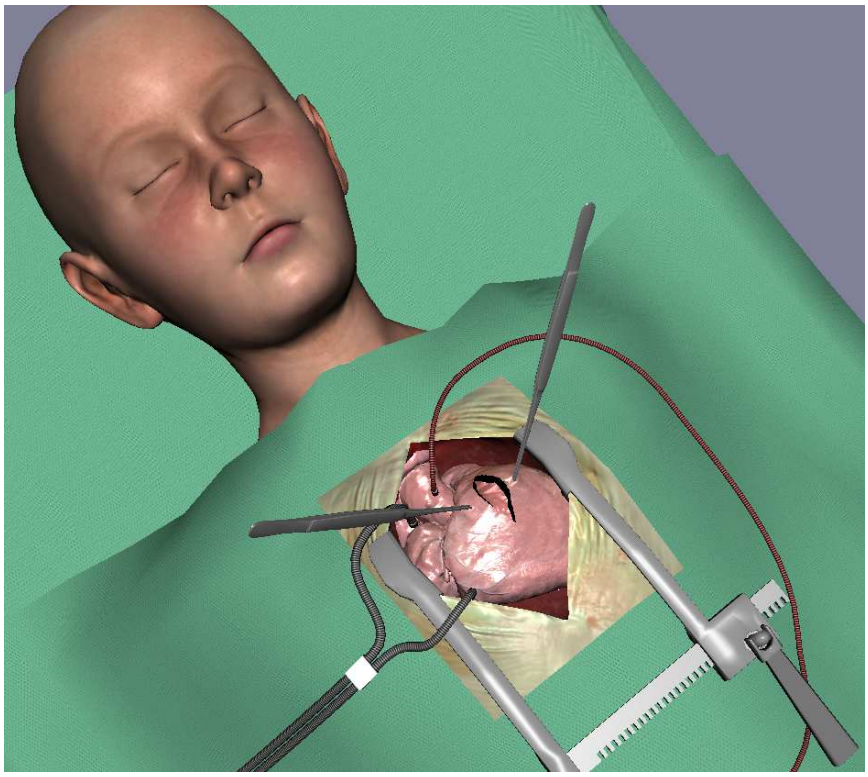


Figure 11.2: Virtual heart environment with child, the open chest, several pieces of green paper covering body and head, and the operating table. The visualization of each element can be turned on and off.

faces. As described in chapter 10 our implementation of force feedback requires off-screen rendering of a low resolution model with a vertex for each *surface point* in the simulation grid. This model is obtained again from the marching cubes algorithm.

11.2.3 Discussion

We see two potential scenarios for the current simulator prototype; patient-specific preoperative planning, and surgical education. For the first scenario we can segment the blood pool, the left sided myocardium and grow vessel walls in an hour in good quality datasets. The right sided myocardium remains a challenge to model accurately, but can be "grown" to a certain thickness surrounding the blood pool by a dedicated volume paint application.

For the educational scenario we can spend the time it takes to create each model. In figure 9.9 we normal mapped the myocardial surface with the coronary arteries drawn manually by a graphics artist. In both patient-

specific preoperative planning and surgical education the final virtual model can be placed into the virtual child of figure for added realism.

Chapter 12

Using the Surgical Simulator for Incision Planning

This chapter is based on the journal paper [13] and abstract [12], co-authored with clinical personnel from both Denmark and Germany, and presents a preliminary informal evaluation of the GPU accelerated surgical simulator for incision planning in training and pre-operative planning. The chapter has been considerably expanded in almost every section compared to the original paper [13] specifically an additional CT dataset is included in the discussion of pre-operative planning.

12.1 Introduction

Recent advances in high-resolution, three-dimensional (3D) imaging modalities such as magnetic resonance imaging (MRI) and multidetector computed tomography (MDCT) have provided new means to virtually reconstruct the morphology of the heart [24, 173, 67, 174, 93, 160]. To prevent degraded image quality as a result of blurring from cardiac motion, the underlying image acquisition is triggered to the resting period of the heart. Hence, both the intracardiac and the extracardiac morphology can be modeled accurately. For patients with complex congenital heart disease, exploration of these virtual reconstructions has become an integrated part of the preoperative planning process in many institutions [67, 93, 160, 174]. In addition, the virtual models can be reproduced as cast models with rapid prototyping [138].

In several areas of surgery, preoperative planning and surgical training using virtual models has been taken one step further. As seen previously in this thesis, surgical simulators are available, which allow entire surgical procedures to be rehearsed in a virtual environment. A typical surgical simulator provides elastic tissue deformations in response to user interaction, and supports cutting and suturing in the virtual model. It is essential that

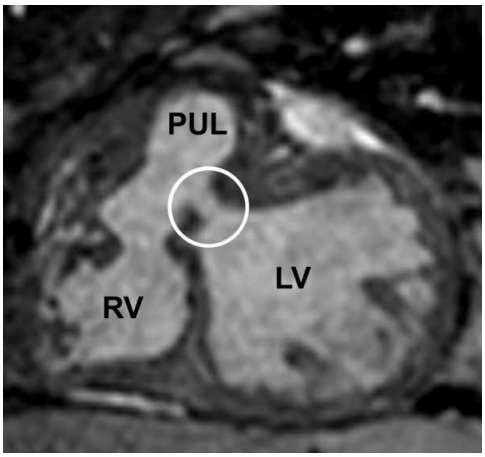


Figure 12.1: Reformatted slice from a isotropic 3D SSFP MRI acquisition [173] in patient 1 showing the larger of two ventricular septal defects (circle). Abbreviations. RV: right ventricle, LV: left ventricle, PUL: main pulmonary artery.

the simulator responds to user interactions in real-time in order to imitate real procedures realistically. For a complex organ like the heart however, the existing techniques have not been able to run interactively until very recently. In fact, very few simulators have previously approached any form of open surgery.

This chapter presents our first experiences using a real-time surgical simulator for incision planning in relation to congenital heart disease. In this initial work we investigate two hypotheses, namely that 1) patient-specific incision planning using deformable virtual models of the individual morphology can be used to pre-operatively determine which incisions provide the best access to access a given defect, and 2) on a generalized virtual model containing any desired ventricular- and/or atrial septal defects, incision simulation can be used as an educational tool to illustrate various incision strategies to access these defects.

12.2 Materials

The study population for each of the two intended scenarios is described below.

12.2.1 Patient-Specific Simulation

To study our first hypothesis of patient-specific simulation, incision simulation was performed to evaluate potential corrective strategies for two patients clinically referred for three-dimensional MRI [173].

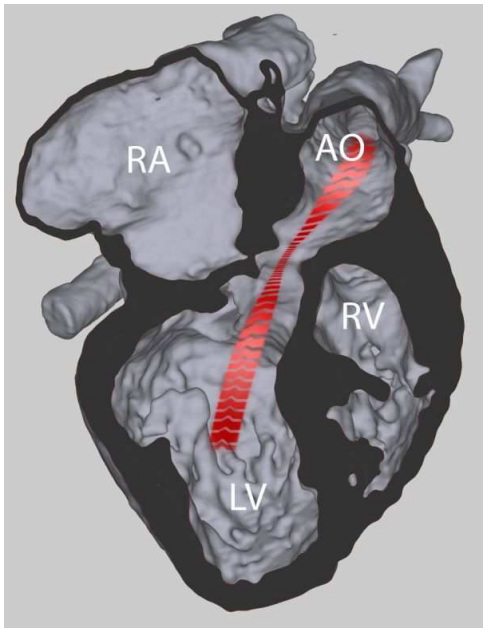


Figure 12.2: Three-dimensional reconstruction in patient 1 showing an intraventricular baffle (streamlined) leading blood from the left ventricle to the aorta. Abbreviations. RA: right atrium, RV: right ventricle, LV: left ventricle, AO: aorta.

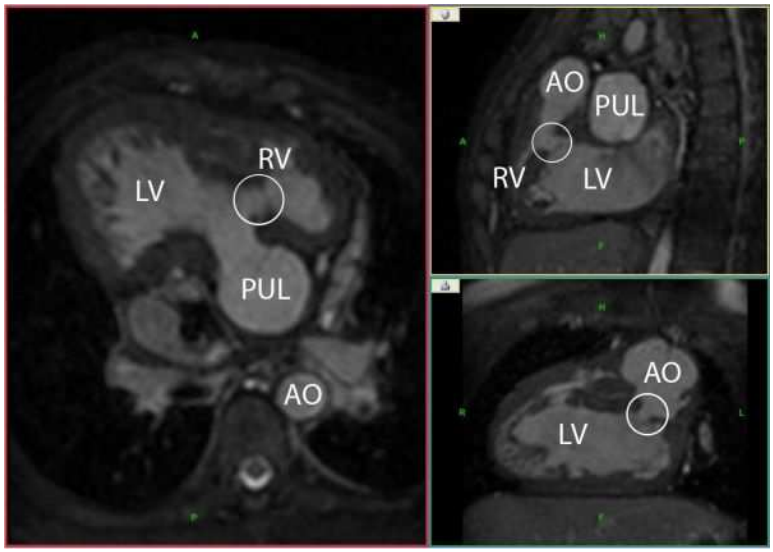


Figure 12.3: Three reformatted images from an isotropic 3D SSFP MRI acquisition [173] in patient 2. The circles mark a restrictive ventricular septal defect as seen from a sagittal- (top right), a coronal- (lower right), and an oblique transversal view (left). Abbreviations. RV: right ventricle, LV: left ventricle, PUL: main pulmonary artery, AO: aorta.

Patient 1 was a ten year-old girl born with double outlet right ventricle, the great arteries side-by-side with the aorta to the right, and a subpulmonary VSD. An intra-ventricular baffle was inserted for a biventricular repair. Subsequently, echocardiography and MRI has revealed that two baffle leakages remain following this procedure (See figure 12.1). Furthermore, the intra-ventricular tunnel appears restrictive (See figure 12.2).

Patient 2 was a five year-old boy with a univentricular heart (dextrocardia, hypoplastic right ventricle, discordant ventriculo-arterial connections, and left pulmonary artery stenosis). Following three Fontan operations resulting in total cavo-pulmonary connection (TCPC), he was examined for a possibly restrictive VSD (See figure 12.3).

Incision simulation was performed to evaluate potential corrective strategies. In both cases the exact positions of the ventricular septal defects were visualized through the chosen incisions. In the first patient we additionally examined the narrow intra-ventricular tunnel. The purpose was to use this information in order to decide on the best surgical approach. The study was approved by the institutional ethics committee on human research.

12.2.2 Generalized Incision Simulation

To examine our second hypothesis of a generalized surgical simulator we obtained both a MDCT dataset of a plastinated adult heart specimen from a patient with dilated cardiomyopathy[1] and an MR based dataset of an adult volunteer's normal heart.

Prior to reconstruction of the CT dataset, we manually introduced a subpulmonary muscular ventricular septal defect (VSD) and an apical VSD in the dataset. From the MR dataset we obtained a configurable model of several atrial- and ventricular septal defects in which the defects were added by hand.

The basic idea in this scenario is to use the surgical simulator as an educational tool to locate the VSDs both trans-atrially and trans-ventricularly, and to subsequently compare the accessibility to the VSD by the two approaches [16, 186, 184]. The purpose of this scenario was to make a preliminary examination of the potential for such a tool in the surgical education: Is it likely to help overcoming learning curves by moving parts of the surgical training to a virtual environment as has happened in the education of laparoscopic surgery [69, 72, 115]. In this context *overcoming a learning curve* captures the process of moving along the learning curve until the point where one can safely perform a specific procedure in a patient.

[1] A disease that make the heart become enlarged

12.3 Methods

Accurate imaging and segmentation are prerequisites to reconstruct a virtual model suitable for incision simulation. Furthermore, a realistic looking, interactive model for tissue deformation is required.

12.3.1 Imaging

MRI was performed on an Intera 1.5T scanner (Philips Medical Systems, Best, Netherlands) using an isotropic, three-dimensional steady state free precession acquisition protocol [173] under general anesthesia with respiration gated to a 5 mm window on the right hemi-diaphragm. Images were obtained with a resolution of 1.75 mm^3 and acquired in end-diastole.

MDCT of the plastinated heart specimen was performed in approximately 10 seconds using a high-resolution acquisition (in plane resolution: 0.34 x 0.34 mm, slice thickness: 0.75 mm, slice gap: 0.50 mm) on a Sensation 16 scanner (Siemens Medical Solutions, Erlangen, Germany)[67].

12.3.2 Segmentation

From each MRI dataset we reconstructed a virtual model of the myocardium and vessel borders using the Virtual Reality Heart software (Systematic Software Engineering, Denmark) and custom software.

The MDCT data of the human heart specimen were easily segmented and converted to a 3D model by defining the threshold value separating the tissue from air using custom software.

12.3.3 Simulation

A Spring-Mass system was used to simulate the elastic properties of the modeled tissue as described in the previous chapters 7, 9 and 10. Spring-Mass systems have been used intensively in the medical simulation literature previously, but have only been running interactively on simple morphology. Fortunately, by resolving the necessary force computations entirely on modern consumer graphics cards, the Spring-Mass system could be applied to complex morphology such as the heart as well. Two-handed interaction was achieved by two Phantom Omnis (Sensable Technologies, USA) providing the exact three-dimensional position and orientation of each hand (See figure 12.4). In addition, each of these devices was programmed with force feedback giving the user the sensation of being able to touch the surface and feel the magnitude of the forces applied during the procedures as described in chapter 10.

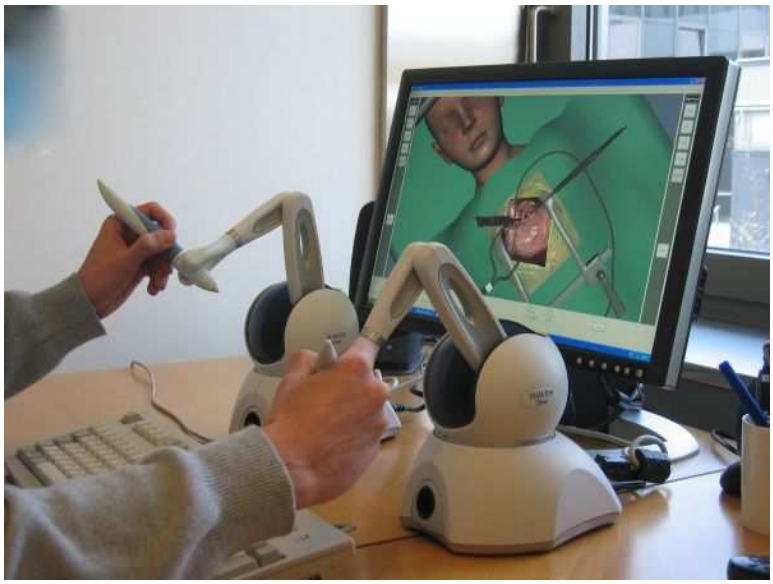


Figure 12.4: Setup of the surgical simulator. Two Phantom Omnis (Sensable Technologies, USA) are used for free-hand interaction with force feedback. Cutting and tissue manipulation can be performed interactively.

12.4 Results

The 3D MRI acquisitions were completed successfully for each of the two patients in less than ten minutes each. Unfortunately, segmentation of the myocardium was quite time consuming and required about 10 hours for each of the two dataset: In approximately 20 minutes our software was capable of identifying the blood pool in the images and hence determine the endocardium and blood pool border. Identifying the epicardium border was much harder and ended up taking practically all the segmentation effort. We ended up tracing a significant part of the epicardium border manually. Vessel walls were automatically "grown" from the endoluminal borders.

The human heart specimen was MDCT scanned successfully with sub-millimeter resolution. As only tissue generated signal in this dataset, segmentation and reconstruction could be done instantly.

The muscular and the apical VSDs and ASDs in both the MR and CT dataset were added manually, requiring just a few additional minutes of work.

In the virtual models arbitrary incisions could be made and the bordering tissue pulled aside, using the elastic behavior of the tissue to provide realistic views of the intra-cardiac morphology.

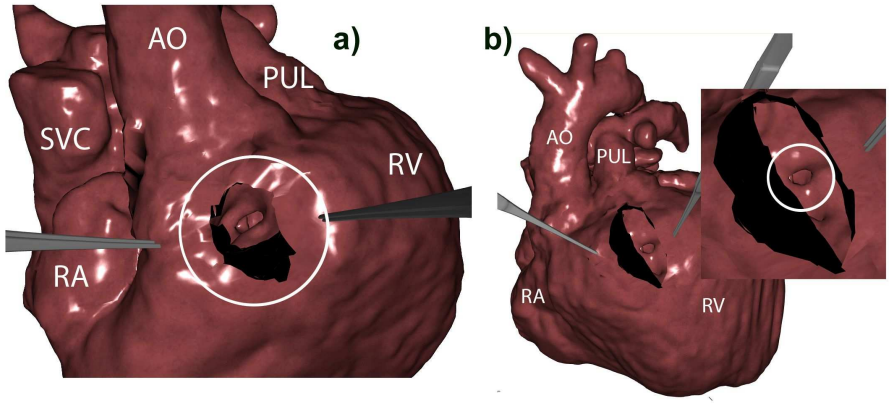


Figure 12.5: Incision simulation in patient 1. a) An incision has been made at the aortic root (in the right ventricle) showing a narrow intraventricular pathway (circle). The tips of two surgical tools that keep the incision open can be seen. b) An incision at the pulmonary root reveals the position of a ventricular septal defect (circle in enlargement). Abbreviations. RA: right atrium, RV: right ventricle, AO: aorta, PUL: main pulmonary artery, SVC: superior vena cava.

12.4.1 Patient Specific Scenario

Figure 12.5 shows two incisions in the virtual reconstruction of patient 1. In figure 12.5 a) an incision at the aortic root was made to visualize the interior baffle pathway (white circle). From this view it is clear that the narrow baffle is restrictive. Surgical replacement of the baffle has been discussed but no final decision made. Figure 12.5 b) shows an incision at the root of the main pulmonary artery revealing the exact location of the larger of the two ventricular septal defects (circle in insert). This location allows closure either by surgery or by catheterisation. Considering the complex surgical procedure necessary to replace the baffle compared to the good clinical condition of the patient, the patient has not been further operated up to now. However, the ideal access to the encircled VSD (See figure 12.5 b)) was clearly demonstrated by the simulator.

Figure 12.6 shows two incisions in patient 2. Figure 12.6 a) shows our initial approach: an incision at the aortic root (viewed from above). The simulator reveals that the location of the VSD (white circle) would be difficult to reach from this incision simply due to the distance. Fortunately the simulator allows us to rethink and redo the incision: By moving it slightly downwards as shown in figure 12.6 b,) the VSD is now more directly accessible (white circle). Very importantly, the course of the coronary arteries is also visible and avoided by the current incision. Based on the available imaging information, it was decided that surgery to enlarge the VSD is necessary.

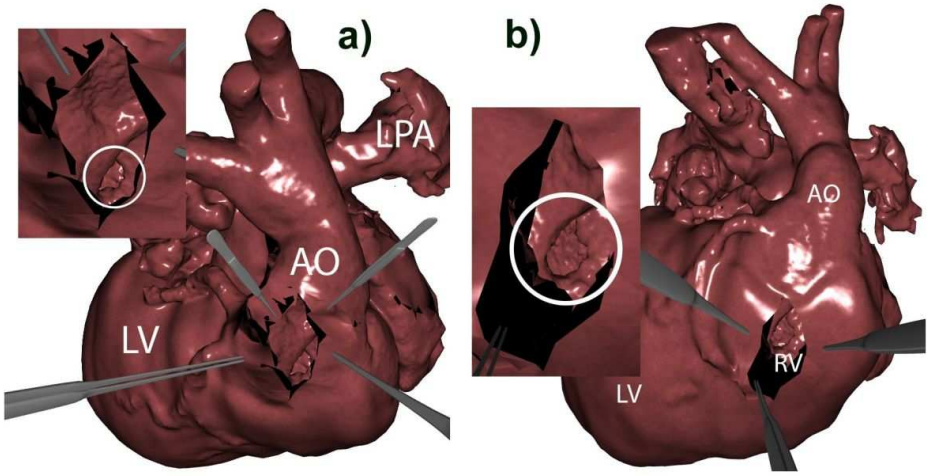


Figure 12.6: Incision simulation in patient 2. a) Viewed from above, an initial incision at the aortic root reveals a restrictive but hard-to-access ventricular septal defect (circle in enlargement). b) From an anterior view, a second incision makes the VSD more easily accessible (circle in enlargement). Abbreviations. AO: aorta, RV: right ventricle, LV: left ventricle, LPA: left pulmonary artery.

Of the two surgical strategies illustrated in figure 12.6, the incision shown in figure 12.6 b) was the most promising and it has been recommended.

12.4.2 Generalized training scenario

Figure 12.7 shows four incisions, two right ventriculotomies, an apical infundibulotomy, and a trans-atrial approach to reveal the muscular and the apical VSDs (white circles) introduced in the plastinated heart specimen. Figure 5a shows an incision covering a significant part of the right ventricular wall perpendicular to the acute marginal branches of the right coronary artery (RCA). Although damaging to the electrical conduction system and hence not feasible for actual surgery, it shows the exact location of the muscular VSD. In figure 12.7 b) the transventricular incision was moved to the pulmonary root avoiding the RCA branches. It is now necessary to use an instrument to get access to and visualize the defect. In figure 12.7c) an apical infundibulotomy provides direct access to the apical VSD. Finally a trans-atrial approach is used in figure 12.7 d): The muscular VSD can be identified when the tricuspid orifice is moved slightly by an instrument. The apical VSD is not visible from this view however [186].

Figure 12.8 shows a trans-ventricular incision (a) and a trans-atrial incision (b) to reveal a mid-muscular VSD that was manually introduced in the 3D MRI of a volunteer. Figure 12.8a shows an incision at the root of

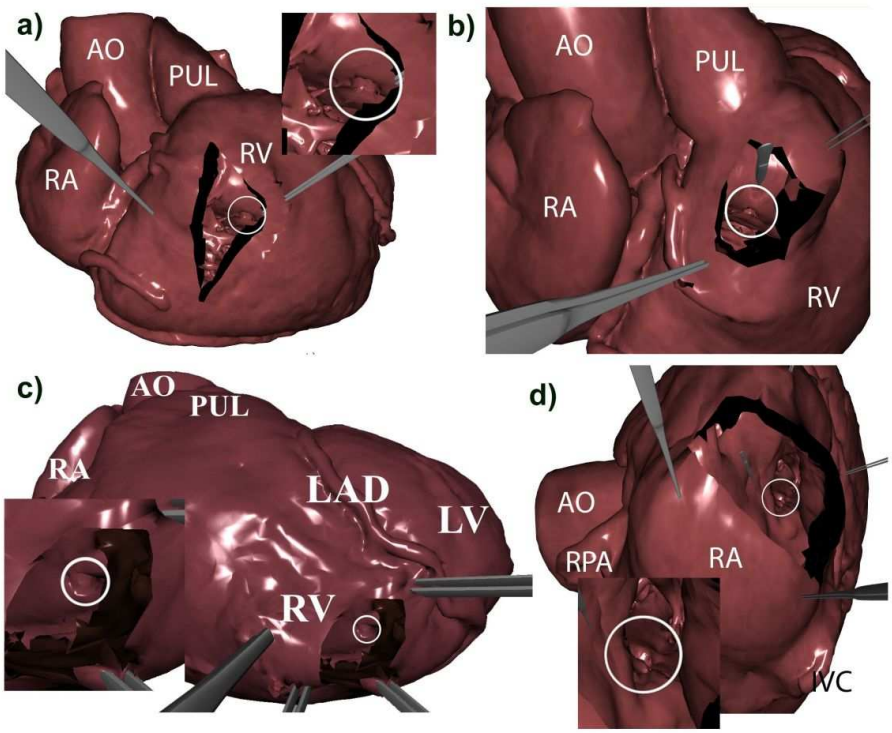
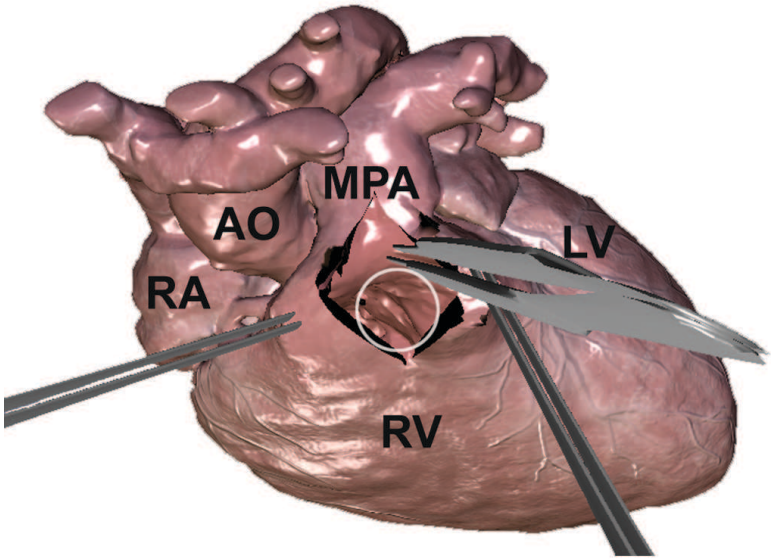
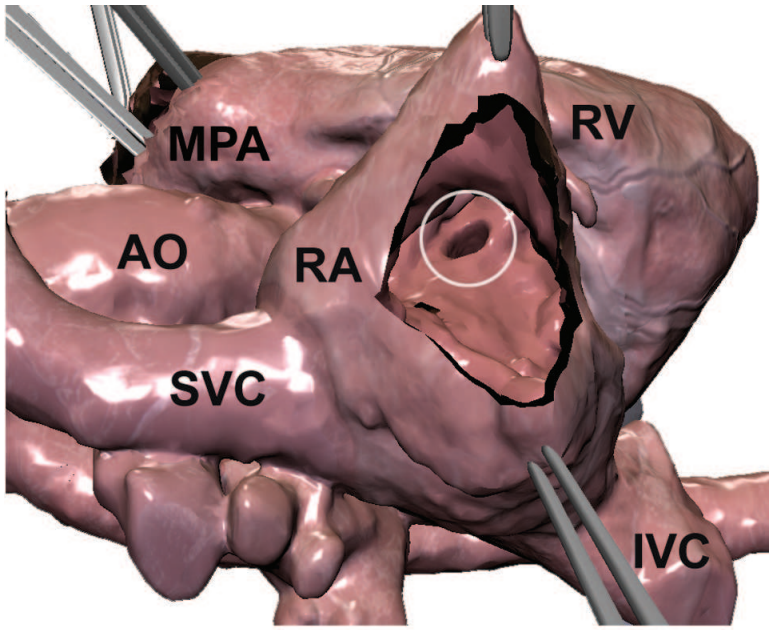


Figure 12.7: Incision planning as a generalized training scenario. Four incisions are used to locate a muscular and an apical ventricular septal defect (circles) in the virtual reconstruction of a plastinated heart specimen. a) A right ventriculotomy reveals the exact position of the muscular VSD. b) The muscular VSD visualized through an incision at the root of the main pulmonary artery (PUL). The right- and leftmost surgical tools hold the incision open, while a third tool (center) pulls the septal wall to reveal the defect. c) The apical VSD visualized through an apical infundibulotomy. d) The muscular VSD visualized trans-atrially. The central tool pulls the tricuspid orifice slightly to improve accessibility. Abbreviations. RA: right atrium, RV: right ventricle, LV: left ventricle, AO: aorta, PUL: main pulmonary artery, RPA: right pulmonary artery, IVC: inferior vena cava, SVC: superior vena cava, LAD: left anterior descending coronary artery.



(a)



(b)

Figure 12.8: Incision simulation as an educational tool. A muscular ventricular septal defect (circles) is accessed by a trans-ventricular incision (a) and a trans-atrial incision (b). Abbreviations. RA: right atrium, RV: right ventricle, LV: left ventricle, AO: aorta, MPA: main pulmonary artery, IVC: inferior vena cava, SVC: superior vena cava.

the main pulmonary artery perpendicular to the acute marginal branches of the right coronary artery and the left anterior ascending coronary artery. The muscular VSD is revealed (white circle) by the central instrument. The access to this VSD by a trans-atrial approach is shown in figure 4b. The initial incision is visible at the top-left part of the subfigure.

12.5 Discussion

For many years preoperative planning in congenital heart disease has relied on the abilities of surgeons to convert two-dimensional imaging information to three-dimensional mental models and surgical strategies. In order to decide on the most optimal surgical strategy methods such as e.g. echocardiography, catheterisation, MRI, or MDCT have been used as the basis for evaluating the feasibility of potential strategies. Virtual reconstructions were recently introduced as a new supplementary tool to assist in this process, giving a three-dimensional overview of the morphology [67, 174]. In this article we have taken this work one step further: By providing an interactive setup that offers surgeons an opportunity to make arbitrary incisions on elastic virtual models, realistic three-dimensional surgical views of the intracardiac morphology can be provided pre-operatively. A superior spatial understanding of potential surgical incisions in relation to individual morphology can be obtained since the simulator lets the user examine morphological details and incisions from any desired angle. Subsequently, when a good understanding of the individual morphology has been reached, a specific surgical strategy can be evaluated by making the desired incisions using the simulator and estimate the accessibility to the relevant cardiac structures.

In patient number 1 we used this tool to visualize a restrictive intraventricular baffle and an associated VSD. The information provided by this representation of the intracardiac morphology has been an important factor in the currently ongoing decision-making for the patient. It is our belief that surgical procedures are facilitated by the improved 3D understanding of complex intracardiac malformations.

In patient number 2 we visualized a restrictive VSD, this time through two slightly different incisions. Due to the risks of impairing right ventricular function associated to an extended ventriculotomy, the initial incision was placed right at the root of the main pulmonary artery. It turned out, however, that through this incision it would be hard to reach the VSD (Figure 12.6a). Consequently, a new incision was made which provided better access to the defect (Figure 12.6b). Again, the incision simulator provided important preoperative information on how to proceed with the actual surgery. Similar experiences regarding the surgical approach of this type of defect have previously been reported [176].

In figure 12.7 we used the simulator with a slightly different purpose, namely to generally illustrate the access to a muscular VSD by a transatrial approach, by a right ventriculotomy, and by an apical infundibulotomy in figure 12.7. In our search to locate the VSD we initially performed a large right ventriculotomy, which in real-life would have severe consequences for the ventricular function (Figure 12.7a). The simulation however provided us with this freedom to get an overview of the morphology. With this overview in mind we were able to quickly locate the VSD through a less extended right ventriculotomy in 12.7 b). As suggested by Van Praagh, Tsang, and Myhre [186, 184, 139] an apical infundibulotomy would be an alternative approach. This is illustrated in figure 12.7 c). To complete the picture, we visualized the VSD by a transatrial approach in figure 12.7 d). Hence, the access to the defect could be examined from several surgical strategies. This first general heart-model was based on CT images and as such we also demonstrated a general heart-model based on MRI. In figure 12.8 we used the simulator to generally illustrate the access to a septal defect by two different incisions in a virtual heart model based on MRI. We specifically demonstrated the access to a ventricular septal defect by a trans-ventricular and a trans-atrial approach. As all defects were introduced manually by post-processing, any imaginable position and size of the VSD would have been possible. We believe that the freedom in the points we are able to teach using an interactive incision simulator makes it a promising new educational tool to illustrate various incision strategies to access septal defects. By this discussion we approve our second hypothesis.

We are currently using the tools described in [174] to segment the 3D MRI and reconstruct the morphological models. This is currently a time consuming process as the software was designed to rapidly identify the blood pool but not the myocardium. We are consequently continuing our research in segmentation algorithms to better detect the epicardium border. In combination with a "3D sculpturing tool" in preparation we optimistically expect to be able to create patient-specific models suitable for incision simulation in just an hour or two in the near future - provided a good quality 3D MRI is available.

Speculating on the basis of future improvements to the simulator, the advantages of surgical simulation are (at least) two-fold. Firstly, we can illustrate various elements of surgical procedures, and secondly, we can allow surgeons to rehearse these elements virtually. The incision planning tool we presented in this chapter is the first step on a long road. When support for suturing and handling of patches is added, we could potentially rehearse complex or rare surgical procedures (e.g. a double outlet right ventricle repair or a Mustard operation) in a virtual environment. In figure 12.6 and 12.7 there was no constraints on the angles and positions from which the heart could be examined and accessed. In future scenarios, the natural limitations caused by the positioning in the thorax needs to be addressed

in order to achieve a realistic training scenario. One step in this direction is shown in the supplementary online movie (included on the DVD of this PhD thesis) and in figure 12.4, in which a model of the thorax was included in the graphics.

The education of young surgeons relies on a master/apprentice relationship and follows the "see one, do one, teach one" principle in broad terms. We believe that in a foreseeable future, surgical simulators will become important tools to aid in these transitions. They could provide numerous training scenarios with an unlimited number of trials that will allow young surgeons to experiment and learn from their mistakes. Eventually, these surgeons will be better prepared for their upcoming work in the operating theatre. Fully developed surgical simulators also hold potential to increase the cost-effectiveness of surgical training. The time used for training could become shorter since particular and even rare cases could be performed repeatedly without any preparation. Furthermore, optimally prepared procedures are likely to minimize the costs of follow-up treatments.

Chapter 13

Conclusion and Future Work

Each of the major chapters 7, 8, 9, 10, and 12 based on published papers have presented conclusions, discussions, and future work sections on each of the subproblems. In this chapter I will shortly recap some of the highlights and revisit the original problem definition.

13.1 Conclusion and Discussion

I will begin the final conclusion by revisiting the original problem definition. The computer science problem formulation from section 1.2.2 is reprinted here for ease:

Problem Simulating and visualizing tissue deformation in models with complex morphology is not fast enough with existing methods and hardware. Heart surgery is such a case, and has thus not previously been simulated.

Hypothesis Through utilization of graphics hardware, a surgical simulator for complex morphology can be constructed - thereby meeting the demands of the previous item.

Method Develop and evaluate the technical components for a GPU-based surgical simulator including soft-tissue simulation, visualization and haptic-interaction.

Limitations The developed techniques should be generally applicable in physics based animation.

Through the last part of the thesis I have presented our work on the GPU accelerated surgical simulator for complex morphology. Through a series of chapters coinciding with published papers I have presented the different modules necessary for a complete surgical simulator; tissue-deformation (chapter 8), visualization (chapter 9) and haptic-interaction (chapter 10).

Although pediatric cardiac surgery has been my major focus throughout this PhD-project, each of the methods presented are generally applicable within the field of physics based animation. The Spring-Mass model itself is used in many different cases not related to surgical simulation, and as such our work on GPU accelerated Spring-Mass is applicable to each of these cases. In many cases the Spring-Mass model is used for real-time interaction and as such our work on visualization and haptics-interaction is also applicable in general. On several occasions through our work on the GPU accelerated surgical simulator, we have been made aware of the fact that many people in the surgical simulation community would like to take advantage of the GPU. It seems that the learning curve is rather steep though. We consequently published an introductory paper [10] on which chapter 7 was based. The original problem formulation has thus been dealt with in detail.

The clinical problem formulation from section 1.2.1 is reprinted here:

Problem Surgery on the heart of a child with a congenital heart defect requires good understanding of the heart morphology in relation to the steps of potential surgical strategies.

Hypothesis Going through the surgical procedure, experimenting and exploring the heart in a virtual setting, provides a better understanding of the heart morphology in relation to potential surgical strategies in both pre-operational planning and in training in general.

Method Develop a working prototype of a real-time surgical simulator for congenital heart defects.

Limitations Although we have a tight interdisciplinary cooperation, the major goal of the clinical problem formulation in terms of this PhD-thesis is as a facilitator for interesting aspects of computer science. As such a formal clinical evaluation and implications of the surgical simulator is not given primary attention in the remaining thesis. We have conduced a preliminary survey of the tool as an incision simulation though which is presented in chapter 12.

In chapter 12 we have demonstrated in cooperation with pediatric surgeons that the surgical simulator as a pre-operative tool and training tool has the potential to give surgeons more information on incision planning. At a more informal level, the surgeons have been involved throughout the entire process and as such we have worked towards solving the problem as stated in the clinical problem formulation. Concerning the clinical hypothesis, we cannot conclude, based on the executed research, that the simulator will

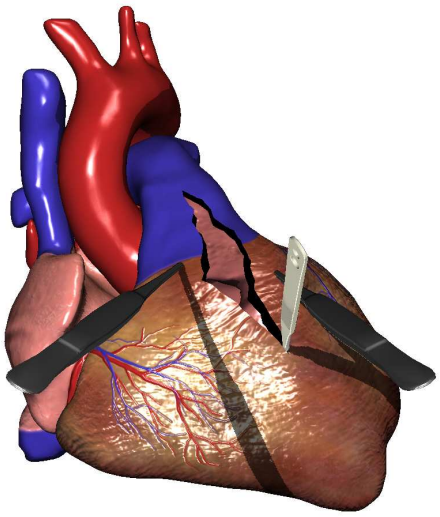


Figure 13.1: A reconstruction of the heart of the author of this PhD thesis.

benefit health-care with measurable improvements. We have some promising indications of the fact though. The clinical problem formulation from section 1.2.1 has been the main motivating factor for this PhD project on both a professional level and on a personal level. I have put my whole heart into this project - literally, see figure (13.1).

13.2 Future Work

Each of the individual contributions of chapters 7, 8, 9, 10, and 12 have sections on future work specific to each method. In this section I will look at the future work concerning surgical simulation in a more broad perspective related to the utilization of graphics hardware.

In the following paragraph we return to what graphics hardware was (and for the largest part, still is) about, namely games. Our published methods have been derived from the problem area of heart surgery, but could be applied in computer games as well, coming full circle with respect to what GPU's are really developed for. The two largest manufacturers of GPU's, Nvidia and ATI have also recognized the use of the GPU for physics calculation in games. In Q2 of 2006 Havok and Nvidia have consequently released the Havok FX [87] for rigid body physics, particle physics, and collision effects on the GPU. In a hardware-configuration with two GPUs, one could be used for physics while the other is used for visualization. The release of Havok FX is in direct competition with the physX chip by Ageia (released in May 2006) which can do rigid body physics, fluid simulation, and cloth simulation on a dedicated chip (The physics processing unit, PPU). In all

cases the hardware acceleration of physics based animation is an important line in future game development. The first steps in enabling true deformable materials in games, accelerated by custom hardware have been taken. Combining this with haptic-interaction and highly detailed visualizations would enable more realistic and innovative games. Through this PhD project I have contributed with specific methods and techniques that can be utilized in such future work.

Concerning the general methodology of creating surgical simulators I have two issues to discuss for future work; the high level language support for the type of application we have developed, and the decoupling of visualization and simulation.

Previous research projects [127, 40, 114] have constructed compilers and libraries for high level programming languages that compile to *either* the CPU or the GPU as the target architecture. This allows the programmer to relatively easy test whether a particular computation has the potential to benefit from a GPU based implementation. However, in many scientific computing applications, including surgical simulation, the interaction and visualization is as important as the computation itself. No explicit constructs for visualization or interaction in relation to computations are present in [127, 40, 114], and neither are constructs for an efficient interplay between the CPU and the GPU. Consequently, if the application requires visualization, interaction, or CPU/GPU cooperation, a performance bottleneck resulting from data transfer overhead is easily introduced as data is transported back and forth multiple times between the CPU and GPU. In the last third part of this thesis, I have presented some specific methods to deal with a relatively small range of these issues. For future development of surgical simulation in specific, and scientific computing in general, a framework (in the spirit of the frameworks presented in section 4.4) specifically for applications of scientific computing (including visualization and simulation) supporting the effective utilization of GPU and CPU would be beneficial.

The other issue that I wish to discuss in general is the decoupling of visualization and simulation. In the existing research on surgical simulation based on tissue deformation, there has traditionally been a strong coupling between what is simulated and what is visualized - for obvious reasons. i.e. every visualized vertex corresponds directly to a simulated node. The mapping presented in chapter 9 was motivated by a jagged visualization, but I would advocate that the decoupling of visualization and simulation is valuable as a general strategy in thinking about medical visualization and simulation. Different modalities of spatial-information, transformed by the deformation and incisions of the simulation-engine, potentially hold a range of different kinds of important information to surgeons. As an example; the heart simulator as presented in this thesis shows a visual surface deformed by the soft-tissue simulation. In that case, the shape of the visual surface corresponds closely to the surface of the simulated particles. As a

supplementary tool of information, we could also embed a visual surface representation of the coronary arteries as a separate model from the heart-surface. This would give surgeons additional information on the location of incisions and deformations in relation to the coronary arteries. We could also use the deformations and incisions to transform a set of data-points in 3d representing risk-areas in surgery (a static risk-area is presented in [167]). In that case, we could evaluate the potential risk of every incisions at every point of time in the simulation. Another example is to apply the deformations to the original MRI to get simulated post-operative MRI that could subsequently be compared to real post-operative MRI for validation. A whole taxonomy of spatial information in relation to a series of deformation and topological changes can be set-up; points in space or voxels, surfaces, volumes, and textures.

Finally we return to the actual software-development of this PhD project; a working prototype of a simulator for surgery on pediatric heart defects. In this PhD-project a considerable amount of resources have been put into the development of the simulator to a level of maturity and stability allowing for it to be used by surgeons and other people not involved in the technical development. Specifically through our clinical paper [13], our hands-on demonstration at the SIGGRAPH Emerging Technologies [11, 7], and a planned hands-on demonstration at the Nordic Meeting for Pediatric Cardiology, this level of stability has been demonstrated. The prototype has of course not reached a level of maturity corresponding to a commercial product and technical support specifically is still necessary to create new heart-models. But all taken into account, the next logical step in the development of the project for surgical simulation in pediatric heart surgery is a more thorough clinical evaluation both for pre-operational planning and training. The goal naturally being to release a tool for pediatric surgeons all over the world, for them to benefit from - the end goal being that children with congenital heart defects will have a higher chance of survival.

Bibliography

Publications by the author

[1] Jesper Mosegaard. Realtime cardiac surgery simulation. Master's thesis, Department of Computer Science, University of Aarhus, Denmark, 2003.

[2] Jesper Mosegaard. Lr-spring mass model for cardiac surgical simulation. *Proceedings of Medicine Meets Virtual Reality 12. Studies in Health Technology and Informatics*, 12:256–258, 2004.

[3] Jesper Mosegaard. Parameter optimisation for the behaviour of elastic models over time. *Proceedings of Medicine Meets Virtual Reality 12. Studies in Health Technology and Informatics*, 12:253–255, 2004.

[4] Jesper Mosegaard, Peder Herborg, and Thomas Sangild Sørensen. A GPU accelerated spring mass system for surgical simulation. *Proceedings of Medicine Meets Virtual Reality 13. Studies in Health Technology and Informatics*, 111:342–348, January 2005.

[5] Jesper Mosegaard and Thomas Sangild Sørensen. Gpu accelerated surgical simulators for complex morphology. In *IEEE Virtual Reality*, pages 147–154, 323. IEEE Computer Society, March 2005.

[6] Jesper Mosegaard and Thomas Sangild Sørensen. Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu. In *Proceedings of Eurographics Workshop on Virtual Environments*, volume 11, pages 105–111. Eurographics Association, 2005.

[7] Jesper Mosegaard and Thomas Sangild Sørensen. Technical aspects of the gpu accelerated surgical simulator. In *SIGGRAPH Application Sketches*, Boston, USA, 2006. in press.

[8] Thomas Sangild Sørensen and Jesper Mosegaard. Surgical planning in congenital heart disease by means of real-time medical visualisation and simulation. In *ACM SIGGRAPH Computer Animation Festival*, 2005.

[9] Thomas Sangild Sørensen and Jesper Mosegaard. Haptic feedback for the GPU-based surgical simulator. *Proceedings of Medicine Meets Virtual Reality 14. Studies in Health Technology and Informatics*, 119:523–528, 2006.

[10] Thomas Sangild Sørensen and Jesper Mosegaard. An introduction to gpu accelerated surgical simulation. In Matthias Harders and Gábor Székely, editors, *Biomedical Simulation: Third International Symposium, ISBMS 2006*, volume 4072 of *Lecture Notes in Computer Science*, pages 93–104. Springer Berlin / Heidelberg, 2006.

[11] Thomas Sangild Sørensen and Jesper Mosegaard. Virtual open heart surgery - training complex surgical procedures in congenital heart disease. In *ACM SIGGRAPH Emerging Technologies*, 2006.

[12] Thomas Sangild Sørensen, Jesper Mosegaard, Gerald Greil, Ole Kromann Hansen, and Vibeke E. Hjortdal. Preoperative planning by surgical simulation on patient-specific high-resolution virtual models. In *The Fourth World Congress of Pediatric Cardiology and Cardiac Surgery*, volume 4, page 230, 2005.

[13] T.S. Sørensen, G.F. Greil, O.K. Hansen, and J. Mosegaard. Surgical simulation - a new tool to evaluate surgical incisions in congenital heart disease? *Interactive Cardiovascular and Thoracic Surgery*, 2006.

Publications by other authors

[14] Michael J. Ackerman. Accessing the visible human project. Technical report, Corporation for National Research Initiatives, October 1995.

[15] Ritesh Agarwal, Yogendra Bhasin, Laks Raghupathi, and Venkat Devarajan. Special visual effects for surgical simulation: cauterization, irrigation and suction. *Proceedings of Medicine Meets Virtual Reality 11. Studies in Health Technology and Informatics*, 94:1–3, 2003.

[16] Christos Alexioua, Qiang Chena, Maria Galogavroub, James Gnanapragasamb, Anthony P. Salmonb, Barry R. Keetonb, Marcus P. Hawa, and James L. Monroa. Repair of tetralogy of fallot in infancy with a transventricular or a transatrial approach. *European Journal of Cardiothorac Surgery*, 22(2):174–183, August 2002.

[17] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[18] Serge Beucher and Fernand Meyer. The morphological approach of segmentation: The watershed transformation. In E. Dougherty, editor, *Mathematical Morphology in Image Processing*, chapter 12, pages 43–481. New York: Marcel Dekker, 1992.

[19] Daniel Bielser and Markus H Gross. Open surgery simulation. *Proceedings of Medicine Meets Virtual Reality 02/10. Studies in Health Technology and Informatics*, 85:57–63, 2002.

[20] Daniel Bielser, Volker A. Maiwald, and Markus H. Gross. Interactive cuts through 3-dimensional soft tissue. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pages 31–38. The Eurographics Association and Blackwell Publishers, 1999.

[21] James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292, New York, NY, USA, 1978. ACM Press.

[22] David Blythe. The direct3d 10 system. In *The Proceedings of ACM SIGGRAPH*, 2006.

[23] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[24] Lawrence M. Boxt. Magnetic resonance and computed tomographic evaluation of congenital heart disease. *Journal of Magnetic Resonance Imaging*, 19(6):827–847, June 2004.

[25] Yuri Boykov and Vladimir Kolmogorov. Computing geodesics and minimal surfaces via graph cuts. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision*, page 26, Washington, DC, USA, 2003. IEEE Computer Society.

[26] P. N. Brett, T. J. Parker, A. J. Harrison, T. A. Thomas, and A. Carr. Simulation of resistance forces acting on surgical needles. *Proceedings of the Institution of Mechanical Engineers, part H*, 211(4):335–347, 1997.

[27] M. Bro-Nielsen, D. Helfrick, B. Glass, X. Zeng, and H. Connacher. VR simulation of abdominal trauma surgery. *Proceedings of Medicine Meets Virtual Reality. Studies in Health Technology and Informatics*, 50:117–123, 1998.

[28] M. Bro-Nielsen, J. L. Tasto, R. Cunningham, and G. L. Merril. PreOp endoscopic simulator: a PC-based immersive training system for bronchoscopy. *Proceedings of Medicine Meets Virtual Reality VII. Studies in Health Technology and Informatics*, 62:76–82, 1999.

[29] Morten Bro-Nielsen. Finite element modeling in surgery simulation. *Journal of the IEEE*, 86(3):490–503, 1998.

[30] Morten Bro-Nielsen and Stephane Cotin. Real-time volumetric deformable models for surgery simulation using finite elements and condensation. *Computer Graphics Forum*, 15(3):57–66, 1996.

[31] Joel Brown, Stephen Sorkin, Cynthia Bruyns, Jean-Claude Latombe, Kevin Montgomery, and Michael Stephanides. Real-time simulation of deformable objects: tools and application. In *Proceedings of the Fourteenth Conference on Computer Animation*, pages 228–258, November 2001.

[32] Pat Brown. Nv_float_buffer. `http://www.opengl.org/registry/specs/NV/float_buffer.txt`, 2002.

[33] Pat Brown. Arb_vertex_program. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt`, September 2004.

[34] Pat Brown. NV_vertex_program3. `http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_vertex_program3.txt`, 2004. Accessed the 24th of July 2006.

[35] Pat Brown and Mark Kilgard. Gl_nv_vertex_program2. `http://www.opengl.org/registry/specs/NV/vertex_program2.txt`, 2002.

[36] Pat Brown and Mark J. Kilgard. NV_fragment_program. `http://www.opengl.org/registry/specs/NV/fragment_program.txt`, May 2005. Accessed the 23th of July 2006.

[37] Pat Brown and Eric Werness. NV_fragment_program2. `http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_fragment_program2.txt`, 2004. Accessed the 23th of July 2006.

[38] Cynthia Bruyns and Mark P. Ottensmeyer. Measurements of soft-tissue mechanical properties to support development of a physically based virtual animal model. In *MICCAI '02: Proceedings of the 5th International Conference on Medical Image Computing and Computer-Assisted Intervention-Part I*, pages 282–289, London, UK, 2002. Springer-Verlag.

[39] Ian Buck, Kayvon Fatahalian, and Pat Hanrahan. Gpubench: Evaluating gpu performance for numerical and scientific applications. In

General Purpose Computing on Graphics Processors, ACM Workshop 2004, pages C–20, 2004.

[40] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[41] Ian Buck, Naga Govindaraju, Mark Harris, Jens Krueger, Aaron Lefohn, David Luebke, Tim Purcell, and Cliff Woolley. Siggraph 2005 gpgpu course. `http://www.gpgpu.org/s2005/`, August 2005.

[42] Ian Buck, Aaron Lefohn, Patrick McCormick, John Owens, Tim Purcell, and Robert Strzodka. Ieee visualization 2005 tutorial. `http://www.gpgpu.org/vis2005/`, October 2005.

[43] Ian Buck and Tim Purcell. A toolkit for computation on gpus. In Randima Fernando, editor, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 37, pages 621–636. Addison Wesley, 2004.

[44] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[45] M. Cenk Cavusoglu, Tolga G Göktekin, and Frank Tendick. GiPSi:a framework for open source/open architecture software development for organ-level surgical simulation. *IEEE Trans Inf Technol Biomed*, 10(2):312–322, April 2006.

[46] Cincinatti Children's Hospital Medical Center. The heart center, encyclopedi. `http://www.cincinnatichildrens.org/health/heart-encyclopedia/`, April 2006.

[47] Wei Chen, Huagen Wan, Hongxin Zhang, Hujun Bao, and Qunsheng Peng. Interactive collision detection for complex and deformable models using programmable graphics hardware. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 10–15, New York, NY, USA, 2004. ACM Press.

[48] Yoo-Joo Choi, Young J. Kim, and Myoung-Hee Kim. Rapid pairwise intersection tests using programmable gpus. *Vis. Comput.*, 22(2):80–89, 2006.

[49] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press.

[50] Stephane Cotin, Herve Delingette, and Nicholas Ayache. Real-time elastic deformations of soft tissues for surgery simulation. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):62–73, 1999.

[51] Stephane Cotin, Herve Delingette, and Nicholas Ayache. A hybrid elastic model allowing real-time cutting, deformations and force-feedback for surgery training and simulation. *The Visual Computer*, 16(8):437–452, 2000.

[52] Stephane Cotin, Christian Duriez, Julien Lenoir, Paul Neumann, and Steven Dawson. New approaches to catheter navigation for interventional radiology simulation. pages 534–542, Palm Springs, California, USA, 26-30 october 2005.

[53] Stephane Cotin, Paul Neumann, Hervé Delingette, Cenk Cavusoglu, Frank Tendick, Susann Luperfoy, Christophe Chaillou, Philippe Meseure, Matthias Harders, Emmanuel Promayon, Kenneth Curley, Harvey Magee, Xunlei Wu, and Kevin Montgomery. Collaborative development of an open framework for medical simulation. Technical report, 2004.

[54] Steven Cover, Norberto Ezquerra, James O'Brien, Richard Rowe, Thomas Gadacz, and Ellen Palm. Interactively deformable models for surgery simulation. *IEEE Comput. Graph. Appl.*, 13(6):68–75, 1993.

[55] Francois Boux de Casson and Christian Laugier. Modeling the dynamics of a human liver for a minimally invasive surgery simulator. In *MICCAI '99: Proceedings of the Second International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 1156–1165, London, UK, 1999. Springer-Verlag.

[56] Herve Delingette. Towards realistic soft tissue modeling in medical simulation. In *Proceedings of the IEEE: Special Issue on Surgery Simulation*, pages 512–523, 1998.

[57] Herve Delingette, Stephane Cotin, and Nicholas Ayache. A hybrid elastic model allowing real-time cutting, deformations and force-feedback for surgery training and simulation. In *CA '99: Proceedings of the Computer Animation*, page 70, Washington, DC, USA, 1999. IEEE Computer Society.

[58] Chen Ding and Ken Kennedy. The memory bandwidth bottleneck and its amelioration by a compiler. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 181, Washington, DC, USA, 2000. IEEE Computer Society.

[59] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee Computer Science Technical Report, 2006.

[60] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[61] John Dubinski. The merger of the milky way and andromedia galaxies. `http://www.cita.utoronto.ca/~dubinski/tflops/`, January 2001.

[62] Kenny Erleben, Jon Sporring, Knud Henriksen, and Henrik Dohlmann. *Physics-Based Animation*. Charles River Media, 2005.

[63] Cass Everitt, Ashu Rege, and Cem Cebenoyan. hardware shadow mapping. Technical report, Nvidia, 2001.

[64] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, New York, NY, USA, 2004. ACM Press.

[65] Randima Fernando, editor. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison Wesley, 2004.

[66] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[67] T.G. Flohr, S. Schaller S, K. Stierstorfer, H. Bruder, B.M. Ohnesorge, and U.J. Schoepf. Multi-detector row ct systems and image-reconstruction techniques. *Radiology*, 235(3):756–773, June 2005.

[68] Gonzalo Frasca. Simulation 101: Simulation versus representation. `http://www.ludology.org/articles/sim1/simulation101.html`, 2001. Accessed the 22nd of July 2006.

[69] M.P. Fried, R. Satava, S. Weghorst, A.G. Gallagher, C. Sasaki D. Ross, M. Sinanan, J.I. Uribe, M. Zeltsan, H. Arora H, and H. Cuellar. Identifying and reducing errors with surgical simulation. *Quality and Safety in Healthcare*, 13 Suppl 1:i19–i26, October 2004.

[70] Sarah Frisken-Gibson. Using linked volumes to model object collisions, deformation, cutting, carving, and joining. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):333–348, 1999.

[71] Sarah Frisken-Gibson. Volume deformation: Modeling shape changes in sampled volumes. In *Course Notes 41 (Volume Graphics)*, 1999.

[72] A.G. Gallagher, E.M. Ritter, H. Champion, G. Higgins, M.P. Fried, G. Moses, C.D. Smith, and R.M. Satava. Virtual reality simulation for the operating room: proficiency-based training as a paradigm shift in surgical skills training. *Annals of Surg*, 241(2):364–372, February 2005.

[73] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[74] Fabio Ganovelli and Carol O'Sullivan. Animating cuts with on-the-fly re-meshing. In *Eurographics 2001*, pages 243–247, 2001.

[75] Paul Gasson, Rudy J. Lapeer, and Alf D. Linney. Modelling techniques for enhanced realism in an open surgery simulation. In *IV '04: Proceedings of the Information Visualisation, Eighth International Conference on (IV'04)*, pages 73–78, Washington, DC, USA, 2004. IEEE Computer Society.

[76] Dominik Göddeke. Gpgpu::tutorials. `http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/index.html`, May 2006.

[77] Joachim Georgii, Florian Echtler, and Rudiger Westermann. Interactive simulation of deformable bodies on gpus. In *Simulation and Visualisation 2005*, pages 247–258, March 2005.

[78] Joachim Georgii and Rudiger Westermann. Mass-spring systems on the gpu. *Simulation Practice and Theory, Elsevier Science*, July 2005.

[79] Sarah Gibson, Joseph Samosky, and Andrew Mor. Simulating arthroscopic knee surgery using volumetric object representations, real-time volume rendering and haptic feedback. In *Proceedings of CVRMed-MRCAS*, pages 369–378, 1997.

[80] Sarah F. Gibson. 3d chainmail: a fast algorithm for deforming volumetric objects. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 149–154, New York, NY, USA, 1997. ACM Press.

[81] Sarah F. F. Gibson and Brian Mirtich. A survey of deformable modeling in computer graphics. Technical report, Mitsubishi Electric Research Lab., Cambridge,, 1997.

[82] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 102–111, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[83] Mounna Gor, Rory McCloy, Robert Stone, and Anthony Smith. Virtual reality laparoscopic simulator for assessment in gynaecology. *BJOG: An International Journal of Obstetrics and Gynaecology*, 110(2):181–187, 2003.

[84] P. J. Gorman, A. H. Meier, and T. M. Krummel. Simulation and virtual reality in surgical education: real or unreal? *Archives of Surg*, 134(11):1203–1208, November 1999.

[85] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. Quick-cullide: Fast inter- and intra-object collision culling using graphics hardware. In *VR '05: Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality*, pages 59–66, 319, Washington, DC, USA, 2005. IEEE Computer Society.

[86] Simon Green. Opengl shader tricks. In *Game Developers Conference*, 2003.

[87] Simon Green and Mark Harris. Physics simulation on nvidia gpus. In *Game Developers Conference*, 2006.

[88] Kim Vang Hansen and Ole Vilhelm Larsen. Using region-of-interest based finite element modeling for brain-surgery simulation. In *MICCAI '98: Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 305–316, London, UK, 1998. Springer-Verlag.

[89] Mark J. Harris. General-purpose computation using graphics hardware webpage. `http://www.gpgpu.org`, May. Accessed the 14th of July 2006.

[90] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[91] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In

HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[92] Dirk Helbing, Illes Farkas, and Tamas Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, September 2000.

[93] Bradley M. Hemminger, Paul L. Molina, Thomas M. Egan, Frank C. Detterbeck, Keith E. Muller, Christopher S. Coffey, and Joseph K. T. Lee. Assessment of real-time 3d visualization for cardiothoracic diagnostic evaluation and surgery planning. *Journal of Digital Imaging*, 19:145–153, April 2005.

[94] Johannes Hirche, Alexander Ehlert, Stefan Guthe, and Michael Doggett. Hardware accelerated per-pixel displacement mapping. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 153–158, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[95] Richard Holbrey. *Virtual Suturing for Training in Vascular Surgery.* PhD thesis, School of Computing, University of Leeds, 2004.

[96] Richard P Holbrey and Andrew J Bulpitt. Metrics and motion analysis for assessment of surgical skills. *Proceedings of Medicine Meets Virtual Reality 11. Studies in Health Technology and Informatics*, 94:124–126, 2003.

[97] Arun Holden. The beating heart of virtual engineering. *Scientific Computing World*, Oct/Nov:26–28, 2000.

[98] John Hu, Chu-Yin Chang, Neil Tardella, Janey Pratt, and James English. Effectiveness of haptic feedback in open surgery simulation and training systems. *Proceedings of Medicine Meets Virtual Reality 14. Studies in Health Technology and Informatics*, 119:213–218, 2006.

[99] Thomas J. R. Hughes. *The Finite Element Method - Linear Static and Dynamic Finite Element Analysis.* Dover, 2000.

[100] Children's Heart Institute. Heart defects: Hypoplastic right ventricle. `http://childrensheartinstitute.org/educate/defects/hypor1.htm`, June 2006.

[101] Thomas Jakobsen. Advanced character physics. Technical report, Gamasutra, 2003.

[102] Jeff Juliano and Jeremy Sandmel. Ext_framebuffer_object. `http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt`, 2005.

[103] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *Proc. ICAT*, pages 205–208, 2001.

[104] Christoph Kaufmann, Scott Zakaluzny, and Alan Liu. First steps in eliminating the need for animals and cadavers in advanced trauma life support. In *MICCAI '00: Proceedings of the Third International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 618–623, London, UK, 2000. Springer-Verlag.

[105] Erwin Keeve, Sabine Girod, and Bernd Girod. Craniofacial surgery simulation. In *VBC '96: Proceedings of the 4th International Conference on Visualization in Biomedical Computing*, pages 541–546, London, UK, 1996. Springer-Verlag.

[106] Amy E Kerdok, Stephane M Cotin, Mark P Ottensmeyer, Anna M Galea, Robert D Howe, and Steven L Dawson. Truth cube: establishing physical standards for soft tissue simulation. *Med Image Anal*, 7(3):283–291, September 2003.

[107] Uwe G. Kühnapfel, Christian Kuhn, M. Hübner, H.G. Krumm, and B. Neisius. Cad-based simulation and modelling for endoscopic surgery. In *Proceedings of MedTech, SMIT'94,*, October 1994.

[108] Mark J. Kilgard. Nv_vertex_program. `http://oss.sgi.com/projects/ogl-sample/registry/NV/vertex_program.txt`, February 2004.

[109] David R. Kincaid and E. Ward Cheney. *Numerical Analysis : Mathematics of Scientific Computing (3rd)*. Brooks Cole, 2001.

[110] Dale Kirkland, Pat Brown, Jon Leech, Rob Mace, and Brian Paul. Arb_texture_float. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_float.txt`, 2004.

[111] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[112] Yuri Kryachko. Using vertex texture displacement for realistic water rendering. In Randima Fernando, editor, *GPU Gems 2*, chapter 18. Addison-Wesley, 2005.

[113] Uwe G. Kuhnapfel, Huseyin Kemal Cakmak, and Heiko Maass. Endoscopic surgery training using virtual reality and deformable tissue simulation. *Computers & Graphics*, 24:671–682, 2000.

[114] Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph.*, 25(1):60–99, 2006.

[115] K.S. Lehmann, J.P. Ritz, H. Maass, H.K. Cakmak, U.G. Kuehnapfel, C.T. Germer, G. Bretthauer, and H.J. Buhr. A prospective randomized study to test the transfer of basic psychomotor skills from virtual reality to physical reality in a comparable training setting. *Ann Surg*, 241(3):442–449, March 2005.

[116] DR1 Lægens Bord. Da mikkels hjerte blev sat i stå. 28. apr. 2005 19:30 on the television channel DR1, available on the Internet `http://www.dr.dk/DR1/laegen/Programmer/2005/050428/20050824110522.htm`, June 2006.

[117] Benj Lipchak. Arb_fragment_program. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt`, October 2003.

[118] A. Liu, C. Kaufmann, and T. Ritchie. A computer-based simulator for diagnostic peritoneal lavage. *Proceedings of Medicine Meets Virtual Reality 2001. Studies in Health Technology and Informatics*, 81:279–285, 2001.

[119] Alan Liu, Christoph Kaufmann, and Daigo Tanaka. An architecture for simulating needle-based surgical procedures. In *MICCAI '01: Proceedings of the 4th International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 1137–1144, London, UK, 2001. Springer-Verlag.

[120] Alan Liu, Frank Tendick, Kevin Cleary, and Christoph Kaufmann. A survey of surgical simulation: applications, technology, and education. *Presence: Teleoper. Virtual Environ.*, 12(6):599–614, 2003.

[121] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[122] Rob Mace. Ati_texture_float. `http://www.opengl.org/registry/specs/ATI/texture_float.txt`, 2002.

[123] J. Harvey Magee. Validation of medical modeling & simulation training devices and systems. *Proceedings of Medicine Meets Virtual Reality 11. Studies in Health Technology and Informatics*, 94:196–198, 2003.

[124] Francis T. Makowski and Luciano Mariella. Computer simulation helps engineers improve ferrari formula one aerodynamics. Technical report, Fluent Inc., 2001.

[125] Frdric Mazzella, Kevin Montgomery, and Jean claude Latombe. The forcegrid: A buffer structure for haptic interaction with virtual elastic objects. In *Proceedings. ICRA '02. IEEE International Conference on Robotics and Automation*, volume 1, pages 939– 946, 2002.

[126] R. McCloy and R. Stone. Science, medicine, and the future. Virtual reality in surgery. *BMJ*, 323(7318):912–915, October 2001.

[127] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters Ltd, 2004.

[128] U. Meier, O. López, C. Monserrat, M. C. Juan, and M. Alcañiz. Real-time deformable models for surgery simulation: a survey. *Comput Methods Programs Biomed*, 77(3):183–197, March 2005.

[129] Microsoft. Directx developer center. `http://msdn.microsoft.com/directx/`. Accessed the 17th of July 2006.

[130] Tomas Moller, Eric Haines, and Tomas Akenine-Moller. *Real-Time Rendering (2nd Edition)*. AK Peters, Ltd., July 2002.

[131] J. Montagnat, H. Delingette, and N. Ayache. A review of deformable surfaces: topology, geometry and deformation. *Image and Vision Computing*, 19(14):1023–1040, December 2001.

[132] Kevin Montgomery. Enabling technologies in surgical simulation: When will the future be here. Presented at TATRIC's third annual Principal Investigators Review, 2003.

[133] Kevin Montgomery, Cynthia Bruyns, Joel Brown, Stephen Sorkin, Frederic Mazzella, Guillaume Thonier, Arnaud Tellier, Benjamin Lerman, and Anil Menon. Spring: a general framework for collaborative, real-time surgical simulation. *Proceedings of Medicine Meets Virtual Reality 02/10. Studies in Health Technology and Informatics*, 85:296–303, 2002.

[134] Kevin Montgomery and Cynthia D Bruyns. Generalized interactions using virtual tools within the spring framework: cutting. *Proceedings of Medicine Meets Virtual Reality 02/10. Studies in Health Technology and Informatics*, 85:79–85, 2002.

[135] Kevin Montgomery and Cynthia D Bruyns. Generalized interactions using virtual tools within the spring framework: probing, piercing,

cauterizing and ablating. *Proceedings of Medicine Meets Virtual Reality 02/10. Studies in Health Technology and Informatics*, 85:74–78, 2002.

[136] Kevin Montgomery, LeRoy Heinrichs, Cynthia Bruyns, Simon Wildermuth, Christopher Hasser, Stephanie Ozenne, and David Bailey. Surgical simulator for hysteroscopy: a case study of visualization in surgical training. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 449–452, Washington, DC, USA, 2001. IEEE Computer Society.

[137] Andrew B. Mor. *Progressive cutting with minimal new element creation of soft tissue models for interactive surgical simulation.* PhD thesis, Robotics Institute, Carnegie Mellon University, 2001. Chair-Takeo Kanade.

[138] S. Mottl-Link, T. Boettger, J.J. Krueger JJ, U. Rietdorf, B. Schnackenburg, P. Ewert, F. Berger, E. Nagel, H.P. Meinzer, A. Juraszek, T. Kuehne, and I. Wolf. Images in cardiovascular medicine. cast of complex congenital heart malformation in a living patient. 112:e356–e357, 2005.

[139] U. Myhre, B.W Duncan, R.B. Mee, R. Joshi, S.G. Seshadri, O. Herrera-Verdugo, and G.L. Rosenthal. Apical right ventriculotomy for closure of apical ventricular septal defects. *Ann Thorac Surg*, 78(1):204–208, July 2004.

[140] Megumi Nakao. *Cardiac Surgery Simulation with Active Interaction and Adaptive Physics-based Modeling.* PhD thesis, Dept. of Medical Informatics, Kyoto Univ. Hospital, 2003.

[141] M. P. Nash and P. J. Hunter. Heart mechanics using mathematical modelling. *Proceedings. of the Second New Zealand Postgraduate Conerence. for Engineering and Technology Students*, 1996.

[142] Jean-Christophe Nebel. Soft tissue modeling from 3d scanned data. In *DEFORM/AVATARS*, pages 85–97, 2000.

[143] L. P. Nedel and D. Thalmann. Real time muscle deformations using mass-spring systems. In *CGI '98: Proceedings of the Computer Graphics International 1998*, page 156, Washington, DC, USA, 1998. IEEE Computer Society.

[144] The Canadian Adult Congenital Heart (CACH) Network. Adult congenital heart disease glossary. `http://www.cachnet.org/achd_index.html`, June 2006.

[145] H. W. Nienhuys and A.F. van der Stappen. *A Delaunay approach to interactive cutting in triangulated surfaces*, chapter Session II, pages 113–129. Springer-Verlag, 2004.

[146] Han-Wen Nienhuys and A. Frank van der Stappen. A surgery simulation supporting cuts and finite element deformation. In *MICCAI '01: Proceedings of the 4th International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 145–152, London, UK, 2001. Springer-Verlag.

[147] OpenTissue. Opensource Project, Physical based Animation and Surgery Simulation. http://www.opentissue.org.

[148] Steve Owen. A survey of unstructured mesh generation technology. `http://www.andrew.cmu.edu/user/sowen/survey/`. Accessed on the 24th of July 2006.

[149] 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. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[150] Jin Seo Park, Min Suk Chung, Sung Bae Hwang, Yong Sook Lee, Dong-Hwan Har, and Hyung Seon Park. Visible Korean human: improved serially sectioned images of the entire body. *IEEE Trans Med Imaging*, 24(3):352–360, March 2005.

[151] A. M. Pearson, A. G. Gallagher, J. C. Rosser, and R. M. Satava. Evaluation of structured and quantitative training methods for teaching intracorporeal knot tying. *Surg Endosc*, 16(1):130–137, January 2002.

[152] pediheart.org. pediheart.org - practitioners site - double outlet right ventricle. `http://www.pediheart.org/practitioners/defects/ventriculoarterial/DORV.htm`, June 2006.

[153] Matt Pharr, editor. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005.

[154] G. Picinbono, J. Lombardo, H. Delingette, and N. Ayache. Improving realism of a surgery simulator: Linear anisotropic elasticity, complex interactions and force extrapolation. Technical report, INRIA, October 2000.

[155] G. Picinbono, J. Lombardo, H. Delingette, and N. Ayache. Improving realism of a surgery simulator: Linear anisotropic elasticity, com-

plex interactions and force extrapolation. *Journal of Visualization and Computer Animation*, 14(3):147–167, 2002.

[156] Pixar. *The RenderMan Interface, V3.1.* 1989. PIX 99:1 1.P-Ex.

[157] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 159–170. ACM Press, 2001.

[158] Xavier Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In Wayne A. Davis and Przemyslaw Prusinkiewicz, editors, *Graphics Interface '95*, pages 147–154. Canadian Human-Computer Communications Society, 1995.

[159] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, 2002.

[160] R.S. Razavi, D.L. Hill, V. Muthurangu, M.E. Miquel, A.M. Taylor, S. Kozerke, and E.J. Baker. Three-dimensional magnetic resonance imaging of congenital cardiac anomalies. *Cardiol Young*, 13(5):461–465, October 2003.

[161] J. N. Reddy. Theory of elasticity - supplementary notes. `ceprofs.tamu.edu/jreddy/MEMA601/LectureNotes.pdf`, 2003.

[162] Charles Y Ro, Ioannis K Toumpoulis, Robert C Ashton, Tony Jebara, Caroline Schulman, George J Todd, Joseph J Derose, and James J McGinty. The LapSim: a learning environment for both experts and novices. *Proceedings of Medicine Meets Virtual Reality 13. Studies in Health Technology and Informatics*, 111:414–417, 2005.

[163] Richerd A. Robb. *Biomedical imaging, visualization, and analysis.* Wiley-Liss, 1999.

[164] Randi J. Rost. *OpenGL Shading Language.* Addison Wesley, 2004.

[165] Martin Rumpf and Robert Strzodka. Using graphics cards for quantized FEM computations. In *Proceedings of IASTED Visualization, Imaging and Image Processing Conference (VIIP'01)*, pages 193–202, 2001.

[166] Tobias Salb, Jakob Brief, Oliver Burgert, Stefan Haßfeld, and Rüdiger Dillmann. Haptic based risk potential mediation for surgery simulation. In *1. International Workshop on Haptic Devices in Medical Application (HDMA), within the scope of CARS conference*, 1999.

[167] Tobias Salb, Tim Weyrich, and Rüdiger Dillmann. Preoperative planning and training simulation for risk reducing surgery. In *International Training and Simulation Conference (ITEC)*, 1999.

[168] Kenneth Salisbury, Francois Conti, and Federico Barbagli. Haptic rendering: Introductory concepts. *IEEE Comput. Graph. Appl.*, 24(2):24–32, 2004.

[169] R. M. Satava. Medical virtual reality. The current status of the future. *Proceedings of Healthcare in the Information Age. Studies in Health Technology and Informatics*, 29:100–106, 1996.

[170] Richard Satava. Report on the metrics for objective assessment of surgical skills workshop. Proceedings of the Telemedicine and Advanced Technology Research Center (TATRC) 3rd Annual Advanced Technology Portfolio Review. Available at: `www.tatrc.org/website_mmvr2003/presentations/satava_files/frame.htm`, 2003. Accessed the 11th of July 2006.

[171] Neal E Seymour, Anthony G Gallagher, Sanziana A Roman, Michael K O'Brien, Vipin K Bansal, Dana K Andersen, and Richard M Satava. Virtual reality training improves operating room performance: results of a randomized, double-blinded study. *Ann Surg*, 236(4):458–63; discussion 463–4, October 2002.

[172] Mads S Sørensen, Andy B Dobrzeniecki, Per Larsen, Thomas Frisch, Jon Sporring, and Tron A Darvann. The visible ear: a digital image library of the temporal bone. *ORL J Otorhinolaryngol Relat Spec*, 64(6):378–381, 2002.

[173] Thomas Sangild Sørensen, Hermann Körperich, Gerald F Greil, Joachim Eichhorn, Peter Barth, Hans Meyer, Erik Morre Pedersen, and Philipp Beerbaum. Operator-independent isotropic three-dimensional magnetic resonance imaging for morphology in congenital heart disease: a validation study. *Circulation*, 110(2):163–169, July 2004.

[174] Thomas Sangild Sørensen, Erik Morre Pedersen, Ole Kromann Hansen, and Keld Sorensen. Visualization of morphological details in congenitally malformed hearts: virtual three-dimensional reconstruction from magnetic resonance imaging. *Cardiol Young*, 13(5):451–460, October 2003.

[175] Stanford University, Computer Graphics Laboratory. Dragon, the stanford 3d scanning repository. `http://graphics.stanford.edu/data/3Dscanrep`.

[176] G. Stellin, M. Padalino, O. Milanesi, M. Rubino, D. Casarotto, P.R. Van, and P.S. Van. Surgical closure of apical ventricular septal defects through a right ventricular apical infundibulotomy. *Ann Thorac Surg*, 69(2):597–601, February 2000.

[177] Robert Strzodka, Michael Doggett, and Andreas Kolb. Scientific computation for simulations on programmable graphics hardware. *Simulation Practice & Theory*, 13:8:667–680, 2005.

[178] Hariom Sur, Alessandro Faraci, and Fernando Bello. Validation of soft tissue properties in surgical simulation with haptic feedback. *Proceedings of Medicine Meets Virtual Reality 12. Studies in Health Technology and Informatics*, 98:382–384, 2004.

[179] C. Sutton, R. McCloy, A. Middlebrook, P. Chater, M. Wilson, and R. Stone. MIST VR. A laparoscopic surgery procedures trainer and evaluator. *Proceedings of Global Healthcare Grid. Studies in Health Technology and Informatics*, 39:598–607, 1997.

[180] G. Székely, C. Brechbühler, R. Hutter, A. Rhomberg, and P. Schmid. Modeling of soft tissue deformation for laparoscopic surgery simulation. In *MICCAI '98: Proceedings of the First International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 550–561, London, UK, 1998. Springer-Verlag.

[181] D. Terzopoulos and K. Waters. Analysis and synthesis of facial image sequences using physical and anatomical models. volume 15, pages 569–579, Washington, DC, USA, 1993. IEEE Computer Society.

[182] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 205–214, New York, NY, USA, 1987. ACM Press.

[183] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[184] V.T. Tsang, T.Y. Hsia, R.W. Yates, and R.H. Anderson. Surgical repair of supposedly multiple defects within the apical part of the muscular ventricular septum. *Ann Thorac Surg*, 73(1):58–62, January 2002.

[185] M. Ursino, J. L. Tasto, B. H. Nguyen, R. Cunningham, and G. L. Merril. CathSim: an intravascular catheterization simulator on a PC. *Proceedings of Medicine Meets Virtual Reality VII. Studies in Health Technology and Informatics*, 62:360–366, 1999.

[186] P.S. Van, J.E. Maye Jr, N.B. Berman, M.F. Flanagan, T. Geva, and P.R Van. Apical ventricular septal defects: follow-up concerning anatomic and surgical considerations. *Ann Thorac Surg*, 73(1):48–56, January 2002.

[187] Loup Verlet. Computer experiments on classical fluids, thermodynamical properties of lennard-jones molecules. *Physical Review*, 159:98–103, 1967.

[188] Luc Vincent and Pierre Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(6):583–598, 1991.

[189] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Trans. Graph.*, 22(3):334–339, 2003.

[190] R. W. Webster, D. I. Zimmerman, B. J. Mohler, M. G. Melkonian, and R. S. Haluck. A prototype haptic suturing simulator. *Proceedings of Medicine Meets Virtual Reality 2001. Studies in Health Technology and Informatics*, 81:567–569, 2001.

[191] J. Weickert. Nonlinear diffusion filtering. In B. Jähne, H. Haußecker, and P. Geißler, editors, *Handbook on Computer Vision and Applications, Vol. 2: Signal Processing and Pattern Recognition*, pages 423–450. Academic Press, 1999.

[192] Wikipedia. Geforce256. `http://en.wikipedia.org/wiki/GeForce_256`. Accessed the 14th of July 2006.

[193] Wikipedia. Graphics processing unit. `http://en.wikipedia.org/wiki/Graphics_processing_unit`. Accessed the 14th of July 2006.

[194] Wingo Sai-Keung Wong and George Baciu. Gpu-based intrinsic collision detection for deformable surfaces: Collision detection and deformable objects. *Comput. Animat. Virtual Worlds*, 16(3-4):153–161, 2005.

[195] Mason Woo, Davis, and Mary Beth Sheridan. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[196] Wen Wu and Pheng Ann Heng. A hybrid condensed finite element model with gpu acceleration for interactive 3d soft tissue cutting: Research articles. *Comput. Animat. Virtual Worlds*, 15(3-4):219–227, 2004.

[197] Hugh D. Young, Roger A. Freedman, T. R. Sandin, and A. Lewis Ford. *University Physics*. Addison Wesley, 1999.

[198] Cyril Zeller. Cloth simulation on the gpu. In *SIGGRAPH application sketch*, July 2005.

[199] Shao-Xiang Zhang, Pheng-Ann Heng, Zheng-Jin Liu, Li-Wen Tan, Ming-Guo Qiu, Qi-Yu Li, Rong-Xia Liao, Kai Li, Gao-Yu Cui, Yan-Li Guo, Xiao-Ping Yang, Guang-Jiu Liu, Jing-Lu Shan, Ji-Jun Liu, Wei-Guo Zhang, Xian-Hong Chen, Jin-Hua Chen, Jian Wang, Wei Chen, Ming Lu, Jian You, Xue-Li Pang, Hong Xiao, and Yong-Ming Xie. Creation of the Chinese visible human data set. *Anat Rec B New Anat*, 275(1):190–195, December 2003.

[200] János Zátonyi, Rupert Paget, Gábor Székely, Markus Grassi, and Michael Bajka. Real-time synthesis of bleeding for virtual hysteroscopy. *Med Image Anal*, 9(3):255–266, June 2005.