DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

DESIGN AND DEVELOPMENT OF A THREE DIMENSIONAL AUGMENTED REALITY SYSTEM AIMING MEDICAL AND ENGINEERING APPLICATIONS

by
Ruha Uğraş ERDOĞAN

October, 2010
İZMİR

DESIGN AND DEVELOPMENT OF A THREE DIMENSIONAL AUGMENTED REALITY SYSTEM AIMING MEDICAL AND ENGINEERING APPLICATIONS

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science
in Electrical and Electronics Engineering, Electrical and Electronics
Engineering Program

by
Ruha Uğraş ERDOĞAN

October, 2010
İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"DESIGN AND DEVELOPMENT OF A THREE DIMENSIONAL AUGMENTED REALITY SYSTEM AIMING MEDICAL AND ENGINEERING APPLICATIONS"** completed by **RUHA UĞRAŞ ERDOĞAN** under supervision of **ASST. PROF. DR. AHMET ÖZKURT** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Ahmet ÖZKURT

Prof. Dr. Mustafa SABUNCU

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGMENTS

The author would like to appreciate Asst. Prof. Dr. Ahmet ÖZKURT for his valuable and important theoretical, technical support and suggestions for the software development process in addition to his management of the research project period. Without those directions and management, this thesis would be very hard to be accomplished and it would be hard to survive from awful periods. The author would like to appreciate Prof. Dr. Cüneyt GÜZELİŞ for the support during the project preparation stage and for the guidance to the graphics processing workshops GAG’09 and GAG’10 respectively. Those workshops were definitely two of the important sources of theoretical and practical knowledge in the field that gave direction to the thesis work. The author would like to appreciate Prof. Dr. Murat ÖZGÖREN, Assoc. Prof. Dr. Adile ÖNİZ, Dr. Onur BAYAZIT and Dokuz Eylül University Faculty of Medicine the Department of Biophysics for their patience, sincerity and most importantly for their helps while I try to stand up again. This M.Sc thesis was completed in the scope of the M.Sc graduate level study at the Graduate School of Natural and Applied Sciences of Dokuz Eylül University. The author would like to appreciate the Graduate School of Natural and Applied Sciences of Dokuz Eylül University for their support during the M.Sc thesis period.

This project is supported in the scope of Dokuz Eylül University Scientific Research Project with Project No. 2008.KB.FEN.027 (Dokuz Eylül Üniversitesi Bilimsel Araştırma Projesi (BAP), Proje No. 2008.KB.FEN.027.). VESTEL supported the establishment of the computer graphics and virtual reality laboratory with a LCD panel.

Ruha Uğraş ERDOĞAN

DESIGN AND DEVELOPMENT OF A THREE DIMENSIONAL AUGMENTED REALITY SYSTEM AIMING MEDICAL AND ENGINEERING APPLICATIONS

ABSTRACT

Three dimensional modeling and simulation software are becoming more widespread in medical applications. Enabling the user to view the 3D models of biological tissues and materials, to interact with the models with the ability to observe the reaction of the models to different force loading conditions in a virtual environment are the main properties of these kind of software. In addition to these properties, the graphical user interface enables the user to easily interact with the software and access its properties. These software specifications give an opportunity to understand the physical and mathematical reasons of dynamical processes in addition to presenting a visual learning environment to the researchers in not only in medicine but also in different fields of science. Considering above, the development of software which will immerse the user into a virtual environment providing an opportunity to observe and to interact with the anatomical models is aimed. Additionally, the software system will be able to simulate the responses of the models depending on different force loading conditions and material properties in real time. Additionally, the development of the necessary hardware platform has been aimed.

Keywords: Real time computer graphics, virtual reality and human interaction, 3-D medical simulation, numerical methods for rigid and elastic object modeling, real time collision detection methods, force and penetration depth computations, graphics processing unit programming, Cg - C for Graphics, HLSL, GLSL, NVIDIA CUDA.

TIP VE MÜHENDİSLİK UYGULAMALARINI AMAÇLAYAN ÜÇ BOYUTLU ARTTIRILMIŞ GERÇEKLİK SİSTEMİ TASARIMI VE GELİŞTİRİLMESİ

ÖZ

Üç boyutlu modelleme ve simülasyon yazılımlarının kullanımı, tıbbi uygulamalarda gün geçtikçe artmaktadır. Bu yazılımların sahip oldukları önemli özelliklerin başında üç boyutlu biyolojik doku ve materyal modellerinin incelenebilmesine, etkileşim kurulabilmesine, farklı yük bindirimleri altındaki davranışlarının üç boyutlu sanal bir ortam içerisinde gözlenebilmesine imkan tanımaları gelmektedir. Bu özelliklere ek olarak, sunulan grafiksel kullanıcı arayüzü, kişinin yazılım ile kolay bir şekilde iletişim kurmasina ve özelliklerine ulaşmasına izin vermektedir. Bu yazılım nitelikleri, sadece sağlık bilimlerinde değil farkli bilim alanlarinda çalışan tüm araştırmacılara görsel bir öğrenme imakanı ve dinamik süreçlerin fiziksel ve matematiksel nedenlerini anlama olanağı sunmaktadır. Bu noktadan yola çıkarak, kullanıcının sanal, üç boyutlu bir ortam içerisinde bulunmasını sağlayacak; istediği anatomik modeli, üç boyutlu ortam içerisinde gerçek zamanlı olarak incelenmesine ve onunla etkileşim kurmasına izin verecek bir yazılım geliştirmek amaçlanmıştır. Geliştirilecek yazılımın belirtilen amaca ek olarak, modellerin materyal özelliklerine bağlı olarak farklı yük durumları altındaki tepkilerinin üç boyutlu sanal bir ortam içerisinde benzetimini gerçek zamanlı olarak yapabilmesi ve ayrıca gerekli donanımsal altyapının hazırlanması amaçlanmıştır.

Anahtar kelimeler: Gerçek zamanlı bilgisayar grafikleri, sanal gerçeklik ve kullanici etkileşimi, tıbbi benzetim, rijit ve elastik nesneler için numerik modelleme, eş zamanlı çarpışma belirleme yöntemleri, kuvvet ve girişim derinliği hesabı, grafik işlem birimi programlama, Cg - C for Graphics, HLSL, GLSL, NVIDIA CUDA.

CONTENTS

Page

M.Sc THESIS EXAMINATION RESULT FORM ... ii
ACKNOWLEDGEMENTS ... iii
ABSTRACT ... iv
ÖZ ... v

CHAPTER ONE - INTRODUCTION ... 1

1.1 Introduction to Real-Time Computer Graphics and Virtual Environments ... 2
1.2 A Reading Guide for the Following Chapters ... 8

CHAPTER TWO - LITERATURE SURVEY ... 10

2.1 Researches on Interactive Real-Time Computer Graphics and Virtual Reality in Medical and Engineering Simulations ... 10
2.2 Researches on the Use of Graphics Processing Unit (GPU) Programmable Pipeline in Computer Graphics and Virtual Environments ... 22
2.3 Researches on Graphics and Physics Software Libraries Developed by Academia and Industry ... 25
2.4 Researches on Augmented Reality Applications ... 27

CHAPTER THREE - DATA STRUCTURES AND SOFTWARE DESIGN PATTERNS ... 28

3.1 Data Structures ... 28
3.1.1 Maps ... 29
3.1.2 Graphs ... 30
3.1.3 Trees ... 34
3.1.4 Scene Graphs ... 36

3.2 Software Design Patterns 38
3.2.1 Visitor Design Pattern 40
3.2.2 Observer Design Pattern 42
3.2.3 Singleton Design Pattern 44
3.2.4 Factory Method Design Pattern 45
3.2.5 Iterator Design Pattern 46
3.2.6 The Façade Design Pattern 48
3.3 What is a Software Engine? 50

CHAPTER FOUR - GRAPHICS PROCESSING UNIT PROGRAMMING FOR GRAPHICS AND GENERAL PURPOSE COMPUTING 51

4.1 Short History of Computing Machines – From Antikythera Mechanism to Today's Massively Parallel GPUs 51
4.2 Shaders 59
4.3 Fixed Function Graphics Pipeline and Programmable Graphics Pipeline Architecture in Detail 60
4.4 Unified Shader Architecture 65
4.5 The Need for High Level Programming Languages for Computer Graphics– Cg HLSL and GLSL 67
4.6 NVIDIA Compute Unified Device Architecture - CUDA and General Purpose Computing 68

CHAPTER FIVE - ESSENTIALS OF REAL TIME GRAPHICS RENDERING 76

5.1 Transformations, Lines, Surfaces and Rendering Techniques in Computer Graphics 77
5.2 Gimbal Lock Problem – Rotation via Euler Angles and Quaternions 77
5.3 Lighting and Implementation of Light Shafts 79
5.4 Texturing and Implementation of Bump Mapping with Parallax Offset 80
5.5 Hand Rigging and Skinning 81

CHAPTER SIX - ESSENTIALS OF REAL TIME PHYSICS RENDERING AND SIMULATION OF DYNAMICAL SYSTEMS 84

6.1 Topological Definitions.. 85
6.1.1 Affine Spaces.. 85
6.1.2 Euclidean Spaces.. 88
6.1.3 Affine Transformations 90
6.2 Important Geometric Primitives for Computer Graphics and Definitions of Convex Combination and Convex Hull 93
6.2.1 Polytopes 94
6.2.2 Polygons 95
6.2.3 Quadrics.. 95
6.3 Minkowski Sum and Its Relation with an Intersection Test 95
6.4 Separating Axis Test 99
6.5 Primitive Bounding Volumes for Collision Detection Used in the Software 101
6.5.1 Axis Aligned Bounding Boxes 102
6.5.2 Sphere Bounding Volumes.. 104
6.5.3 Oriented Bounding Boxes.. 104
6.6 Collision Detection Pipeline Used in the Software 105
6.6.1 Collision Masking.. 106
6.6.2 Broad Phase 107
6.6.3 Narrow Phase.. 110
6.6.3.1 Gilbert - Johnson - Keerthi Algorithm (GJK) for Collision Detection between Convex Objects and Expanding Polytope Algorithm (EPA) for Penetration Depth Calculation 111
6.6.3.2 Solving the Constraints at Mechanical Joints–Linear Complementary Problem (LCP) 112
6.7 Mass-Spring Systems and Numerical Solutions for Governing Differential Equations.. 112
6.7.1 1-D 2-D and 3-D Mass-Spring Systems and Governing Differential Equations 113
6.7.2 Explicit Euler Integration 116

6.7.3 Second and Fourth Order Runge Kutta Integration 118
6.7.4 Verlet Integration 122
6.8 Mesh Topology Processing and Mesh Refinement – An Example to Mesh Cutting 124
6.9 Haptic Rendering with Rigid and Deformable Models 126

CHAPTER SEVEN - FEATURE SEGMENTATION TRACKING AND POSE ESTIMATION METHODS USED FOR AUGMENTED REALITY APPLICATION DEVELOPMENT DURING THE THESIS WORK 127

7.1 Feature Segmentation 127
7.2 Feature Tracking and Pose Estimation 128

CHAPTER EIGHT - ESTABLISHMENT AND CURRENT SETUP OF COMPUTER GRAPHICS AND VIRTUAL REALITY LABORATORY 131

8.1 VESTEL LCD Panel 133
8.2 Polhemus Fastrak Motion Tracking System 133
8.2.1 Reference Frame Alignment 135
8.2.2 Boresighting 136
8.2.3 Hemisphere Tracking 137
8.2.4 Output Data 137
8.2.5 Angular Operational Envelope 137
8.2.6 Position Operational Envelope 138
8.3 Sensable Phantom Omni Haptic Device 138
8.4 5DT Data Glove 5 Ultra USB Left and Right Pairs 139
8.5 5DT HMD 800 – 26 3-D Head Mounted Display 140
8.6 Logitech QuickCam Pro 9000 Webcams 141
8.7 ATI X1550 and NVIDIA GeForce GTX295 141

CHAPTER NINE - SOFTWARE DEVELOPMENT AND HARDWARE INTEGRATION RESULTS 143

9.1 Software Development Tools Used During the Thesis Work 143
9.2 Implementations Completed during Augmented Reality (AR) Application Research 144
9.3 Development Result of the Immersive Interactive Virtual Environment for Collaborative Anatomy Inspections in Medical Education................................ 147
9.4 Implementations Completed during Mathematical Elements of Computer Graphics and Real Time Graphics Rendering Research 164
9.5 Implementations Completed during Collision Detection Research 167
9.6 Experiences with SOFA – Simulation Open Framework Architecture 170
9.7 Development Stages of the Graphics User Interface using Qt Development Kit.. 174
9.8 Experiences with Cg and GPU Programming for Graphics.......................... 175
9.9 Experiences with NVIDIA CUDA and Performance Comparisons for Further Projects and Possible Implementations 180
9.10 Experiences with NVIDIA PhysX and Performance Comparisons for Further Projects and Possible Implementations 183
9.11 Construction of Mesh Spring Structures and Implementation of Topology Processing and Refinement for Mesh Cutting Operation Using Bullet Engine .. 184
9.12 Haptic Rendering Implementation Results... 186

CHAPTER TEN - CONCLUSIONS 188

REFERENCES.. 194

APPENDICES 221

CHAPTER ONE

INTRODUCTION

Real time computer graphics rendering and physics simulation cover broad range of fields ranging from mathematics to software design; from hardware design of human-computer interfaces to arts and system dynamics modeling.

The studies (Azuma, 1997), (Grady, 2003, p. 56), (Grady, 2003, p. 116), (Grady, 2003, p. 123) related with computer graphics and simulation engines such as (NVIDIA, 2008), (Coumans, 2010) and (The SOFA Team at INRIA Grenoble, 2009) aim the most realistic graphics in the 2D or 3D medias. At the same time, graphics hardware performance and architectures are vital for effective visualization systems when screen refresh rates and resolution are of concern. By the rapid developing graphics hardware technologies (Refer to chapter 2 and chapter 4), not only fast, high resolution and realistic images can be rendered via many display methods such as using the programmable graphics pipeline (Refer to chapter 4, chapter 5 and chapter 9) of the graphics processing units but also performance demanding scientific and general purpose computations can be accomplished by utilizing their massively parallel architectures (Refer to chapter 4 and chapter 9.).

The other variable which must be studied on is the realism and the 3-D perception of the images; because the human brain uses visual stimuli and other senses in order to perceive its real physical surrounding. The more realistic visual, auditory, tactile, olfactive stimuli, dynamically consistent and intelligent virtual environments the computer systems are able to generate, the more realistic and immersive perception of the virtual 3-D environment by the brain is accomplished. Therefore, various types of sensors can be used to create more realistic bio-feedback for more realistic perception.

The Virtual Reality (VR) and the Augmented Reality (AR) systems are based on the computer graphics, numerical modeling of systems and hardware components for

creating motion feedback in order to create immersive and realistic perception (Refer to Chapter 2.).

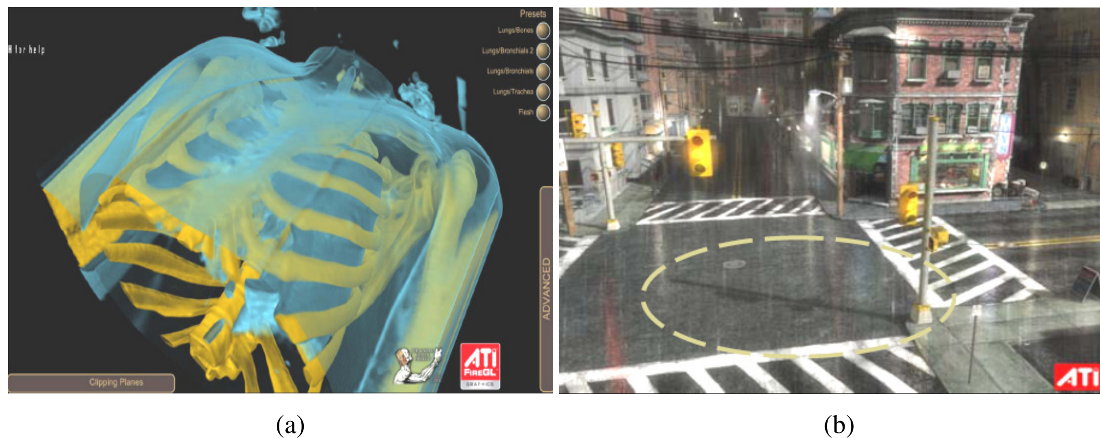
In this study, VR and AR systems are developed for the purpose of education in the areas of engineering and medicine. The main idea is to create computer graphics (Refer to chapter 5 and chapter 9) and dynamical system simulation (Refer to chapter 6 and chapter 9) based synthetic and semi-synthetic virtual (Refer to chapter 7 and chapter 9) environments, in which the user can interact with all synthetic and semi-synthetic objects by using hardware feedback components. For this purpose, hardware components (Refer to chapter 8) and also the necessary software modules (Refer to chapter 5, chapter 6 and chapter 7) must be combined together in order to create the sense of reality and immersion. The necessary graphics rendering and dynamical simulation or in other words physics rendering modules and hardware communication modules (Refer to chapter 3) were developed and applied in several applications in a specialized laboratory environment which has been equipped with VR hardware components and computer systems (Refer to chapter 8, chapter 9 and chapter 10). The figure 1.2 shows the fundamental block diagram of the thesis study.

1.1 Introduction to Real-Time Computer Graphics and Virtual Environments

Computer graphics has attracted a great attention from the researchers since mid 1970s. This attention was mostly motivated by the development in graphics hardware as will be mentioned in the following chapters. One of the first graphics hardware developed by IBM can be inspected at (Elliot, 2010). 1970s and early 1980s were mostly dominated by 2-D computer graphics some of which could be just rendered offline. Beginning from early 1980s, technological researches and investments pioneered by academia and industry resulted in significant technical and scientific leap in the field and opened new horizons for possible applications. The graphics hardware improvements that started at late 1980s allowed desktop computers to use graphics acceleration hardware that was once found just in workstations. These graphics accelerators enabled 3-D graphics applications run in interactive rates.

Therefore, algorithms concerning 2-D and 3-D graphics that were developed by academia and industry became executable in real-time at interactive rates not only on workstations, but also on desktop computers of normal users. This trend pushed the limits more and more in 1990s. This progress enabled graphics processing units to play an important role in computer graphics, scientific visualization, several optimization applications, entertainment and films. By the early 2000s, the researches of several institutions led to graphics processing hardware with massive power of parallel numeric computation. In this period, algorithms for generating more life like and interactive visualizations, games and the computation source demanding numerical and scientific computations began to be executed on graphics processing hardware, harnessing its computational power. Hence, general purpose central processing units are offloaded for other computational and control tasks. An introductory coverage of graphics processing unit architecture can be found in (Möller, Haines, & Hoffman, 2008).

One of the common motivating problems throughout all of the above period was the performance demand of real-time computer graphics based interactive applications, which should run at least at 15 frames per second (fps). On the other hand, computation power and parallelism need of the scientific, numerical applications and physics simulations were the other concerns that the researchers should have handled. Such applications include medical simulations, astrophysical simulations, molecular dynamics simulations, flight simulations for military, volumetric visualizations, visualization of differential equations, aerodynamics and fluid dynamics simulations and 3-D graphics rendering in entertainment field. Figure 1.1 presents relatively recent application examples from (Tatarchuk & Shopf, 2007) and (Tatarchuk, 2006).



(a) (b)

Figure 1.1 (a) An example of GPU based real-time medical visualization on ATI FireGL workstation graphics accelerator (Tatarchuk & Shopf, 2007). (b) An example of real-time rendering of a scene with lighting, shadows and rain (Tatarchuk, 2006).

The term *interactive virtual environments* or its more popular name *virtual reality (VR)* can be regarded as a special case of a simulation. As indicated in (Heim, 1998), different research groups use different terms for the same concept such that, the researchers at MIT, University of North Carolina preferred the term *virtual environments*, military scientist prefer *synthetic environments*, researchers at Human Interface Technology Lab at the University of Washington at Seattle refer to *virtual worlds* and Japanese researchers prefer *tele-existence*. Virtual reality can be considered as a 3-D interactive simulation of a real world environment or of a certain physical process. The user is immersed into the computer generated synthetic environment via head mounted display where he or she can interact with the virtual environment via haptic device or data gloves. At this point, efficient collision detection gains importance. (Bergen, 2004) and (Ericson, 2005) are important sources on the subject. Furthermore, the user can walk around in the virtual environment via motion tracker device. The users can even have meetings and collaborations with other users in the same synthetic environment but at the same time at the different real world place via network connections. All of these features mean that the user can manipulate, deform and change the virtual environment; and at the same time the virtual environment reacts according to the user actions in an intelligent way in order to make the user's senses perceive the virtual environment as real as possible. These interactions require numerical and stable solutions to linear or nonlinear differential equations governing the simulated system dynamics. Two

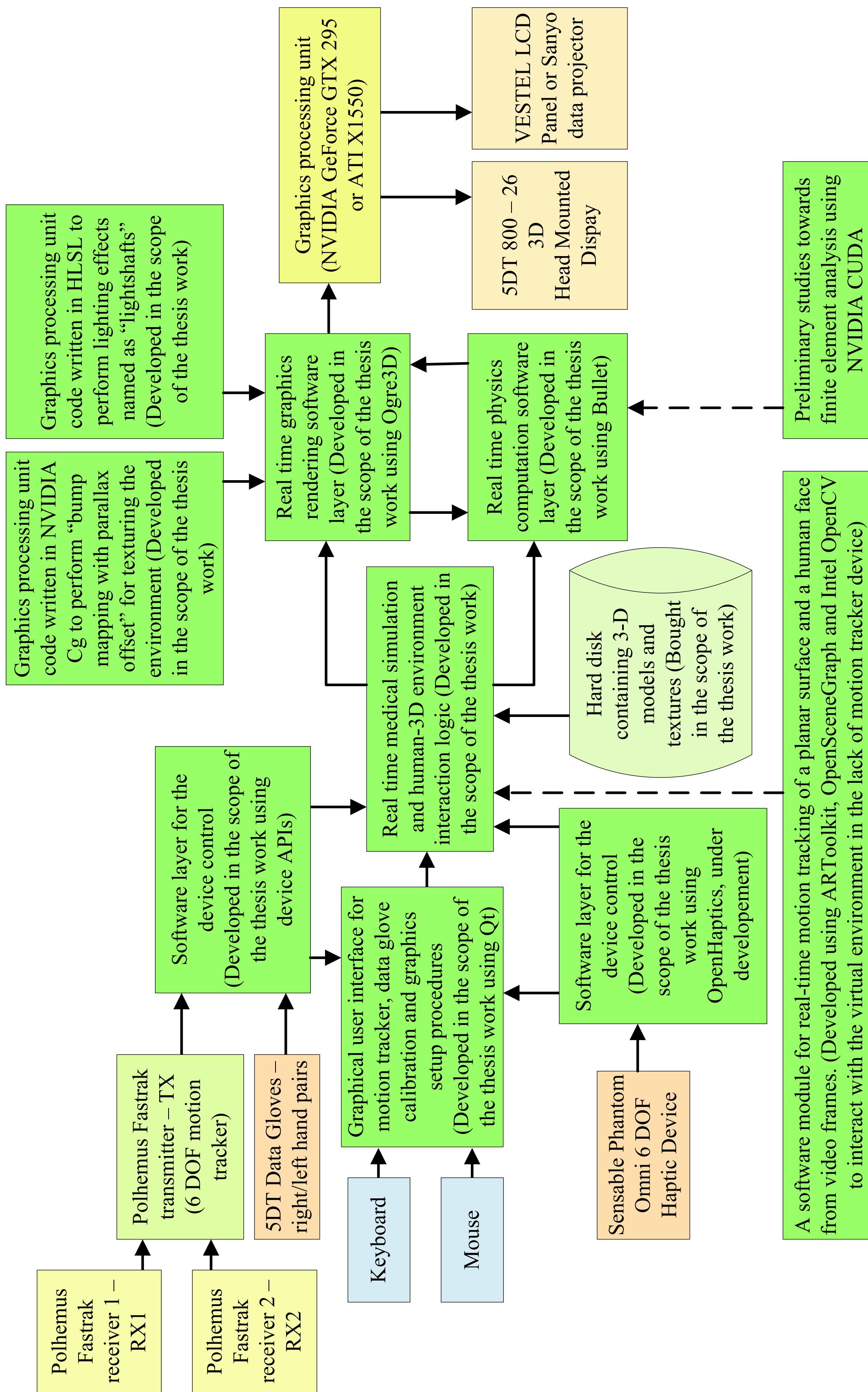


Figure 1.2 The overview of the 3-D virtual interactive environment developed during the thesis work.

important sources on the subject are (Khalil, 2002), (Hutton, 2004). Several applications of virtual reality can be found in (Grady, 2003). A historical development and technical terms of virtual reality can be found in (Heim, 1998). Considering the properties of a virtual reality system mentioned above, the overview of the completed software development during the thesis work targeting the generation of a collaborative dynamic virtual anatomy laboratory is presented in figure 1.2. Figure 9.2 presents the complete software layer diagram developed.

The term *augmented reality (AR)* can be regarded also as an interactive virtual environment, but with an exception. In augmented reality, the user and the virtual agents – intelligent or not – are in the real world. The real world surrounding the user is projected to the eyes via video or transparent head mounted displays. The virtual agents are registered with the features segmented from the video. These features may be natural environmental features as well as recognizable artificial features imposed by the humans. The accurate tracking of the recognized features in the real world and the accurate registration in real time are the key concerns of augmented reality applications. The historical development of augmented reality and technical terms can be found in (Azuma, 1997). A relatively recent work on a medical augmented reality application can be found in (Reitinger, Bornik, Beichel, & Schmalstieg, 2006).

Then what are the important components of an interactive immersive virtual environment that enable it to simulate the reality?

Vision is one of the most important senses of human. A virtual reality system attempting to immerse a person in a life like virtual 3-D environment should render the environment by appropriate real-time rendering techniques and by benefiting from the computational power of graphics processors. The static and dynamic systems in the virtual environment should be modeled mathematically such that their behaviors will be consistent. The preferred display scheme in these applications is the usage of the 3-D stereoscopic head mounted displays (HMDs). These displays

have two screens on which the synthetic virtual environment or the augmented real environment is rendered.

Human tactile sensory system should also be considered by the virtual reality system. The collisions of the user hand trying to touch a virtual object should be detected, computations should be done to calculate the contact points, the contact directions and the forces generated due to the collisions. Then necessary physical reactions should be simulated by the virtual environment. Data gloves or haptic devices are used for creating tactile senses in immersive systems.

Human auditory system should also be in concern to simulate the real world in a virtual environment so that the synthetic environment behaves acoustically consistent.

Human olfactory system has a vital role in many real world situations to perceive the environment. Therefore, a virtual environment in which collaborators live in should consider generating necessary stimuli in accordance with the environmental constraints and situations.

Finally, the user will expect to interact with intelligent virtual agents in the virtual environment as in the real world. So the virtual reality system should have intelligence and a capability to learn in order to evolve. This evolving intelligence can be used by the virtual environment to work in collaboration with the user such as an intelligent simulator evaluating or correcting the wrong actions of its user or to work against the user as an opponent such as a game.

All of the concepts mentioned above can be expressed and implemented in pure mathematics. Therefore, prior to attempting to design such a system, the researcher should understand how each of those components built up mathematically, algorithmically and then implemented programmatically. In order to be able to develop a simulation or immersive virtual interactive environment in which visual entities of the real environment and dynamics of the systems are simulated as

consistent as possible; the researcher should have a well established background in theoretical and practical aspects of computer graphics, central processing unit and graphics processing unit architectures and their programming, mathematics particularly in differential equations, topology and numerical analysis. Otherwise, the end product will just have empty but attractive names called virtual reality or augmented reality. In order to fill inside of these names theoretically and practically, the interested researcher should divide the whole work into its constituents that are mentioned above and study them carefully.

In the light of above concerns, as indicated briefly in the previous paragraphs, the scope of the thesis is to design and to develop a 3-D interactive virtual environment in which users are immersed to work collaboratively on medical anatomical operation scenarios. The virtual environment is aimed to be dynamic so that, the users can grasp the anatomical body parts, get medical information about that part and apply forces to soft tissues to deform them. The user can cut the soft body tissues to simulate a medical operation. These interaction options are presented to the users with a 3-D graphical user interface shown to the user upon a collision between a rigged and skinned user hand (Refer to chapter 5) and the corresponding anatomical model (Refer to chapter 9). Instead of rigging a hand mesh and using a data glove, other methods such as just capturing hand features then estimating the hand and fingers rotation and translation matrices by inverse kinematics from a camera can be implemented. In addition to the software development, the establishment of a new computer graphics and virtual reality laboratory in Dokuz Eylül University Electrical and Electronics Engineering Department is included in the scope of the thesis work (Refer to chapter 8). The simulation logic of the software is planned to be modular so that it can be suitable for the engineering simulations as well (Refer to chapter 9).

1.2 A Reading Guide for the Following Chapters

This section serves as a guide for the researcher for branching to the appropriate chapter of interest. Chapter two will provide a literature survey on virtual reality, graphics processor programming for virtual environments and on augmented reality

respectively. Chapter three will give an overview of data structures and software design patterns used throughout the thesis work. Chapter four will be about graphics processor unit programming for graphics and general purpose computing. Chapter five and chapter six will give a mathematical review about real time rendering and numerical methods for physics simulation used in the thesis work respectively. Chapter seven will be on feature segmentation, tracking and pose estimation methods used for augmented reality application developed during the thesis work. The reason of the development of a video based real time tracking system in the scope of an augmented reality application is the lack of the motion tracking and data glove equipment for the two years of the thesis project period. Chapter eight will give information on the computer graphics and virtual reality laboratory establishment process that has been completed in Dokuz Eylül University Electrics and Electronics Engineering Department (DEU EEE) in the scope of the thesis work. Chapter nine and chapter ten will be the software development results of the thesis and conclusion respectively.

A reader may find chapters two, three and four too exhaustive or overwhelming. But a wise and dedicated researcher will know that the time and the effort put into the mathematical theory, algorithmic details and into the previous applications of other research groups in the field are going to pay when the time comes for the software, algorithm design and implementation. Additionally, a researcher with a solid working background in the field can easily trace the problems and be on the confident side by comparing the results with the theory and previous researches during the algorithm implementation phase. But beyond all these, the time invested in mathematical theory and well accepted applications of the research groups; provide a researcher a different perspective to handle the problems and an enhanced imagination for new solutions.

CHAPTER TWO

LITERATURE SURVEY

Computer graphics with its roots originating from diverse fields of mathematics has been a wide research area in computer science since late 1970s. Evolving graphics hardware, together with the improving mathematics and graphics software libraries led to the field of real time computer. The related researches enabled the development of software that is able to generate 3-D life like virtual immersive interactive environments based on computer graphics. The user in this virtual environment can interact with its surrounding, collaborate with other users in the environment, simulate various dynamical systems in science and engineering in real time, intelligently interact with the computer, train and even entertain.

The aim of this chapter is to provide a literature survey on the recent applications conducted by the industry and the academia on virtual reality, the usage of graphics processing units as general purpose computation units and augmented reality. The applications are targeted to engineering and medical applications; but the researcher will find other diverse application areas. Hence, the researcher can immediately branch to the mathematical method or application reference of interest.

2.1 Researches on Interactive Real - Time Computer Graphics and Virtual Reality in Medical and Engineering Simulations

Researchers both in academia and have worked on applications of computer graphics and virtual reality targeting medical and engineering simulations. Virtual reality has found wide application area in medicine. One of the hot topics is 3-D realistic soft tissue deformations modeling in a virtual surgery. Mass-spring models, linear finite element method and nonlinear finite element method are generally used for modeling soft tissues. Mass-spring models are easy to simulate in real-time; however they are unable to simulate the process physically consistent. On the other

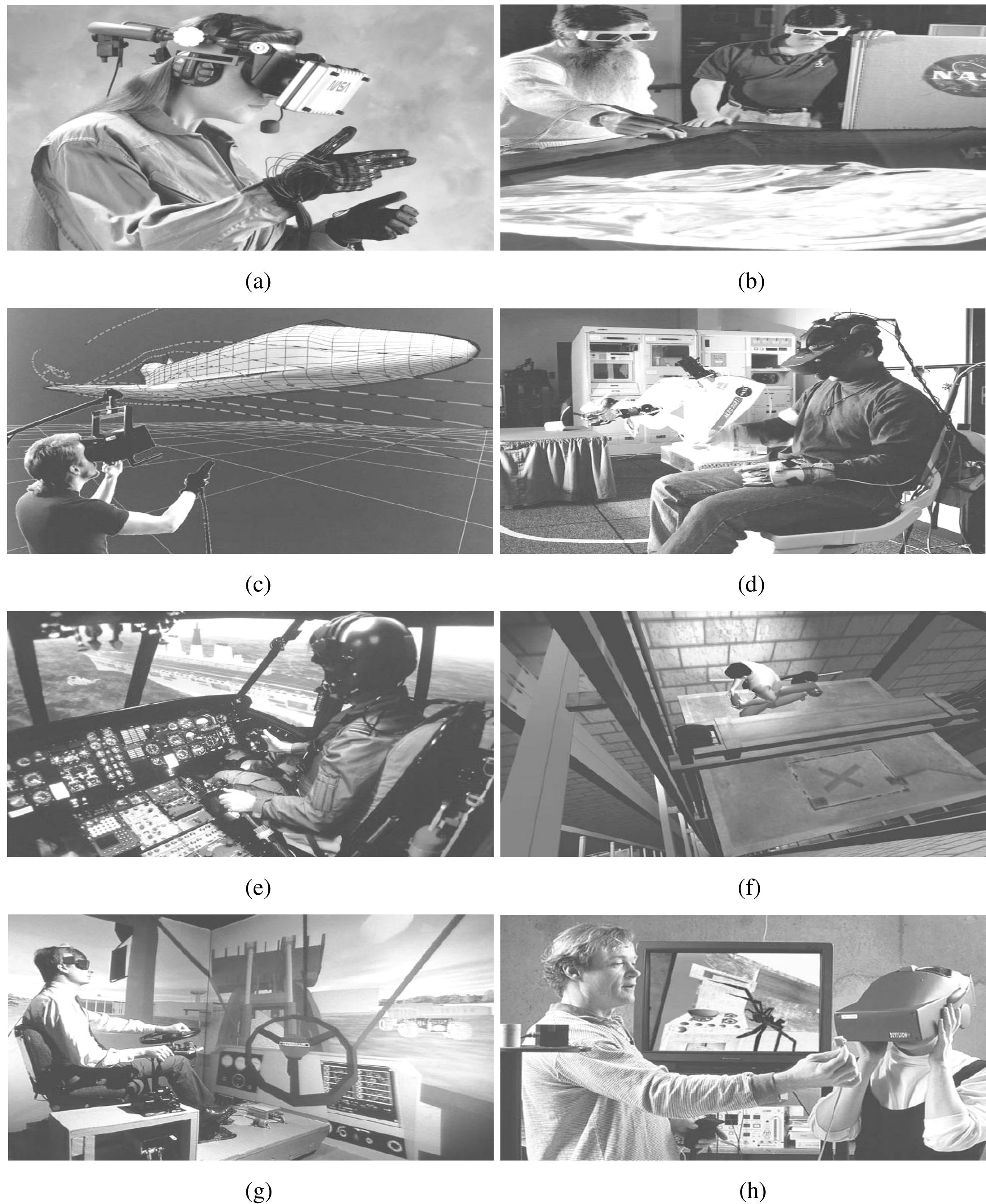
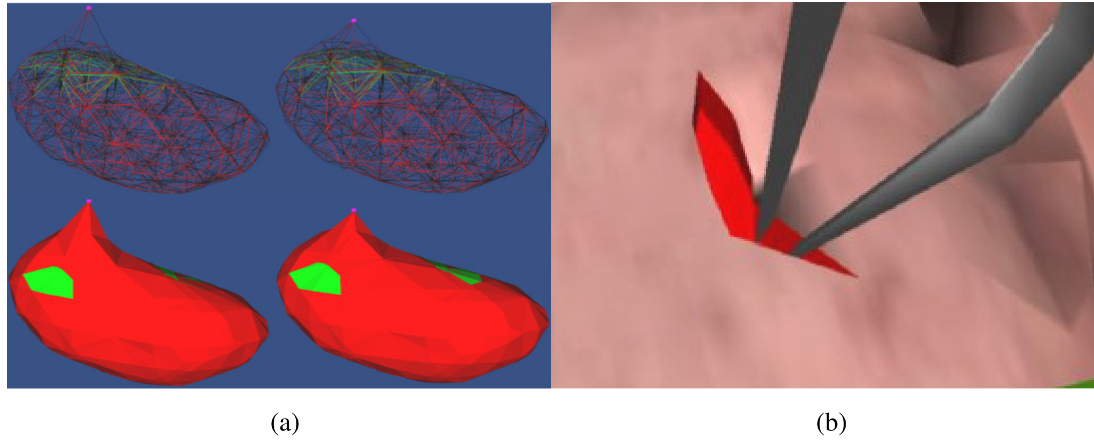


Figure 2.1 (a) The first fully functional VR display in history named as The Virtual Interactive Environment Workstation (VIEW) in NASA Ames Research Center (Grady, 2003, p. 56). (b) Shutter glasses in combination with flat-panel screens for three dimensional displays utilized in NASA Ames Research Center (Grady, 2003, p. 83). (c) NASA Virtual Windtunnel utilizing VR (Grady, 2003, p. 109). (d) NASA the Dextrous Anthropomorphic Testbed demonstrates VR-controlled robot to gather rock samples on distant planets (Grady, 2003, p. 116). (e) A helicopter flight simulator utilizing VR (Grady, 2003, p. 123). (f) Worker training with VR to fix elevators (Grady, 2003, p. 131). (g) The CAVE environment to design a wheel loader (Grady, 2003, p. 144). (h) VR therapy in medicine (Grady, 2003, p. 159).

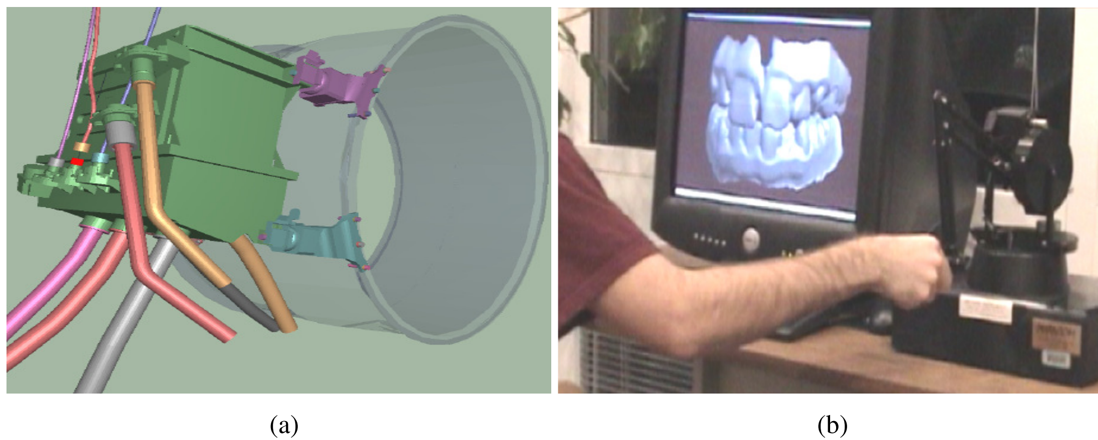
hand, finite element models can capture the physical characteristics of the dynamical system, they are very hard to simulate in real-time especially when the number of elements gets higher. But the advances in graphics hardware and the programmability of newer graphics processing units, enabled the researchers to perform computation power demanding tasks in real-time on graphics processing units. For the remaining details about development process of virtual reality, the researcher should refer to (Heim, 1998) and (Grady, 2003). Prior to moving onto the details of medical applications, the usage of virtual reality in diverse application fields conducted by National Aeronautics and Space Administration (NASA), industry field and medical therapists will be illustrated in figure 2.1.

For soft tissue modeling in a virtual environment, (Yan, Gu, Huang, Lv, Yu, & Kong, 2007) uses nonlinear finite element method for soft tissue modeling in real time. Additionally, for real time collision detection with soft tissue they use a spatial hashing collision detection method. They claim the superiority of their method over traditional mass-spring models and linear finite element models. The related work is shown in figure 2.2 (a). In (Wang, Becker, Jones, Glover, Benford, Greenhalg, & Vloeberghs, 2007), the authors propose the use of boundary element method for several topological operations such as prodding, pinching and cutting on soft tissues. In response to these operations, haptic and visual feedback is generated for the user in real time. In (Wang, & et al., 2007), the authors use boundary element method to model only the surface of the elastic objects. The related work is shown in figure 2.2 (b). In (Hamam, Nourian, El-Far, Malric, Shen, & Georganas, 2006), collaboration in distributed surgery simulation is emphasized. Another research on interaction in a distributed and shared virtual environment is (Glencross, Otaduy, & Chalmers, 2005). The research emphasizes on the challenges in visualization, collision detection, haptic rendering, dynamic system modeling and artificial intelligence while building such an interactive and intelligent environment. Figure 2.3 (a) and (b) shows a collaborative CAD prototyping application and haptic feedback application mentioned in (Glencross, & et al., 2005) respectively.



(a) (b)

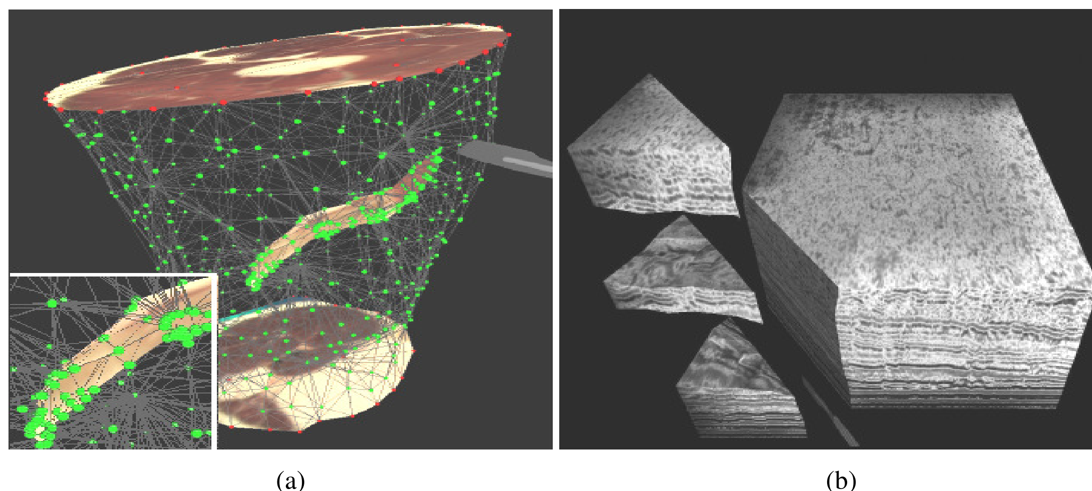
Figure 2.2 (a) Linear strain deformation of human kidney (Yan, & et al., 2007). (b) An example of soft tissue cutting with haptic feedback (Wang, & et al., 2007).



(a) (b)

Figure 2.3 (a) An example of collaborative CAD prototyping application. (b) Real-time haptic rendering application. Both applications can be found in (Glencross, & et al., 2005).

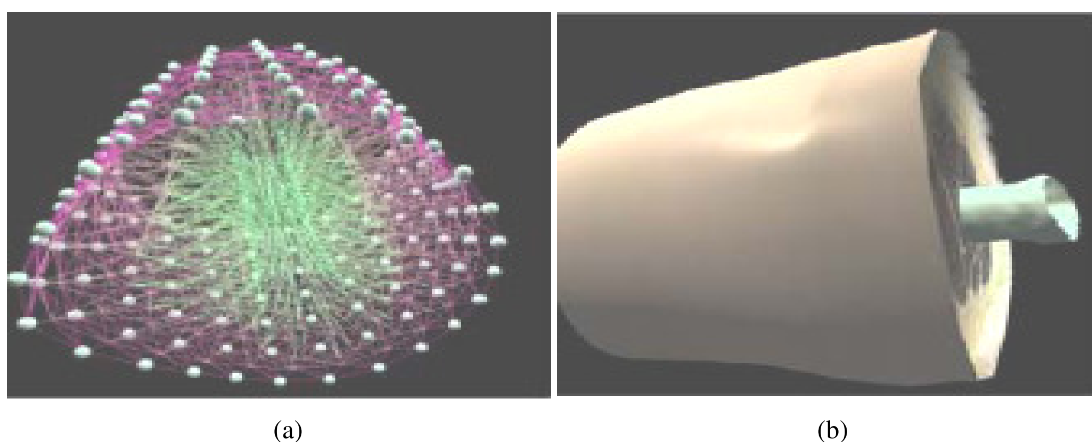
The other important work on simulating surgical cuts is (Bielser, & Gross, 2002). In that work, tetrahedral primitives are used for volumetric modeling in addition to adaptive subdivision scheme dynamically in order to keep the mesh topology consistent. For tissue deformation modeling they apply a relaxation scheme. For collision detection, they utilize a two stage hierarchical collision detection scheme. The first stage detects the boundary an element colliding with the surgical tool, the second stage finds the tetrahedral that is in contact with the surgical tool. Haptic feedback is also provided in real-time during the simulation. The related wok is shown in figure 2.4 (a).



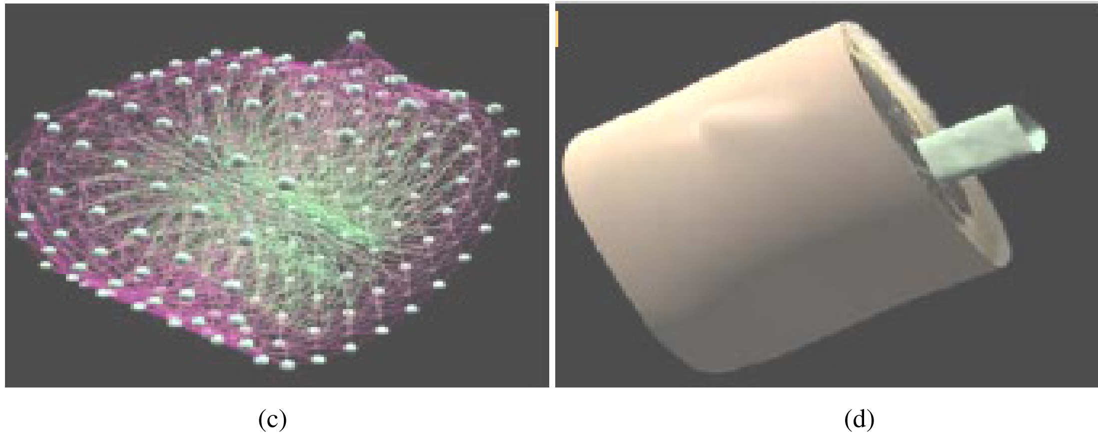
(a) (b)

Figure 2.4 (a) Collision detection and topology processing of tetrahedral meshes (Bielser, & Gross, 2002). (b) Processing intersections of tetrahedral meshes and a state machine approach to progressive subdivision (Bielser, & et al., 2003).

The authors of (Bielser, Glardon, Teschner, & Gross, 2003) propose an algorithm that consistently and accurately processes intersections of tetrahedral meshes in real time. Progressive subdivision and its state machine control are mentioned in that paper. The related work is shown in figure 2.4 (b). An application of real-time computer graphics and virtual reality in orthopedic surgery is covered in (Qin, Pang, Chui, Wong, & Heng, 2008). The authors propose a novel modeling framework for multilayered soft tissue deformation based on layered structure of real human organs. Considering performance issues, they employ a 3-D mass spring system for modeling biomechanical properties of the tissues. In order to increase the efficiency and interactivity, the authors use a physics processing unit. Their research is shown in figure 2.5.



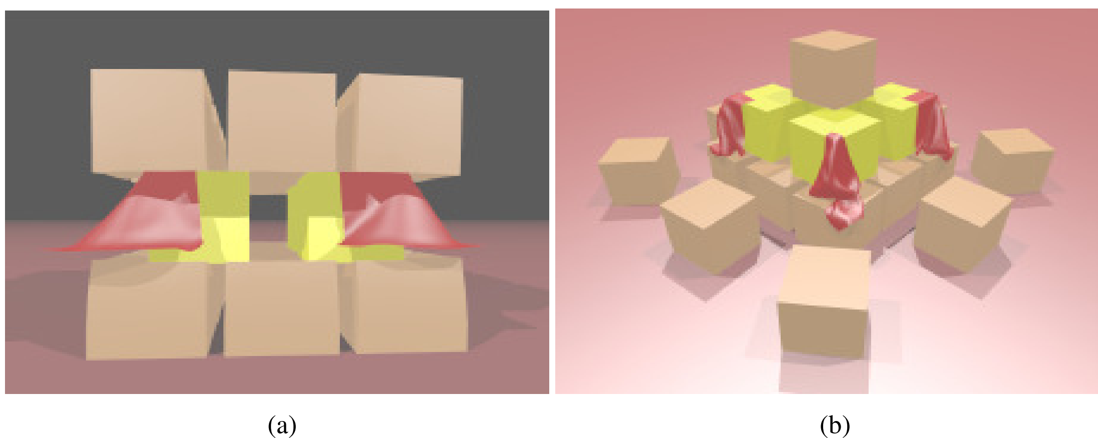
(a) (b)



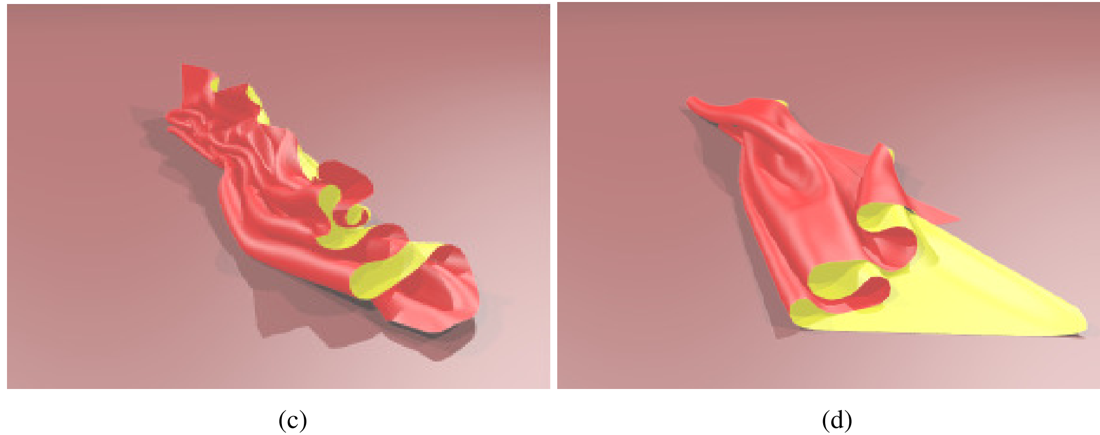
(c) (d)

Figure 2.5 (a) and (c) State of 3-D mesh spring models while pulling and pushing. (b) and (d) Texturized models corresponding to 3-D mesh spring models in (a) and (c) respectively (Qin, & et al., 2008).

Contact handling is a subfield in interactive computer graphics. A good theoretical and implementation coverage of constrained dynamics formulation with implicit complementary constraints, a time stepping algorithm based on progressive constraint manifold refinement (CMR) for progressive refinement of the constrained dynamics problem ensuring non-penetration, a solver based on iterative constraint anticipation for mixed linear complementary problems (MLCP) are given in (Otaduy, Tamstorf, Steinemann, & Gross, 2009). These topics are vital for many of the contact handling and collision detection problems. The proposed unified contact solver can cope with rigid bodies, co-rotational Finite Element Models (FEM), and mass spring systems. Figure 2.6 represents the results of the unified contact solver proposed by (Otaduy, & et. el, 2009).



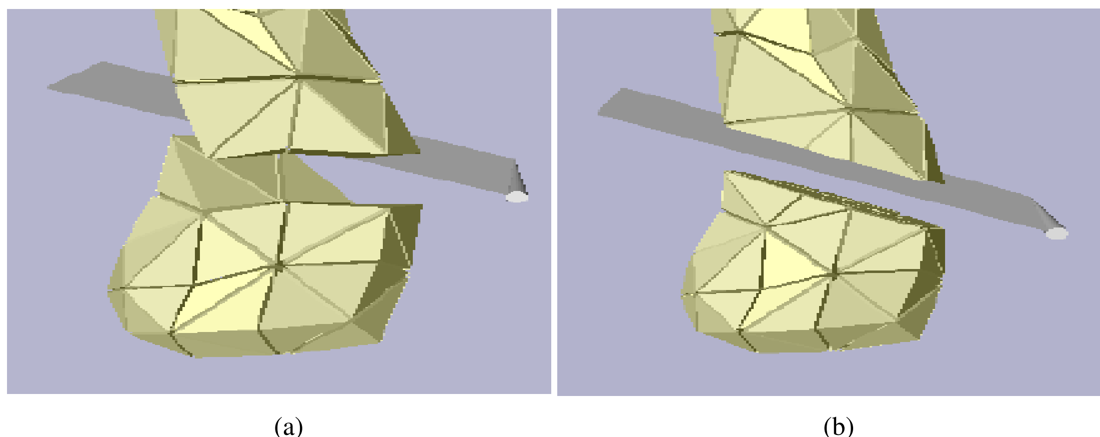
(a) (b)



(c) (d)

Figure 2.6 (a) and (b) demonstrate contact handling of rigid bodies (yellow), co-rotational FEM models (orange) and mass spring clothes (red) by the unified contact solver. (c) and (d) demonstrate the interpenetration in the mass spring model of a cloth ensuring that response to the interpenetrations does not add energy to the system (Otaduy, & et al., 2009).

Another application is the pathogical object removal in a hysteroscopy simulator as given in (Steinemann, Harders, Gross, & Szekely, 2006). The authors propose a hysteroscopy simulator in which cutting of soft deformable tissues is modeled by a tetrahedral mass spring system. A hybrid model is proposed that performs tetrahedral decomposition of the 3-D model, approximates the cut trajectory, new surface generation after the cut. Figure 2.7 represents some results from their work.



(a) (b)

Figure 2.7 (a) Tetrahedral mesh is cut along existing edges, nodes and faces. (b) After cutting with hybrid approach and snapping the nodes to the sweep surface (Steinemann, & et al., 2006).

A novel algorithm for efficient splitting of deformable solids along arbitrary piecewise linear crack surfaces in cutting and fracture simulations is proposed in

(Steinemann, Otaduy, & Gross, 2006). In this work, a meshless discretization of the deformation field and a novel visibility graph for fast update of shape functions in meshless discretization are proposed. Splitting operation is handled in two steps. Crack surfaces are synthesized as triangle meshes, these newly synthesized surfaces are used to update the visibility graph and thus the meshless discretization of the deformation field. Their results are given in figure 2.8.

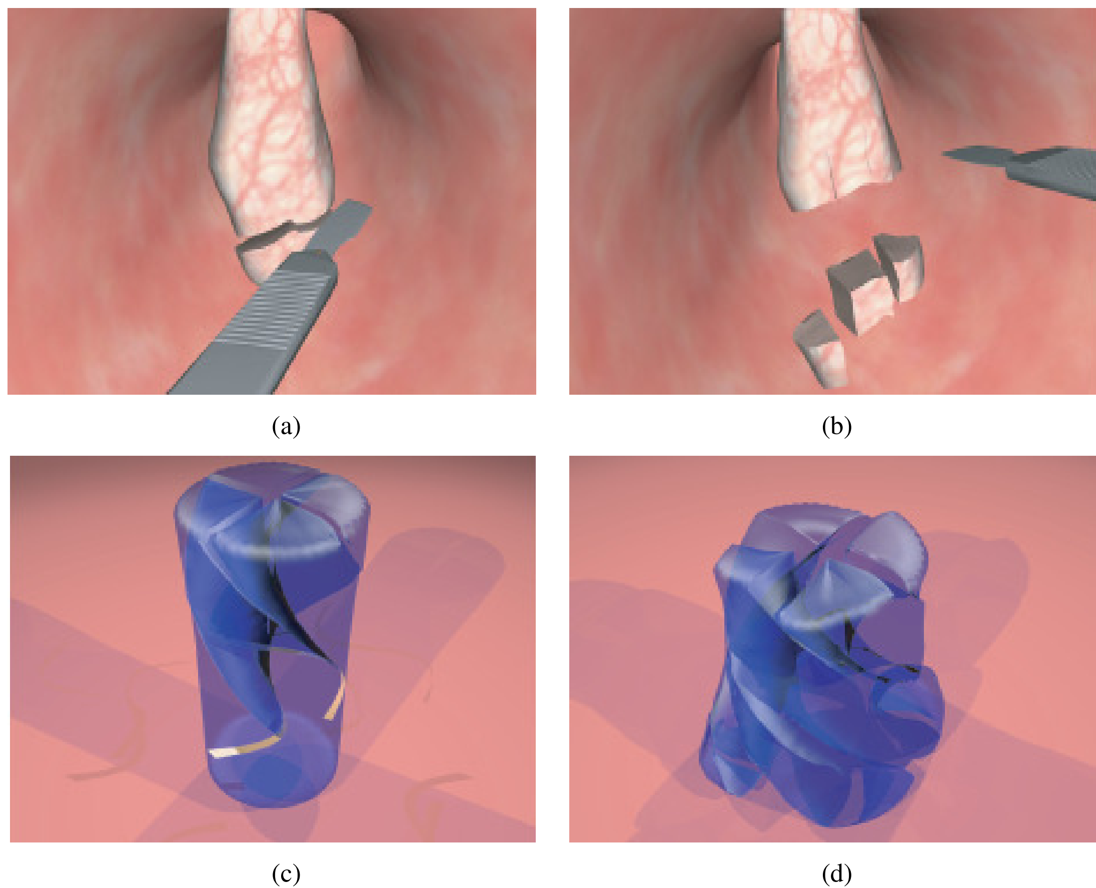
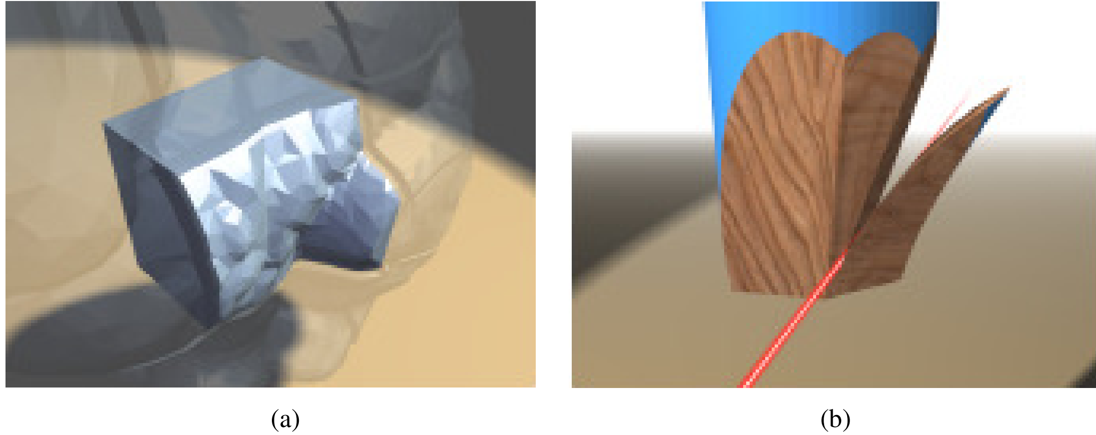


Figure 2.8 (a) and (b) represent surgical cuts. (c) and (d) represent spiral cuts (Steinemann, & et al., 2006).

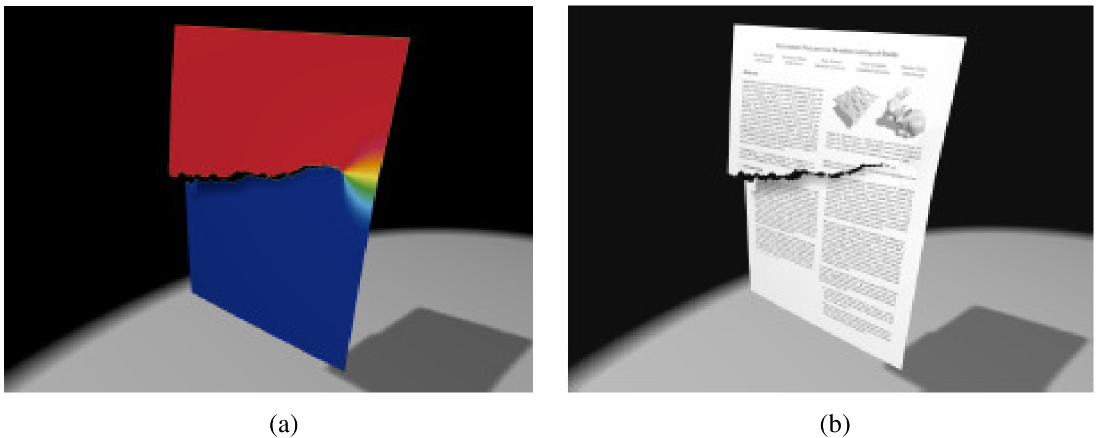
Convex or non-convex polyhedral elements can be simulated and deformed by using discontinuous Galerkin finite element method (DG FEM) with simple polynomial basis functions in (Kaufmann, Martin, Botsch, & Gross, 2008). They claim the superiority of DG FEM over standard FEM for incompressible materials. Additionally, the authors propose techniques for volumetric mesh generation, adaptive mesh refinement, and robust cutting. The results are in figure 2.9.



(a) (b)

Figure 2.9 (a) An example of non-convex element. (b) An example of topological change of convex element (Kaufmann, & et al., 2008).

The extended finite element method (XFEM) is adopted for simulating highly detailed cutting and fracturing of thin shells using low resolution meshes in (Kaufmann, Martin, Botsch, Grinspun, & Gross, 2009). Custom basis functions are used in the approximation process. It is claimed that cutting discontinuities by proposed method is possible in higher resolutions than the underlying mesh. The results are shown in figure 2.10.



(a) (b)

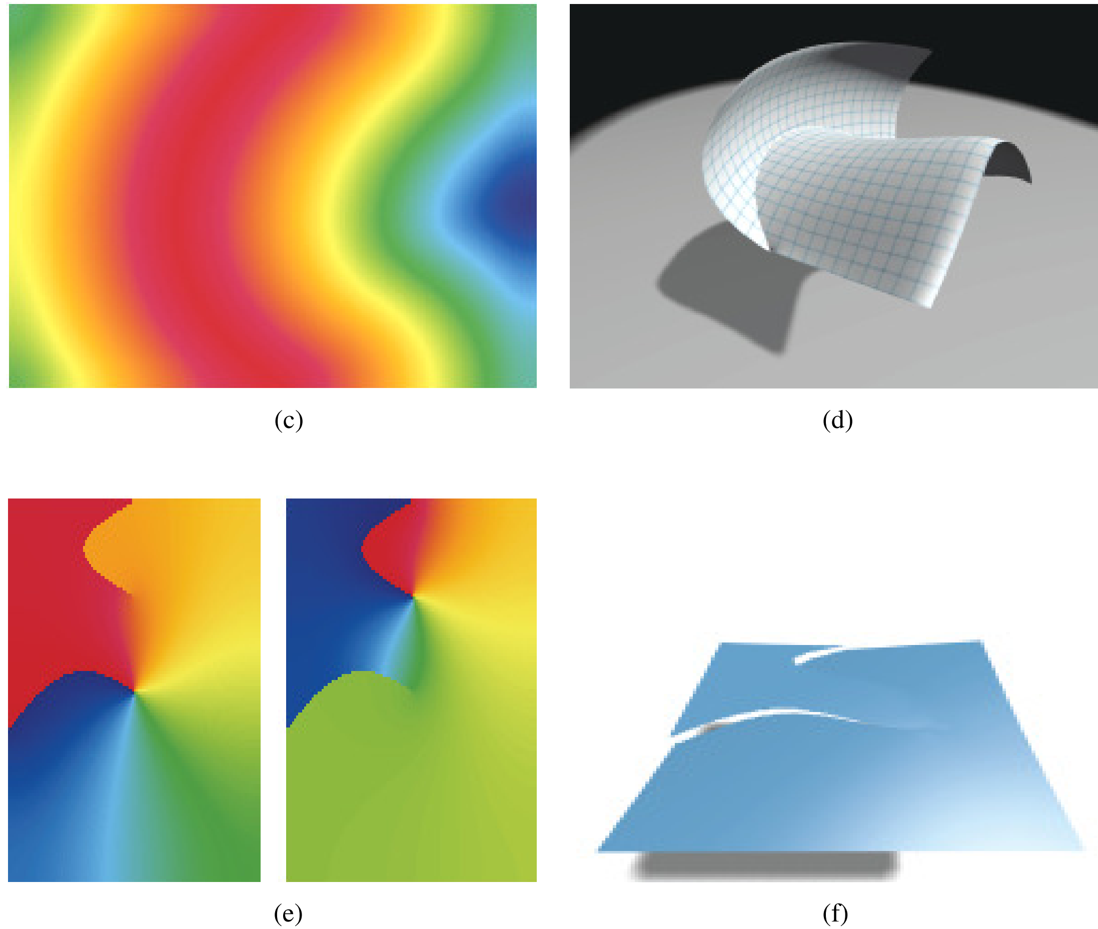
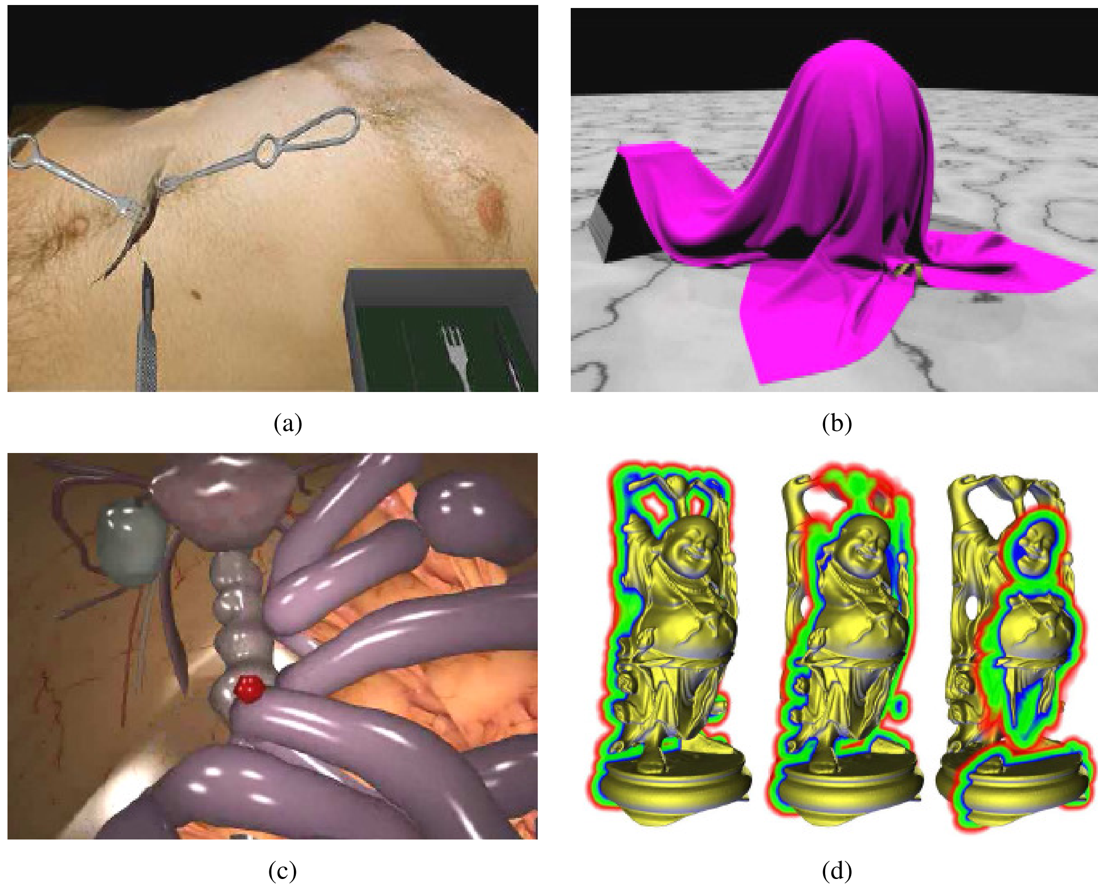


Figure 2.10 (a) Represents the harmonic enrichment function for a partial cut in a single element. (b) Represents the corresponding quad element behavior. (c) Represents a C^0 continuous enrichment element is used to add a crease to an element in as shown in (d). (e) Represents harmonic enrichment textures for multiple cuts within an element. (f) Represents the simulation of the element (Kaufmann, & et al., 2009).

Topological changes of dropping viscoelastic balls in an Eulerian fluid simulation are handled in (Wojtan, Thuerey, Gross, & Turk, 2009).

Collision detection is a vital concept for interactive virtual environments and medical simulators. Advances in deformable collision detection based on various approaches such as bounding volume hierarchies (BVHs), distance fields and spatial partitioning is discussed in (Teschner, Kimmerle, Heidelberger, Zachmann, Raghupathi, Fuhrmann, Cani, Faure, Thalmann, Strasser, & Volino, 2004). The related work is shown in figure 2.11.



(a) (b)

(c) (d)

Figure 2.11 (a) An example of deformable collision detection during virtual surgery. (b) An example use of bounding volume hierarchies for detection between rigid floor and deformable cloth. (c) Real time collision detection between intestine and mesentery. (d) Distance fields generated for collision detection between *Happy Budha* and other models (Teschner, & et al., 2004).

Another technique for collision detection for deformable volumetric bodies is the ray-traced collision detection. The detection and contact force generation using this technique is presented in (Hermann, Faure, & Raffin, 2008). Volumetric collision detection for deformable objects is covered in (Heidelberger, Teschner, & Gross, 2003) using layered depth image (LDI) decomposition of the intersection volume.

The researches on collision detection have been conducted for a long time. Especially, collision detection between rigid objects is a well-studied area. The motivation is towards the accurate collision detection of deformable topologies. But to understand the new concepts, the researcher should have a well established theoretical background on necessary data structures, mathematics and numerical methods. The collision detection methodologies given in the following references

form a basis of collision detection scheme that is used during the thesis work. Therefore (Tropp, Tal, & Shimshoni, 2005), (Möller, 1997), (Gottschalk, Lin, & Manocha, 1996), (Devillers & Guigue, 2002), (Hoff, Zaferakis, Lin, & Manocha, n.d.), (Möller, 2001), (Lin & Gottschalk, 1998), (Hubbard, P. M., 1995), (Barequet, Chazelle, Guibas, Mitchell, & Tal, 1996), (Held, Klosowski, & Mitchell, 1995), (Baraff, 1989), (Larsson & Möller, 2001), (Tan, Chong, & Low, 1999), (Held, 1998), (Eberly, 2008), (Jiménez, Thomas, & Torras, 2001), (Karabassi, Papaioannou, & Theoharis, 1999), (Barber, Dobkin, & Huhdanha, 1996), (Bielser, Maiwald, & Gross, 1999), (Teschner, Heidelberger, Müller, Pomeranets, & Gross, 2003), (Heidelberger, Teschner, Keiser, Müller, & Gross, 2004), (Baraff, 2001) and (Bergen, 1998) should definitely be studied.

The following reference papers will include researches on important numerical methods that are also used in the thesis work for collision detection, distance measurement, time of impact (TOI) calculation, penetration depth, solution of constraints and necessary mathematical topology. One of the fundamental algorithms for solving proximities between convex objects is the Gilbert-Johnson-Keerthi (GJK) Algorithm. Its mathematical theory and applications are studied in (Gilbert, Johnson, & Keerthi, 1988), (Bergen, 1999), (Vlack & Tachi, 2001), (Eberle, 2004) and (Kataria, n.d.). Expanding Polytope Algorithm (EPA) is important in calculating the penetration depth. Its theory and application are given in (Heidelberger, Teschner, Kaiser, Müller, & Gross, 2004), (University of North Carolina at Chapel Hill Department of Computer Science, 2004) and (Bergen, n.d.). For more detailed coverage of the concepts, the researcher should refer to (Bergen, 2004) and (Ericson, 2005).

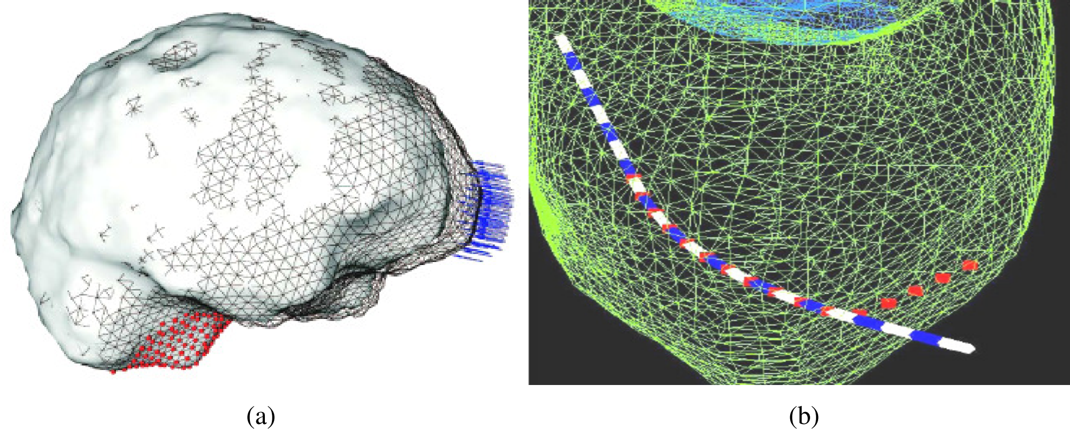
Differential equations, their numerical solution methods and stability are the heart of a physically consistent simulation or virtual environment. Moreover, as the computational power of the hardware increases, the use of finite element method increases resulting in more physically consistent simulations when compared with the mass spring systems. In addition to the references given above, the theoretical and implementation aspects are studied for simulating fundamental dynamic systems

such as cloths and volumetric elements in (Provot, 1996), (Desbrun, Schröder, & Barr, 1999), (Nielsen & Cotin, 1996), (Cotin, Delingette & Ayache, 1999), (Müller, Stam, & James, 2008), (Müller, James, Stam, & Thuerey, 2008), (James, 2008), (Nealen, Müller, Keiser, Boxermann, & Carlson, 2005), (Müller, Heidelberger, Hennix, & Ratcliff, 2006), (Müller, McMillan, Dorsey, & Jagnow, 2001), (Stam, 2009) and (Thuerey, 2008). For further details, the researcher should refer to (Press, Teukolsky, Vetterling, & Flannery, 2007), (Sewell, 2005), (Bathe, 1996), (Cook, Malkus, & Plesha, 1989), (Strang, 1986), (Hutton, 2004), (Ferreira, 2009), (Eberly, 2004), (Khalil, 2002), (Lander, 1999a), (Lander, 1999b) and (Lander, 1999c).

2.2 Researches on the Use of Graphics Processing Unit (GPU) Programmable Pipeline in Computer Graphics and Virtual Environments

Programmable graphics pipeline has dominated the fixed function graphics pipeline since early 2000s. Many application developers make use of this to perform graphics and numerical tasks on GPUs rather than central processing units (CPUs).

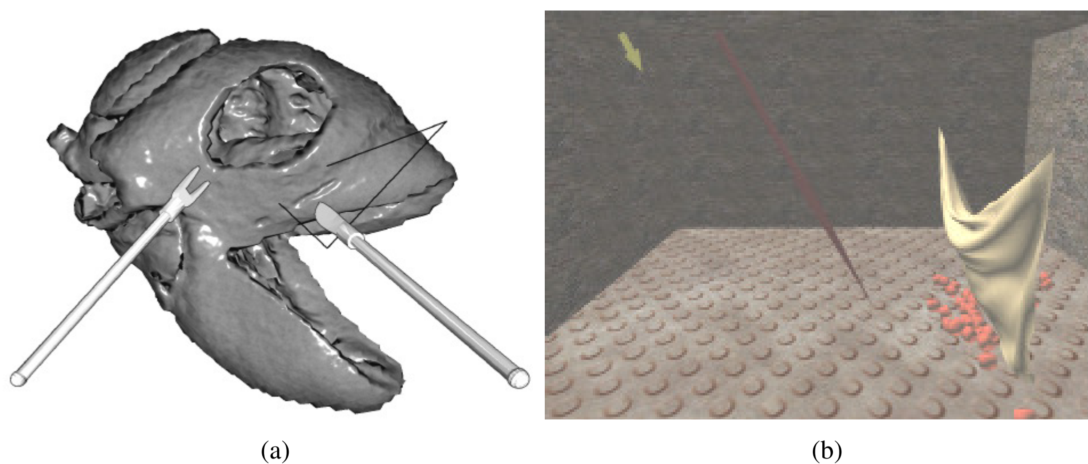
As seen from the researches from the previous section, most of the works depend heavily on numerical solutions especially on finite element method (FEM) and its derivatives. For computational power demanding virtual reality applications or simulations where high number of vertices, triangles and faces are present, GPUs are now alternative to CPUs for numerical computations. In (Taylor, Cheng, & Ourselin, 2008), the authors simulate a biomechanical model in real time. In that work, nonlinear Lagrangian FEM is used for modeling soft tissues. Their research is shown in figure 2.12 (a). Another research using GPU acceleration for cardiac intervention is (Yu, Chiang, Chen, Zheng, Cai, Ye, Zhang, S., Zhang, Y., & Mak, 2009) and their result are shown in figure 2.12 (b).



(a) (b)

Figure 2.12 (a) Overlaid images of the undeformed (wire-frame) and deformed (surface) brain model with 46655 elements. Anchor nodes at the brain stem are shown as red points, displacement direction of the displaced nodes are shown as blue arrows (Taylor, & et al., 2008). (b) White blue catheter and heart wall interaction (Yu, & et al., 2009).

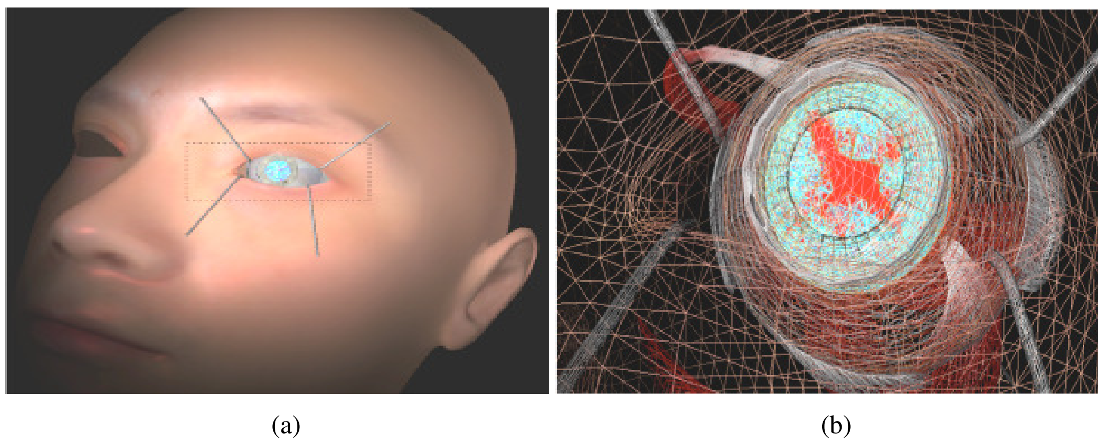
Another surgical simulation utilizing GPU acceleration with spring mass system is (Mosegaard, Herborg, & Sørensen, 2005). Their research is shown in figure 2.13 (a). NVIDIA CUDA based system is used for surgery simulation in (Liu & De, 2008), in (Farias, Almeida, Teixeira, Teichrieb, & Kelner, 2008) for deformable body physics simulation, in (Rasmusson, Mosegaard, & Sørensen, 2008) for volumetric mass spring damper models. Another application of mass spring systems are 2-D topologies such as clothes. This topic is examined in (Georgii & Westermann, 2005) based on GPU. Their work is given in figure 2.13 (b).



(a) (b)

Figure 2.13 (a) Heart surgical simulation (Mosegaard, & et al., 2005). (b) Mass spring system for simulating 2-D topology i.e. cloth (Georgii & Westermann, 2005).

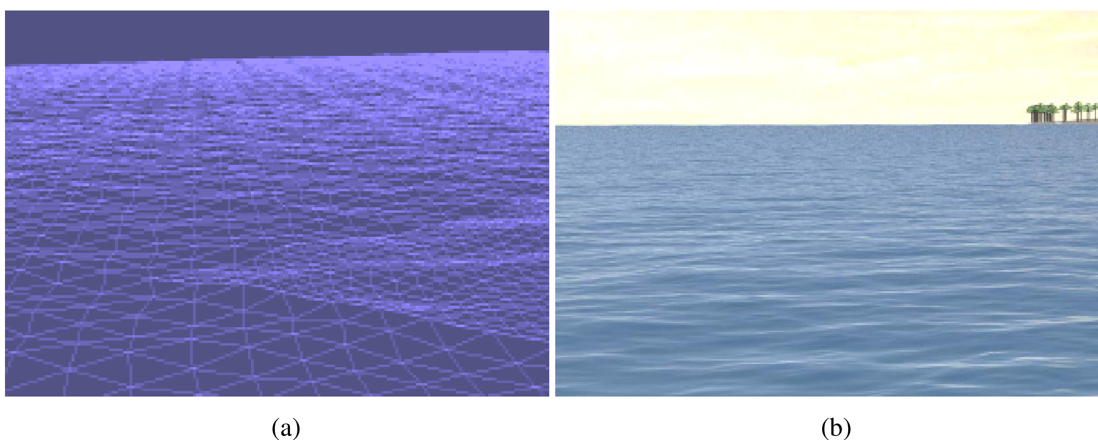
Similar researches that should be inspected are (Ranzuglia, Cignoni, Ganovelli, & Scopigno, 2006) and (Liu, Jiao, Wu, & De, 2008). Additionally, (Göddeke, Buijssen, Wobker, & Turek, 2009) presents an overview of GPU cluster computing for finite element applications. An important research from INRIA is presented in (Comas, Taylor, Allard, Ourselin, Cotin, & Passenger, 2008). Results from that research are given in figure 2.16.



(a) (b)

Figure 2.16 (a) A solid rendering. (b) Wireframe rendering of a real-time eye surgery using FEM in SOFA (Comas, & et al., 2008).

An implementation of ocean surface generation, adaptive tessellation and optical effects generation on the GPU is presented in (Li, B., Wang, Li, Z., & Chen, 2009) with the shown results in figure 2.17.



(a) (b)

Figure 2.17 (a) A wireframe rendering. (b) A solid rendering of a real time ocean simulation with optical effects on GPU (Li, & et al., 2009).

Programming GPUs towards important numerical computations are studied in (Lahabar & Narayanan, 2009), (Krüger & Westermann, 2003), (Spampinato, Elster, 2009), (Fujimoto, 2008), (Jang, Park, & Jung, 2008), (Velamparambil, Cormier, Perry, Lemos, Okoniewski, & Leon, 2008), (Amorim, Haase, Liebmann, & Santos, 2009), (Bolz, Farmer, Grinspun, & Schröder, 2003) and (Huang, Ponce, Park, Cao, & Quek, 2008).

2.3 Researches on Graphics and Physics Software Libraries Developed by Academia and Industry

OpenSceneGraph is a real time graphics rendering engine needed to manage scenes with huge number of nodes. It has been used in the thesis work for initial application development phase and for augmented reality application development. But the choice for a real time rendering engine for final application is Object Oriented Graphics Rendering Engine (Ogre3D). This decision was given due to its ease of integration with the preferred physics rendering engine Bullet, well designed documentation, shader handling and ease of scene management. The interested researcher should refer to (OpenSceneGraph, 2010) and (Martz, 2007) for OpenSceneGraph; to (Jacob, 2010) and (Junker, 2006) for Ogre3D.

Sofa is a well designed open source physics simulation framework developed at Institut National De Recherche En Informatique Et En Automatique - INRIA Grenoble. Although many of physical processes can be simulated, it is mostly specialized for medical applications. Because of its specialized structure, the developer should have a well understanding of numerical concepts especially nonlinear finite element modeling, advanced collision detection techniques and etc…. At first sight, the software modules seem tightly connected to each other, therefore the developer should carefully inspect and do necessary modifications on the source code for using modules independently with custom software modules and graphics engines. Sofa supports GPU general purpose processing with NVIDIA CUDA (Compute Unified Device Architecture). The developer need not to write a C++ code, a XML script can also be used for application development. But

integration with custom software modules should be concerned, if a XML script is used. The researcher interested in Sofa should refer to (The SOFA Team at INRIA Grenoble, 2009) and (The Sofa Team, 2008). Refer to section 9.6 for implementation results accomplished using SOFA through the thesis period.

Bullet3D is an industry standard physics engine used by SONY Playstation, Microsoft Xbox360, Nintendo Wii, AMD, movies such as Toy Story 3 and many other scientific simulation purposes. As a physics engine, Bullet was the preferred one throughout the thesis work. The main reasons for this choice were the ease of integration with the preferred graphics engine Ogre3D, the well designed open source engine software, availability of tutorials and papers and most importantly Bullet is well suited for the researchers who desire to understand the fundamental concepts of contact detection and collision detection methodologies for several topological constructs, mass-spring models for deformable objects, numerical calculations for fluids and particles, physical constraints and the numerical ways used in handling them, 2-D and 3-D elements that are triangle and tetrahedral respectively and their construction, proximity detection, penetration depth, necessary software interrupt generation and all the related numerical analysis concepts. One more important point with Bullet is that the theoretical mathematical concepts given in many fundamental books such as (Ericson, 2005), (Bergen, 2004) can easily be followed in the source code of Bullet. But prior to integration with the custom or open source graphics engine, the source code of Bullet should be inspected carefully. The researchers interested in Bullet should refer to (Coumans, 2010), (Coumans, 2009) and (McShaffry & et al., 2009).

NVIDIA PhysX is a C++ physics engine developed by NVIDIA for its GPUs. Rigid and soft objects, collision models can be handled with this engine. The engine also supports physics rendering based on GPU. The interested researcher should refer to (NVIDIA, 2009b), (NVIDIA, 2008) and section 9.10 for implementation results.

Computational Geometry Algorithms Library (CGAL) is a C++ computational geometry library developed by well known collaborative institutions. The researchers

interested in computational geometry should definitely search on (CGAL, 2010) and (CGAL, 2009). Besides, this field is a vital area of mathematics.

2.4 Researches on Augmented Reality Applications

The researchers interested in augmented reality may use (Azuma, 1997), (Brown, Julier, Baillot, & Livingston, 2003), (Barakonyi, Psik, & Schmalstieg, 2004), (Vallino, & Brown, 1999), (Harada, Nazir, Shiote, & Ito, 2006), (Reitinger, Zach, & Schmalstieg, 2007), (Reitmayr & Schmalstieg, 2007), (Reitmayr & Schmalstieg, 2004), (Pathomaree & Charoenseang, 2005), (Piekarski & Thomas, 2003), (White, Feiner, & Kopylec, 2006), (Goose, Sudarsky, Zhang, & Navab, 2002), (Mizuno, Kato, & Nishida, 2004), (Reitmayr & Schmalstieg, 2001), (Marathe, Carey, & Taylor, 2007), (Fürnstahl, Reitinger, & Schmalstieg, 2006), (Reitinger, Bornik, Beichel, & Schmalstieg, 2006) as a starting point for current applications. In figure 2.18, some of those researches are illustrated.

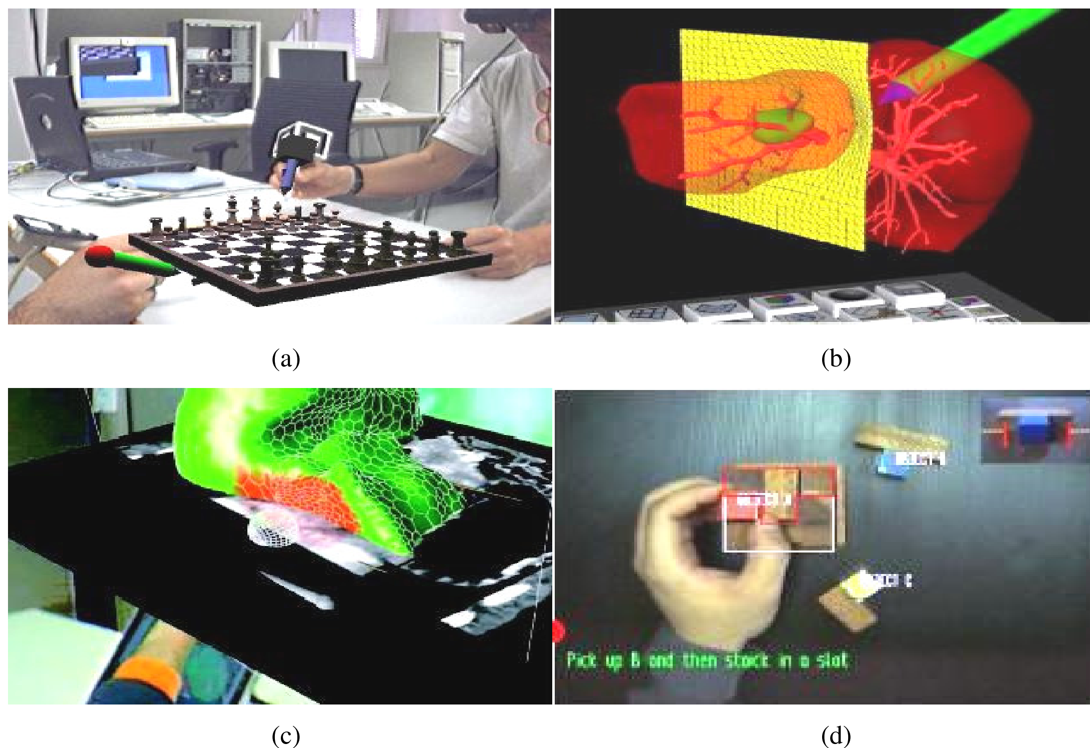


Figure 2.18 (a) Playing chess in a collaborative AR environment (Reitmayr, & Schmalstieg, 2001). (b) Global mesh partitioning (Fürnstahl, & et al., 2006). (c) Liver surgery planning with AR (Reitinger, & et al., 2006). (d) Task assisting with AR (Pathomaree & Charoenseang, 2005).

CHAPTER THREE

DATA STRUCTURES AND SOFTWARE DESIGN PATTERNS

The developments in the architecture of computation machines and the increasing diverse application areas of these machines from scientific simulation, embedded applications, and interactive 3-D applications to entertainment have become one of the reasons of the evolution in software design and programming paradigms. Therefore, programming and software design have undergone several periods starting from mechanical scheme, hardwired scheme of 1940s, machine language, assembly language to more flexible, performance oriented and portable functional programming schemes. Finally, object oriented programming paradigms starting from late 1960s have resulted in more reusable, modular, portable, manageable and maintainable software.

Today's modern software runs on both sequential and parallel computing architectures. Therefore it is essential to understand certain data structures and software design patterns in order to design, to manage and to maintain software as a solution to a specific problem at hand. Hence, this chapter is going to explain important data structures, their mathematical origins and software design patterns that are benefited from, during the software development process in the scope of this thesis. The chapter will end with the definition of a "software engine" and its relation with data structures and software design patterns.

3.1 Data Structures

Data structures are fundamental concepts for computer science. When a research is done on the data structures, it will be seen that all have a well-established mathematical and theoretical roots. In this part, brief mathematical aspects in addition to an introduction will be given on the data structures that are fundamental to understand for the scope of the thesis goal. For the excellent theoretical and applied coverage of data structures and algorithms, the researcher should refer to

(Cormen, Leiserson, Rivest, & Stein, 2003). Additionally, excellent information about applied data structures specifically for C++ can be found in (Smith, 2004).

3.1.1 Maps

Maps are data structures that have two fields as a primary key and a value. A map provides a mapping between the primary key and the memory slot where the corresponding value is stored. The memory slots constitute the hash table. Hence, the key-value relationship in a map can be considered as an associative memory as given in (Smith, 2004); that is, a particular value can be searched in, removed from, inserted to a map or can be modified by using a particular key. Although maps provide fast, random access and dynamic size change in runtime, their implementations should guarantee that all the values should have a unique key. Therefore, there should be a one-to-one (injective) and onto (surjective) function or in another words a bijective function that maps the key values in its domain to the corresponding memory slots in its range where the corresponding values are stored. This is depicted in figure 2.1.

Let K be a space of used keys in the map. Considering K as the domain of bijective function *f(k)*, then,

$$f : K \rightarrow M \quad , M \subset N \tag{3.1}$$

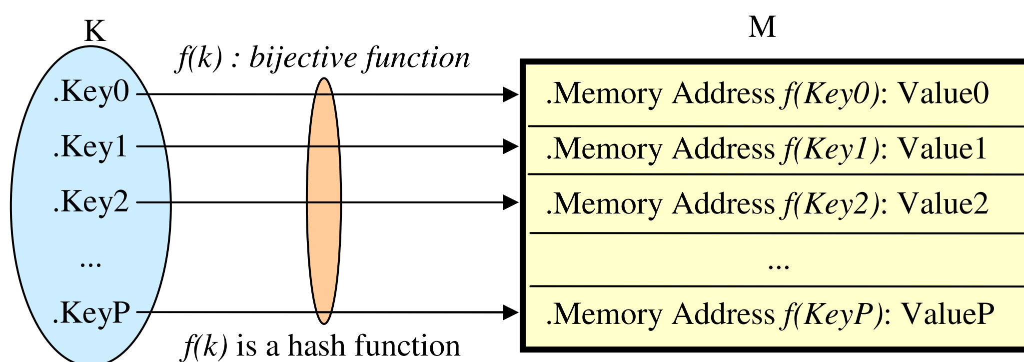


Figure 3.1 Mapping between keys and memory slot addresses of the hash table.

Essentially, while designing the hash function *f(k)*, several constraints should be considered. First of all, *f(k)* should be deterministic because the storage and the retrieval of the value will be performed using the same corresponding key value that belongs to the domain of *f(k)*. Secondly, to minimize the latencies, *f(k)* should map the key value to the memory address *f(key)* as soon as possible during the value storage and retrieval. Thirdly, *f(k)* should uniformly map the key values from its domain to its range that is the memory space of the computing machine reserved for the hash table. In other words, biasing towards the same memory address should be avoided. Finally, keeping the third constraint in mind, while designing *f(k)*, memory collision that is the mapping of the key value to an occupied memory address should be handled.

Although there are several methods for resolving collisions and designing hash functions such as collision resolving by chaining, hash function generation by multiplication or division, universal hashing and etc…, these are out of the scope of the thesis. The interested researcher should refer to (Smith, 2004, chap. 5), (Cormen, & et al., 2003, chap. 11), (Knuth, 1973) and (Marsaglia, 1996).

As an application example from the thesis work, maps are used to store pointers to render models and also pointers to collision models with corresponding keys.

3.1.2 Graphs

A graph is formed by a nonempty set of vertices *V* and a nonempty set of edges *E*. Typically, set *E* can be either empty or nonempty according to the topology of the graph. In literature, a graph *G* is typically denoted as follows;

$$
\begin{aligned}
G = (V, E) \ , \ \text{where}\ & V = \{v_0, v_1, \ldots, v_M\}, \\
& E = \{(v_i, v_j) : \forall i, j \in N, v_i \in V \wedge v_j \in V\} \\
& \ni \ E \text{ forms a binary relation on } V.
\end{aligned}
\tag{3.2}
$$

Simply speaking, a binary relation on V is a subset of all ordered pairs (v_i, v_j); in other words, a subset of the Cartesian product V x V as depicted in equation (3.3).

$$E \subseteq VxV = \{(v_i, v_j) : \forall v_i \in V \wedge \forall v_j \in V\} \tag{3.3}$$

Edges have weights w_{ij} such that,

$$\exists (f : E \rightarrow \Re) \ni w_{ij} = f(E_{ij}) \tag{3.4}$$

Graphs can be represented in three ways in computer memory: Sets, adjacency list and adjacency matrix. These representations will be exemplified using the figure 3.3.

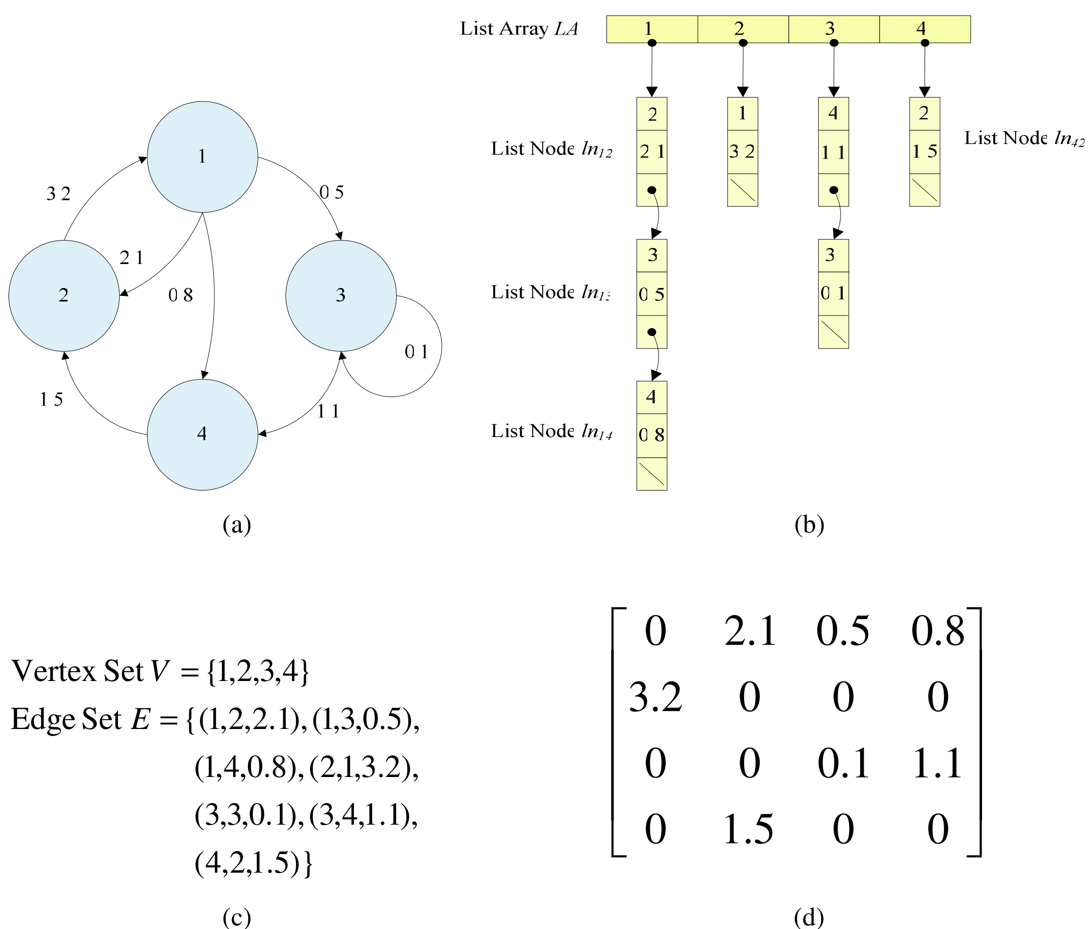


Figure 3.3 (a) An example Graph G. (b) The adjacency list representation of G. (c) The set representation of G. (d) The adjacency matrix of G.

Referring to the figure 3.3, the set representation uses two sets. One of the sets is for the vertices and the other is for edges. Each triple element in the edge set represent the start vertex, end vertex and connection weight respectively.

The adjacency list representation is composed of an array of lists for $\forall v_i \in V$. Hence, every member of the list in the array depicts the edge formed by $v_i \in V$ and $v_j \in V$. That is,

$$\forall v_i \in V, v_j \in list(v_i), \exists\ edge(v_i, v_j) \in E \ \ni list(v_i) \text{ is a list of vertices connected to } v_i \tag{3.5}$$

The adjacency matrix representation uses *NxN* adjacency matrix *M* where *N* is the number of vertices in the graph. *Row i of M* represents the start vertex, *column j of M* represents end vertex and M_{ij} represent the connection weights.

According to their connection topologies, graphs can be divided in two main groups as directed graphs and undirected graphs. As opposed to directed graphs, the edges of the undirected graphs are composed of unordered pairs such as,

$$edge(v_i, v_j) = edge(v_j, v_i) \tag{3.6}$$

If a vertex of a directed graph has a cycle edge, then it is called directed cyclic graph else it is called directed acyclic graph. Undirected graphs do not have cycles. These are shown in figure 3.4.

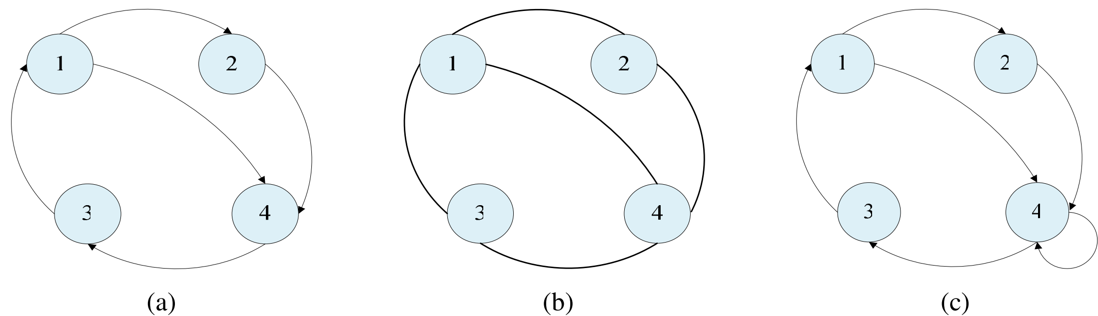


Figure 3.4 (a) Directed graph. (b) Undirected graph. (c) Directed cyclic graph respectively.

There are graph algorithms some of which are also used in the thesis work that worth mentioning. First of them is finding a path, that is querying if there exists a path between vertices v_i and v_j in other words querying whether v_j is reachable from v_i or not; or reachability can be checked in the reverse direction if the graph topology permits. The simplest methods for this task are breadth-first search and depth-first search. Breadth-first search looks at all vertices length one away relative to the start vertex v_i where the search is initiated from. If the target vertex v_j is found the algorithm terminates, else all the vertices at length two away relative to v_i are searched. The graph is traversed in this scheme till the target vertex v_j is found. If v_j is not found null is returned. At the end of the traversal, a tree containing all the reachable vertices from v_i is built. This tree contains the shortest paths to all the reachable vertices from v_i. On the other hand, depth-first search chooses one of the edges from the most recently discovered vertex v_i. Then, the search progresses along that path until the target vertex v_j reachable from v_i is found or an edge that has been traversed is reached. Then the search backtracks the most recently traversed edge e_k to the vertex v_k where e_k originates from. Then the search selects an edge originating from v_k that has not been traversed and traverses that path. This search scheme continues until the reachable target vertex v_j is discovered or all the reachable edges are traversed.

In graph theory and also in computer science, the shortest path between the vertex v_i to a reachable vertex v_j is of particular interest. Mathematically, it is the path that has the least connection weight sum when traversed from v_i to v_j. That is,

$$\text{Shortest path } p = \min_{v_i \text{ connected to } v_j} \left(\sum_{k=i}^{j-1} w_k \right) \ni \ w_k \text{ is the connection weight between vertices } v_k \text{ and its descendant } v_{k+1} \tag{3.7}$$

The researcher should refer to (Dijkstra, 1959) for a detailed explanation of a fundamental algorithm to find the shortest path.

The spanning tree is another interesting concept in graphs. It can be regarded as the subset of the edges that are connected and have no cycles making every vertex in the graph reachable. For algorithmic details, the researcher should refer to (Kruskal, 1956) and (Prim, 1957).

As an application example from the thesis work, while using OpenSceneGraph (OSG), a directed acyclic graph is created for implementing the scene graph in order to store the 3-D virtual environment, the 3-D models that the environment constitutes, graphics rendering tasks in appropriate graph vertices. Therefore, fast storage and retrieval of models in vertices, search of desired vertices and performing all rendering tasks are done efficiently. A spanning tree can be used to represent a 3-D virtual scene and its contents so that the scene will need less memory storage and still all the 3-D scene contents can be reachable via a pointer.

In mathematical perspective, graphs find use in topological processing of meshes composed of several vertices i.e. a mass-spring system representing an elastic model can be thought as a graph such that each mass is a vertex, each spring is an edge connecting masses and finally connection weight of the related graph edge is the corresponding spring constant. Furthermore, in optimization theory, a neural network topology can even be represented as a graph.

For the sake of simplicity, the further details on graph data structures and in general on the graph theory will not be covered here. But the researcher should definitely refer to (Cormen, & et al., 2003, chap. 22, chap. 23, chap. 24, chap. 25, chap. 26, chap. 27, app. B), (Smith, 2004, chap. 7) for very interesting applications in computer science and refer to (Diestel, 2005) for theoretical details.

3.1.3 Trees

Trees can be thought as a special case of graphs. Graph algorithms mentioned in section 3.1.2 are valid with some modifications for trees. Similar to a graph, a tree is a set of vertices. Simply speaking, a tree or a free tree is an undirected acyclic

connected graph as mentioned by (Cormen, & et al., 2003, p. 1085). If the tree is undirected acyclic but unconnected it is called as a forest. A free tree is called a rooted tree if one of the vertices in the vertex set is selected as a root vertex and the rest of the vertices form connected subtrees. Hence, a recursive structure can be noticed at first glance. The depth of a vertex v_i is the length of the path from the root vertex v_r to v_i. If vertex v_c at depth level d is connected to v_p that is on the previous depth level $d - 1$, v_c is the child vertex of v_p and v_p is the parent vertex of v_c.

One of the important types of trees is the binary tree which has at most two children. When it has ordered vertices (any ordering relation can be chosen) in its structure such that the values in vertices at left relative to their parent are smaller and the values in vertices at right relative to their parent are greater than their parent, it is called as binary search tree. Binary search trees are suitable for search purposes in the sense of their algorithmic complexity which is O(logN) for average case analysis and worst case analysis. Other types of trees are red-black trees, B-Trees, random trees, AVL trees and etc… The figure 2.5 shows a binary search tree. The details and information on further tree types and related algorithms can be found in (Smith, 2004, chap. 6). Additionally, (Cormen, & et al., chap 12, app. B) will be a good starting point for further theoretical details.

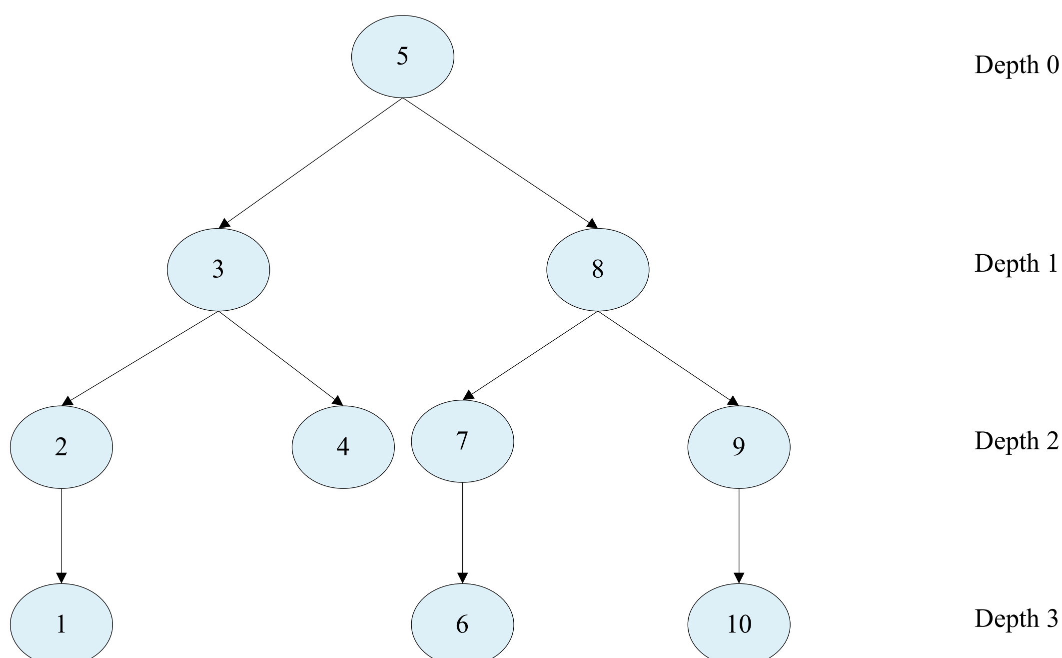


Figure 3.5 Binary search tree.

From the thesis work perspective, a tree is created to implement a scene graph when using Ogre3D to include the virtual world contents and several application specific tasks that are also mentioned in section 2.1.3. In fact, Ogre3D employs an octree by default which is a special type of tree data structure that will be mentioned in the following sections.

3.1.4 Scene Graphs

It should be noted a priori that the term *node* that will be used in this section is equivalent to the term *vertex* used in the previous sections 3.1.2 and 3.1.3. A scene graph is a data structure that is used in simulators and computer games to manage virtual models according to the logical and spatial relationships between them and perform several graphics and physics rendering tasks in the virtual world. Hence a hierarchical representation of the scene data is maintained. Technically, it can be implemented as a directed acyclic graph or as a tree. A scene graph consists of several nodes. A node can technically represent a model, an affine transformation, an animation, sound, a light or any kind of entity that a virtual scene includes. Each transform performed on a parent node affects its child node during the graph traversal in runtime.

Dispatching the transform type, in other words, defining which operation should be performed on a particular node can be done in several ways as depicted in (Wikipedia, 2010a). The transform dispatching is done according to the type of the node. In object oriented programming languages such as C++, virtual functions and runtime type identification techniques are widely used for transform dispatching. These techniques are the implementations of polymorphism property of object oriented programming. Application of the visitor design pattern as explained in section 3.2.1 is another way for transform dispatching. Both ways have pros and cons. For technical information on C++ and object oriented programming, the researcher should refer to (Stroustrup, 2000) and (Stroustrup, 2008). A sample scene graph might be as in figure 3.6.

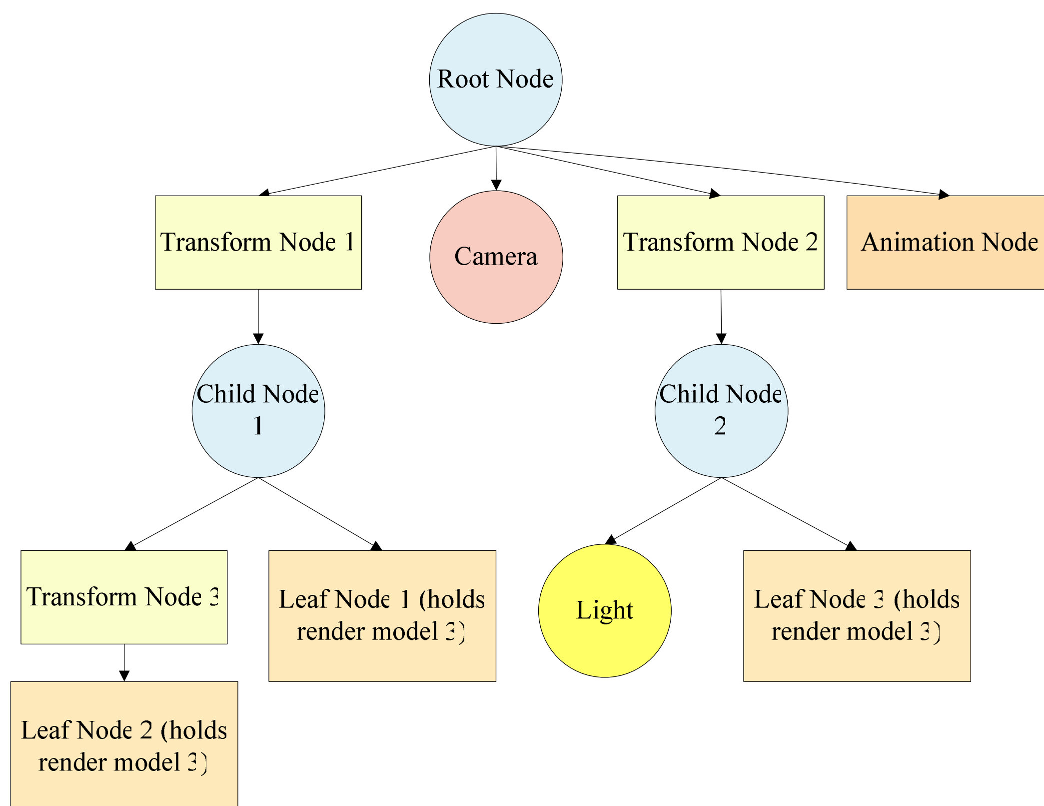


Figure 3.6 A sample scene graph.

The scene graph in figure 3.6 represents a virtual scene with a camera, a light, an animation and three render models. *Camera* has no degrees of freedom in the scene hence it is static and holds camera parameters i.e. near clip plane, far clip plane, perspective parameters and etc…. *Light* is not static as camera for this particular scene and it holds light parameters i.e. light type, light power, attenuation constant and etc…. *Animation Node* deals with the animation related tasks i.e. querying for the key frame, interpolation and etc… *Transform Node 1* and *Transform Node 2* apply affine transformations to *Child Node 1* and *Child Node 2* respectively. A second affine transform is applied to the *Leaf Node 2* by *Transform Node 3*. Notice that *Transform Node 1* effects both *Leaf Node 1* and *Leaf Node 2*; but on the other hand *Transform Node 3* only affects *Leaf Node 2*. So, *Leaf Node 1*, *Leaf Node 2* and *Leaf Node 3* can rotate and translate independent from each other. Additionally, Light undergoes the same affine transformations as with as they are connected to the same node which is *Child Node 2*.

The operations are applied by traversing the scene graph forward from the root node up to the leaf node and then traversing backwards to the root node. When traversing forward to the leaf nodes, pre-render operations are performed; when traversing backward to the root node, post-render operations are performed. Tasks such as culling, depth sorting, render state manipulation, several environmental effects, affine transformation, event dispatching and handling, animation operations are accomplished at different stages of the scene graph traversal. The implementation is specific to the developed scene graph. More information on implementation details can be found in (Foster, 2010).

Some of the scene graphs that are widely used today are OpenSG, OpenSceneGraph, X3D, Java3D, Gizmo3D, RenderWare, NetImmerse Gamebryo, OpenPerformer and Ogre3D. Details on these scene graphs and particularly on the development history of scene graph technology can be found in (Avi, 2007). For implementation and technical details on OpenSceneGraph and Object Oriented Graphics Rendering Engine - Ogre3D both of which are used during the thesis work, the researcher should refer to (Martz, 2007) and (Junker, 2006) respectively. For additional tutorials on OpenSceneGraph and Ogre3D refer to their web sites (OpenSceneGraph, 2010) and (Jacob, 2010) respectively.

3.2 Software Design Patterns

Software engineering and especially object oriented software design rely on the extensive use of software design patterns. Design patterns are the tested, optimum design solutions of the problems that have been come across during the development process in software engineering for years. In this part, important design patterns that are used in many software as well as in scene graphs such as OpenSceneGraph, Object Oriented Graphics Rendering Engine (Ogre3D) and also in GUI development kits such as Qt are going to be introduced briefly in order to understand the simulation development process during the thesis work. For more detailed coverage of the software design patterns, the involved researcher should refer to (Gamma,

Helm, Johnson, & Vlissides, 1995). Additionally, a mathematical theory of software design patterns can be found at (Eden, Gil, Hirshfeld, & Yehudai, 1998).

Design patterns can be grouped into three main classes as creational patterns, structural patterns and behavioral patterns.

When the history of computers is inspected, it can be noticed that the tendency of the progression is towards the easily programmable and reconfigurable systems instead of hard-wired computing devices with fixed functionalities. This is one of the reasons why the programming languages were born. With this thought in mind, creational design patterns can be regarded as design methodologies that contain information about when, how and which primitive objects should be instantiated for the system to perform a specific complex task. Therefore, these types of patterns enable a system to reconfigure itself for more than one task easily. Some examples of creational patterns are abstract factory, factory method, singleton and builder design patterns.

On the other hand, the structural design patterns deal with the composition of classes and objects instantiated. Compositions of interfaces of classes and also of primitive objects to perform different and more complex tasks are the scope of the structural patterns. Composite, proxy, adapter and flyweight design patterns are some examples for this kind of design pattern.

And finally, the behavioral design patterns deal not only with the algorithms the objects implement but also with the flow of control between the objects interconnected to perform more complex tasks. Therefore, as mentioned by (Gamma, & et al., 1995, p. 221), the developer can focus on the way objects interconnected and need not have to deal with the flow of control. Observer, mediator, template method and interpreter design patterns are some examples for behavioral design patterns.

3.2.1 Visitor Design Pattern

The visitor design pattern belongs to the class of behavioral patterns. It establishes an abstraction between the function that contains the defined algorithm and the structure composed of objects that are instantiated from same or different classes on which the visitor will operate. Therefore, in order to add new algorithm, there remains no need to alter the object classes, as indicated by (Gamma & et al., 1995).

It will be better to give an example to motivate the concept. Consider a 3-D simulation software utilizing a scene graph data structure for functional and spatial grouping of several node objects instantiated from different classes, for adding new node objects to the graph and also for performing several rendering operations such as culling, level of detail (LOD) modification, vertex processing, texture processing and altering the transformation matrices of several node objects forming the scene graph. If all these operations are implemented as member functions in each different class, then there will exist unnecessary code overhead that may cause conflicts. As indicated in (Gamma, & et al., 1995), each new operation could be added separately, and also the node hierarchy should be independent of the functions that will operate on them. This will also lead to node objects of different classes that will consume less memory. So, the solution is to develop a class that will contain the necessary operations and then to instantiate an object from that class named as a visitor. The visitor class and the scene graph structure will be independent. When a new functionality is demanded, it will simply be a member of a child visitor derived from the abstract parent visitor. Additionally, no modification will be done to the graph node classes. Therefore, a minimal code development effort will be needed.

The visitor pattern works simply as follows. It contains two hierarchies of classes. The different types of objects forming the data structure are instantiated from the first class hierarchy which is the object hierarchy. The visitors that embody the necessary functionalities and operate on the data structure's objects from different classes are instantiated from the second class hierarchy which is the visitor hierarchy. The function that will be called when the visitor is accepted by the object of the data

structure – in the above example, the nodes of the scene graph- is selected using the concept of *double dispatch.* This means that, the visit function call of the visitor object is done considering the function signature, the runtime type of both the visitor and the visited object.

For the general implementation case, the object class hierarchy and visitor class hierarchy implementation details can be followed from figure 3.7 which is taken from (Gamma, & et al., 1995). In this figure, the visit functions are declared in *Visitor class*. The implementations of the declared functions are done in *ConcreteVisitor1* and *ConcreteVisitor2 classes* which inherit from *Visitor class*. These functions are called according to the double dispatch concept defined above. Element class declares the accept function. The implementation of that function is done in *ConcreteElementA* and *ConcreteElementB classes*. *ObjectStructure* hold *Element objects* together.

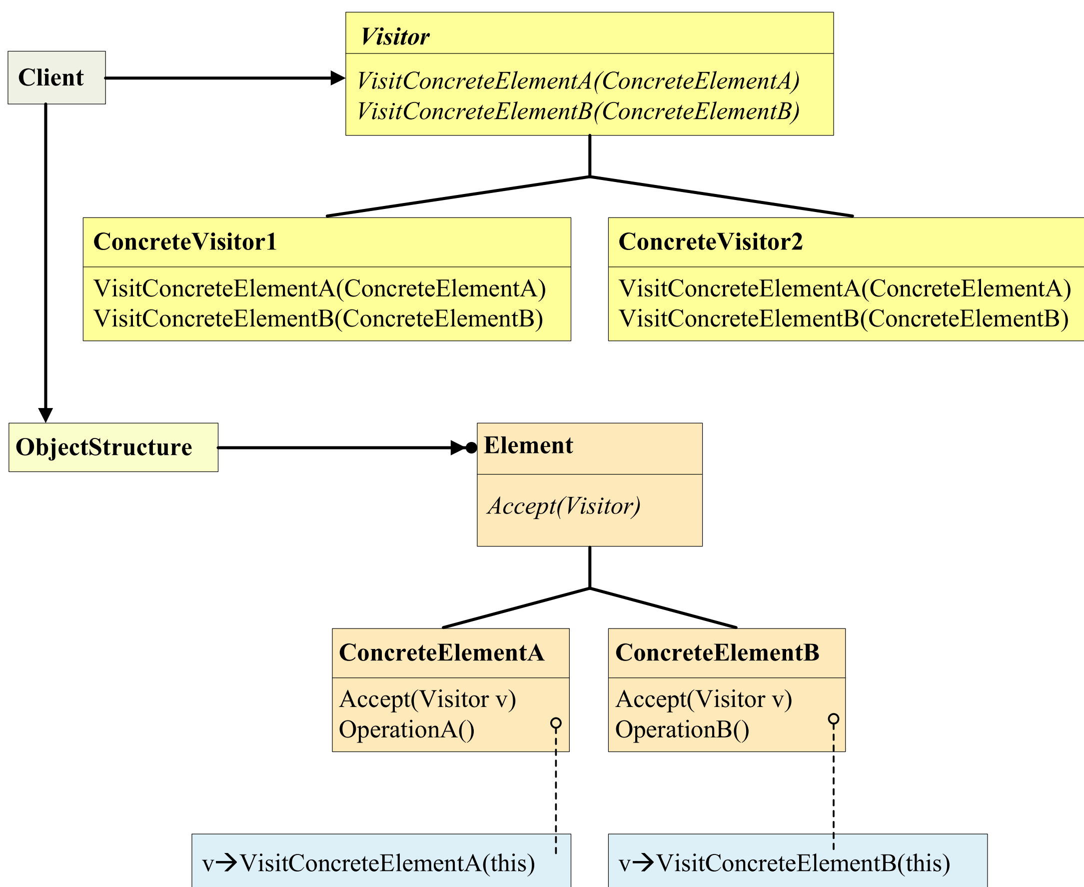


Figure 3.7 Visitor design pattern class diagram (Gamma, & et al., 1995, p.334).

For the specific 3-D simulation software example given in the preceding paragraphs, Visitor corresponds to the parent class of visitor classes specialized for culling, coordinate transformation computing, texture processing and so on. These specialized visitor classes in the example correspond to *ConcreteVisitor1* and *ConcreteVisitor2* classes in figure 3.7. *ObjectStructure* corresponds to the scene graph, *Element* corresponds to the nodes, *ConcreteElementA* and *ConcreteElementB* correspond to nodes derived from a parent class and *Client* corresponds to the application.

3.2.2 Observer Design Pattern

The observer design pattern belongs to the class of behavioral patterns. This pattern is composed of at least two objects instantiated from subject and observer classes respectively. Subject and its observers are decoupled. Additionally, the objects instantiated from observer classes are independent of each other. This structure leads to an increased reusability. The observer objects are registered with the subject object. The goal of the pattern is to define a relationship between the subject object and its observer objects so that when the subject object changes its state, the observer objects that depend on the subject object are notified and their states are updated automatically.

The subject object encapsulates the data. On the other hand, the observer objects encapsulate their own member functions that operate on the data encapsulated by the subject object. As mentioned by (Gamma, & et al., 1995), this pattern can be used when there are two abstractions one dependent on the other so that encapsulating these abstractions in separate objects increases reusability and the independent modification of the object classes. The pattern is also suitable for cases when a change in one object requires a change in the dependent objects without the knowledge of the number of dependent objects and without the knowledge of who those objects are.

The class diagram of the observer pattern is summarized in figure 3.8 which is taken from (Gamma, & et al., 1995). In the figure, *Subject class* provides a registration interface for any number of observers; on the other hand *Observer class* provides an updating interface for the subject state change notifications. *ConcreteSubject* and *ConcreteObserver classes* are child classes of *Subject* and *Observer classes* respectively. *ConcreteSubject* stores the state in which *ConcreteObserver objects* are interested; and it notifies them when that state changes. *ConcreteObserver* stores the state it is interested in and implements observer update interface to synchronize that state with *ConcreteSubject*.

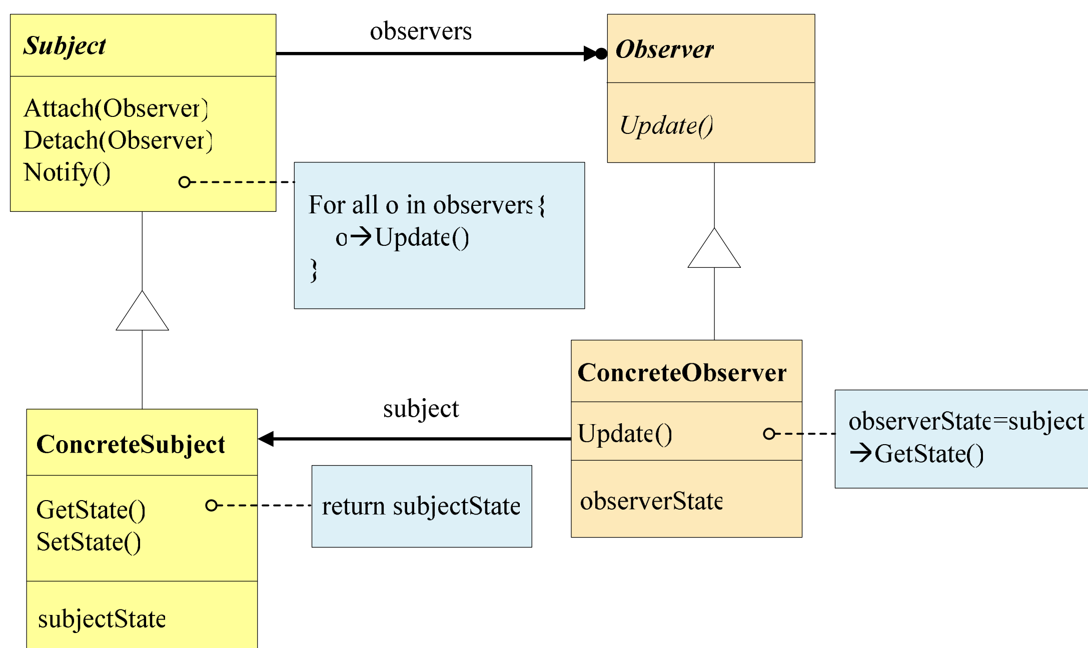


Figure 3.8 Observer design pattern class diagram (Gamma, & et al., 1995, p. 294).

Two of the application examples where this pattern is used in the scope of this thesis are as follows. The first of them is the graphical user interface development by using Qt Toolkit. The signal / slot model of Qt Toolkit implements the observer design model. In Qt, the controls can send signals to other controls for notifications. The signals contain the event information and the slots contain the functions for state update as depicted by (Blanchette & Summerfield, 2008). The second section where the observer pattern used is the Ogre3D rendering engine. Several observers are registered to the corresponding subjects within the engine in order to receive notifications upon state changes during the simulation and then act accordingly. For

instance, (Junker, 2006, p. 38) depicts that FrameListener is a way to notify the application about the frame-started and frame-ended events during the simulation in Ogre3D rendering engine.

3.2.3 Singleton Design Pattern

The singleton design pattern belongs to the class of creational patterns. It can be thought as the implementation of mathematical concept of singleton in which a singleton means a set with only one element. This is depicted at (Wikipedia, 2010b). The term singleton has also correspondences in set-theoretic construction of natural numbers, in axiomatic set theory and in topological constructions in mathematics as mentioned by (Wikipedia, 2010c).

This pattern finds use when there is a need for only one instance of a certain class and only one access to that instance. One might think that declaring a global variable can satisfy this need, but as depicted by (Gamma, & et al., 1995, p. 127), although object that is accessible is instantiated, declaring a global variable does not guarantee preventing multiple object instantiations.

The class diagram of the pattern is given in figure 3.9 which can be found in (Gamma, & et al., 1995, p. 127). In the figure, *Singleton class* defines *Instance operation* and lets its clients to access its data.

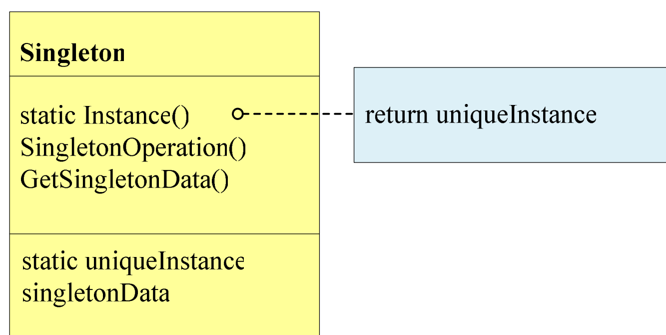


Figure 3.9 Singleton design pattern class diagram (Gamma, & et al., 1995, p. 127).

As an example, the rendering system of Ogre3D used throughout the thesis work can be considered. The engine implements the rendering system using the singleton design pattern, because there should be only one instance of the render system and only one access point for the clients of that system.

3.2.4 Factory Method Design Pattern

The factory method design pattern belongs to the creational patterns class. It defines an interface in a parent class for instantiating an object without defining its class. The subclasses can override the creating function which is named as factory method to define the class of object that will be instantiated. Therefore, not only a common interface is established between different classes from which objects are instantiated, but also flexibility is gained in application by delegating the subclasses to take the responsibility of knowledge of object instantiation and freeing the parent class from estimating which classes might be needed in the application for the future. As a result, each new application developed using the participant classes of the factory method design pattern can derive a class with different functionality when a need occurs without breaking the common interface persistent in the application or in the framework.

The class diagram of the factory method design pattern is given in figure 3.10 which is taken from (Gamma, & et al., 1995, p. 108). In the figure, *Creator class* declares the factory method to instantiate an object from *Product class*. *Product* declares the interface for objects instantiated by the factory method. *ConcreteCreator* is the subclass of *Creator class*. It is responsible for the implementation of the factory method to instantiate *ConcreteProduct object*.

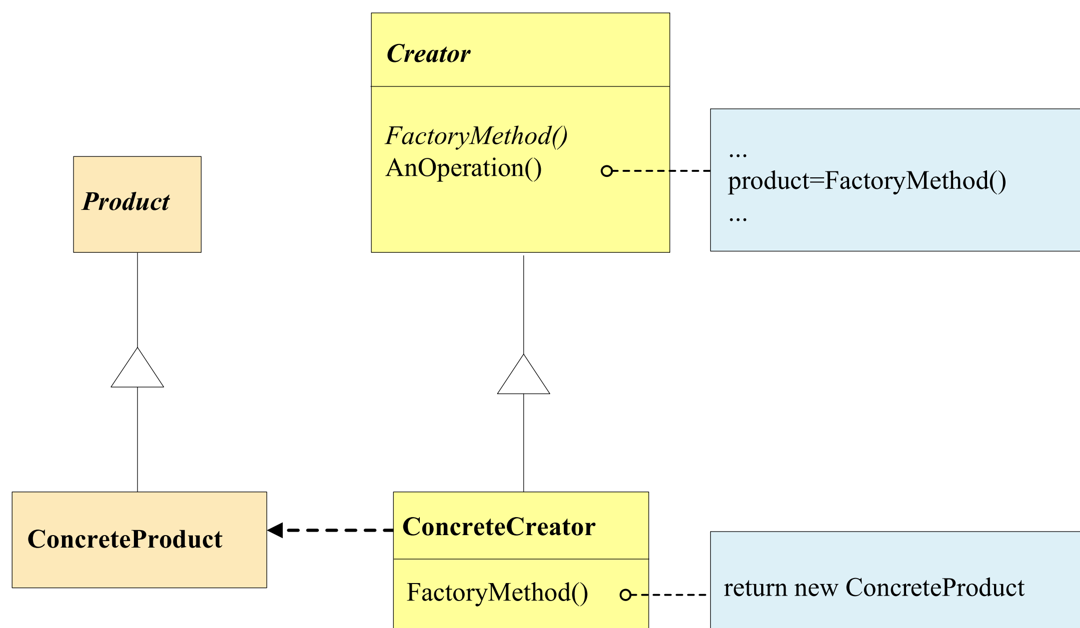


Figure 3.10 The factory method design pattern class diagram (Gamma, & et al., 1995, p. 108).

Many software frameworks are examples where the factory method design pattern is used extensively. Ogre3D rendering engine widely uses this pattern to create instances of abstract interfaces as depicted by (Junker, 2006, p. 38). For example, Scene manager API of the engine acts as a factory for instantiating different objects such as cameras and lights which is also mentioned by (Junker, 2006, p. 57).

3.2.5 Iterator Design Pattern

Iterator design pattern is one of the behavioral patterns. The pattern aims to access the elements of an aggregate object without the need for the knowledge of the inner structure of that object. An aggregate object is an object instantiated from a class with no user constructor, no private or protected non-static data members, no parent class and with no virtual functions. Two examples are lists and vectors. Detailed explanation with an example can be found at (Wikipedia, 2010d).

In addition to accessing the elements of an aggregate object, in many cases, there will be a need for traversing the elements in different directions or a need for multiple traversals on the aggregate object. These tasks can be accomplished trivially

by encapsulating each different traversing algorithm in each of the aggregate classes. The end result will be unwanted increase in the code size, difficulty in development and maintenance of the software. Instead, the traversing algorithm can be decoupled from the aggregate class and it can be put into an iterator. By this way, a need to modify the aggregate class won't exist any more and all different traversing algorithms can be put into the iterator class as mentioned in (Gamma, & et al., 1995, p. 258).

Polymorphic iteration is the key concept in this pattern that decouples the aggregate object and the iterator. Therefore, the iterator does not have to know the particular type of the aggregate object it is traversing. Hence a uniform and transparent interface for traversing aggregate objects instantiated from different classes is maintained. Therefore, the iterator class does not need to be modified when a change occurs in the class of the aggregate object being traversed.

The class diagram of the iterator design pattern is given in figure 3.11 which is taken from (Gamma, & et al., 1995, p. 258). In the figure, *Iterator class* declares an interface for traversing elements. *ConcreteIterator* class not only implements the interface that *Iterator class* declared but also knows the position in the current traversal of the elements of *aggregate object*. *Aggregate class* declares an interface for creating *Iterator object*. *ConcreteAggregate class* implements that interface in order to create suitable *ConcreteIterator object*. One point should be considered here. As seen in figure 3.11, the factory method pattern is used in the *Aggregate class* hierarchy in order to create the appropriate *ConcreteIterator object*; *Aggregate class* has no knowledge which *ConcreteIterator object* to create at compile time. That task is passed to *ConcreteAggregate class*. *ConcreteAggregate classes* create suitable *ConcreteIterator objects* at runtime.

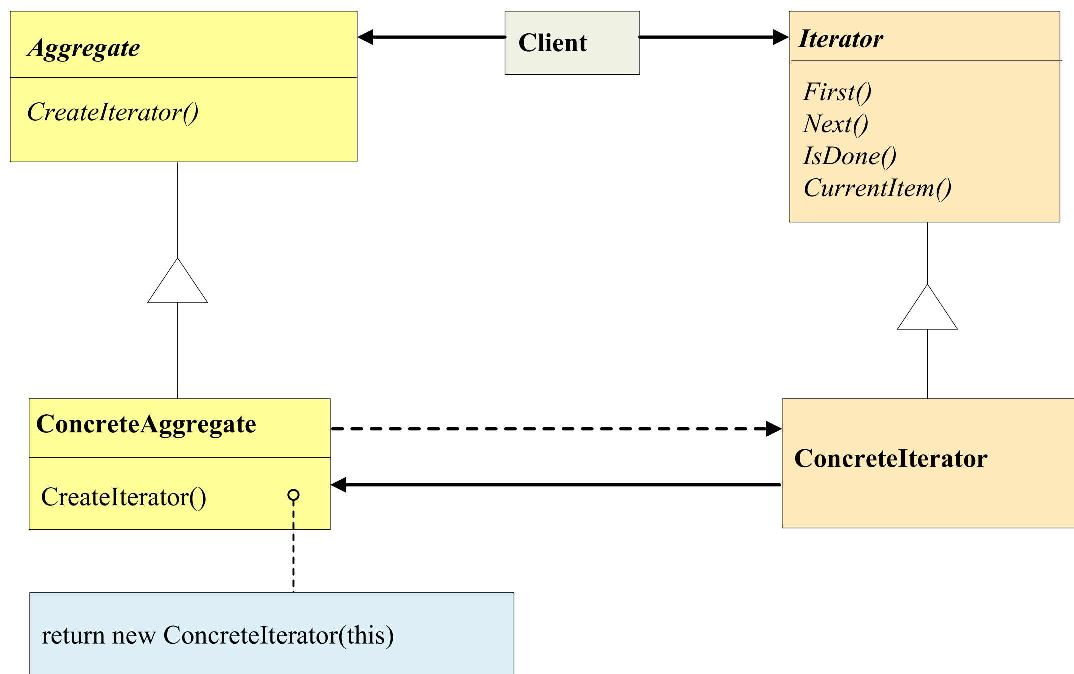


Figure 3.11 Iterator design pattern class diagram (Gamma, & et al., 1995, p. 258).

As an example, C++ Standard Template Library (STL) uses this pattern to access and traverse the elements of objects instantiated from template classes such as vector, map and list. C++ STL is widely used to keep track of the render objects and their corresponding collision shapes in appropriate data structures like maps. The elements of maps and other data structures implemented in C++ STL can easily be accessed and traversed by appropriate iterators. Similarly, in Ogre3D engine, the elements of the scene graph are manipulated and traversed using the appropriate iterators.

3.2.6 The Façade Design Pattern

This pattern belongs to the class of structural design patterns. For the researches interested in the word *façade*; the word is from the French Language in which it is used to mean the exterior of building. This explanation will certainly make things clear for understanding the pattern. The detailed explanation of the literal meaning can be found at (Wikipedia, 2010e). The aim of the pattern is to provide a simple interface enabling the client objects to access the subsystems of a complex system thereby, abstracting the clients from the complexity of the subclasses. What a client

sees is just one simple interface that has the ability to get the full potential of the subsystems. On the other hand, the subclasses in the system are unaware of the façade object they are communicating with. Although this pattern simplifies the development process, it may limit benefiting from the full potential of the subclasses; because providing one simple interface to all subsystems may limit customizability of the subsystems. Therefore, the pattern should have a second access point to the subsystems for the clients wanting to customize and more functionalities to the subsystems.

The class diagram of the façade pattern is given in figure 3.12 which is taken from (Gamma, & et al., 1995, p. 187). In the figure, *Façade class* is responsible of transmitting requests of the clients to the appropriate subsystem classes. Upon receiving the request from the object instantiated from Façade class, the subsystem objects perform the related tasks.

The façade design pattern is used in Ogre3D rendering engine to implement *Root class*. The client object can access the required functionality of the root object instantiated from *Root class*. Therefore, a simple interface for the client is established to use the various functions of the rendering engine. The detailed explanation of the concept can be found at (Junker, 2006, p. 46).

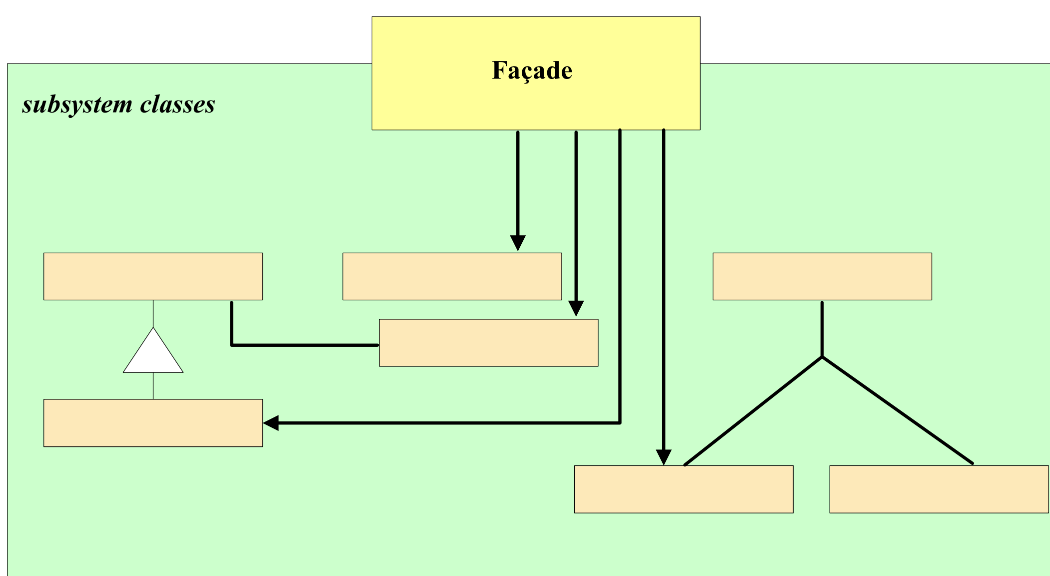


Figure 3.12 Façade design pattern class diagram (Gamma, & et al., 1995, p. 187).

3.3 What is a Software Engine?

The visual rendering and physics rendering tasks of the virtual environments in many simulation software are accomplished via independent set of functions in two independent software assemblies aimed for a common task. These software assemblies are named as graphics engine and physics engine respectively.

Considering the explanations in the previous sections of this chapter, the concept of engine can be explained as follows. Engine in software is a collection of modules that implement required data structures and algorithms and designed by benefiting from the related software design patterns where necessary to accomplish a common task. These tasks can be graphics rendering, physics rendering as well as video processing and audio processing.

Figure 3.13 depicts the first real time graphics engine coded by means of getting reference from (Seddon, 2005) for creating a 3-D virtual environment. This practical study helped in understanding the composition of the software modules, communication between the software modules and working principles of a simple graphics engine.

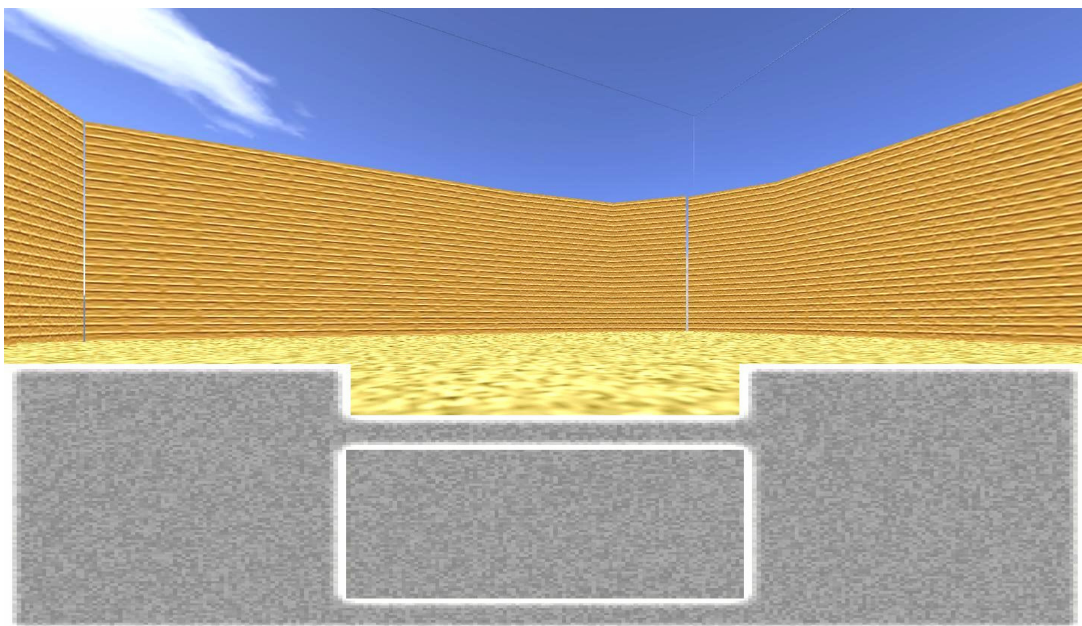


Figure 3.13 A 3-D scene from the first real time graphics engine coded for in-depth study.

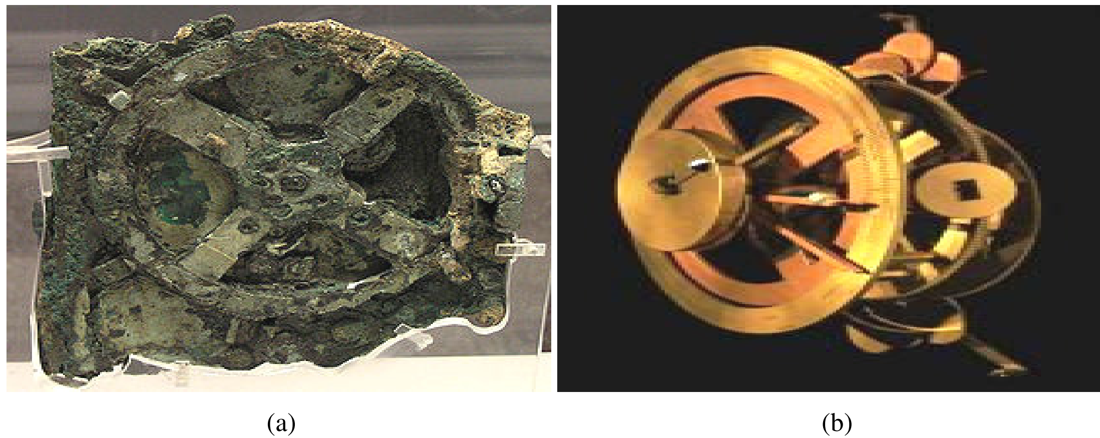
CHAPTER FOUR

GRAPHICS PROCESSING UNIT PROGRAMMING FOR GRAPHICS AND GENERAL PURPOSE COMPUTING

Graphics processing units (GPUs) have undergone a rapid evolution period since late 90s up to now. Today, GPUs are far beyond a simple hardwired 2-D rendering control units. They evolved into programmable massively parallel computational processors with their flexible architecture specialized for matrix and vector calculations and with their own programming languages. This chapter serves as a survey to understand the fundamentals in computation and to leverage the power of modern GPUs. The development history of “the computation” and of GPUs, the hardware architecture of modern GPUs, the benefits of parallelism, the use of GPUs for graphics and general purpose computing, the need for high level programming languages for GPUs and the related programming languages are briefly covered in this chapter.

4.1 Short History of Computing Machines – From Antikythera Mechanism to Today’s Massively Parallel GPUs

Accurate and fast computing has always been a need for humans since ancient ages. The oldest computing machine discovered so far and named as the Antikythera Mechanism belongs to Ancient Greek. It is thought to have been built at about 150 – 100 BC with the intent to calculate the cycles of the Solar System and astronomical positions. With its complex mechanical gear structure, it is accepted as the first known analog computer. Figure 4.1 depicts the main fragment and the 3-D rendering of the machine. No sign of such technically complex computing machine was found until the 14th century when mechanical astronomical clocks appeared in Europe. An astronomical clock invented by Al-Jazari in 1206 is considered to be the first programmable computer as depicted by (Wikipedia, 2010f). For further details, the researcher should refer to (Freeth, Jones, Steel, & Bitsakis, 2008), (Wikipedia, 2010g) and (Edmunds, 2010).

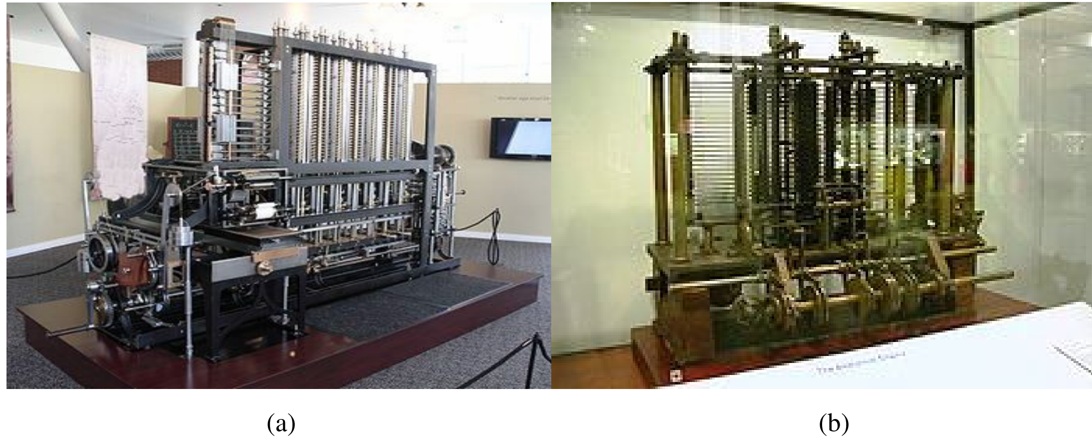


(a) (b)

Figure 4.1 (a) The main fragment of the Antikythera Mechanism (Wikipedia, 2010g). (b) The 3-D rendering of the complete computing machine (Edmunds, 2010).

For the investigation of the roots of today's powerful processors and programming languages, a long jump is needed from the mid ages to the time of Charles Babbage (26.12.1791 – 18.10.1871). Babbage was an English mathematician, philosopher, inventor and mechanical engineer who originated the concept of programmable computer (Wikipedia, 2010h).

The inspiration for his inventions was mostly due to the high error rate of calculations performed by humans at that time. His intent was to mechanically calculate mathematical tables to prevent the human errors. Towards this aim, he began building a special purpose mechanical machine which he named as the Difference Machine in 1822. The difference machine can automatically calculate polynomial functions. As polynomial functions can also be used to approximate trigonometric and logarithmic functions, the machine would find a very wide usage area. The operation principle of the difference machine was based on Newton's divided differences. If the initial value of a polynomial (and of its finite differences) is calculated by some means for some value of X, the difference engine can calculate any number of nearby values using the method of finite differences (Wikipedia, 2010i). Therefore there was no need for multiplication and division during the computations. In addition to this technically complex machine, Babbage also designed a printer for his difference engine that is highly complicated for the 19th century. The reconstruction of the difference engine is seen in figure 4.2 (a).

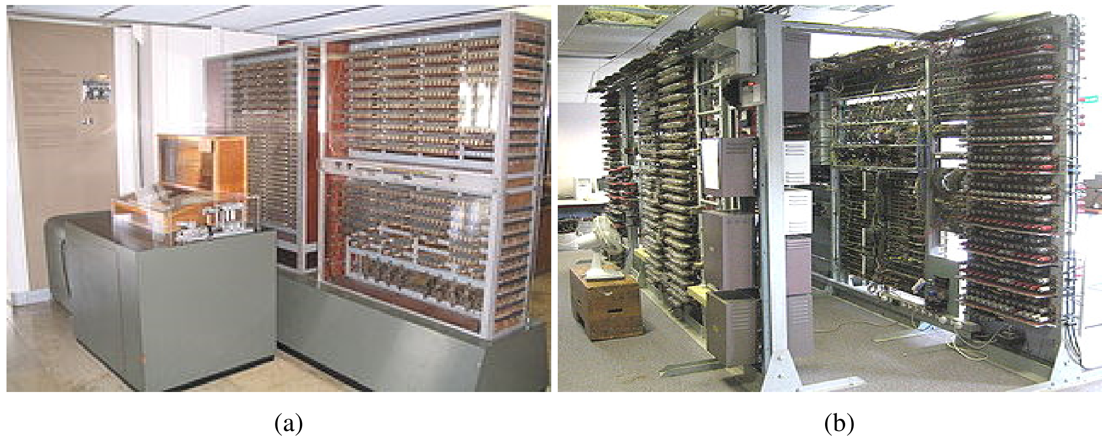


(a) (b)

Figure 4.2 (a) The fully operational Difference Engine at Computer History Museum in Mountain View, CA. (Wikipedia, 2010i). (b) A trial model of a part of the Analytical Engine at the Science Museum, London (Wikipedia, 2010j).

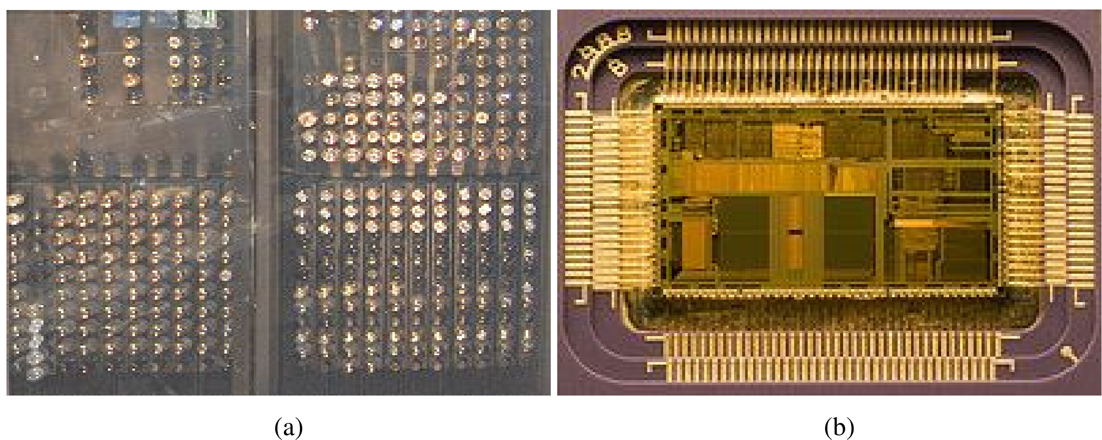
In 1837, Babbage designed the first version of his second computing machine named as the Analytical Engine. A trial model of a part of the Analytical Engine is seen in figure 4.2 (b). This machine is the first mechanical general purpose computer. The important point when compared to the Difference Engine which was a special purpose machine was that the Analytical Engine was programmable via punched cards. Ada Lovelace was the first mathematician and the first computer programmer who first wrote a program to compute a sequence of Bernoulli Numbers for the Analytical Engine. The Analytical Engine had several features such as sequential control, conditional branching and looping in addition to mechanical units to implement today's memory units, arithmetical logical units (ALUs) for arithmetic operations and comparisons and optionally for square roots calculations. The complex instructions that the user's program includes are computed by the ALU of the machine which was a mill relying on its own internal procedures. The punched cards on which the user's program was written were of three different types aimed for arithmetical operations, numerical constants and read write operations. These punched cards were being inserted into their own readers on the machine. For more information, the researcher should refer to (Wikipedia, 2010j). The language used by the machine can be regarded as the origin of the today's assembly language. As the machine has support for conditional branching and memory read write operations, the machine can be called as Turing complete in the context of Computability Theory. More on this can be found at (Wikipedia, 2010k).

Analog computers were being used in the 20th century for the computations regarding scientific problems. Those machines used mechanical or electrical model of the scientific problem for computation. But important point was that, they were not programmable and not accurate. Hence they can be considered as specific purpose computing machines. One of the first steps in 1937 towards today's digital computing machines was a relay based calculator named Model K whose designer was George Stibitz. It was the first model that used binary circuits to perform arithmetic operations. The other important step was the programmability. The first programmable, fully automatic computing machine was the electomechanical device Zuse Z3 designed by Konrad Zuse. It performed binary arithmetic and floating point arithmetic. It was a program controlled device that used punched cards. The picture (a) of figure 4.3 shows Zuse Z3 replica. Following Zuse Z3, the non-programmable Atanasoff Berry computer designed in 1941 was important for its vacuum tube based computation, binary numbers and its regenerative capacitor memory that allowed a feed back mechanism to be established for feeding back the stored elements into computation. The period of World War II witnessed many technical improvements in programmability and hardware of computing machines. For breaking German secret ciphers, the British Colossus computers were developed in 1943. The picture (b) of figure 4.3 shows a Colossus rebuild. It had limited programmability capabilities, but thousands of vacuum tubes in its architecture were reliable and electronically reprogrammable. The Harvard Mark I computing machine developed in 1944 was another important electromechanical device with limited programming capabilities. ENIAC that was designed in 1946 at the U.S Army's Ballistic Research Laboratory was the first general purpose computer that would highlight the future designs. The handicap of the device was its inflexible architecture and the need to change the wiring to reprogram the device. The picture (a) of figure 4.4 shows the vacuum tubes of ENIAC.



(a) (b)

Figure 4.3 (a) Zuse Z3 replica at Deutsches Museum in Munich (Wikipedia, 2010l). (b) A rebuilt version of the Colossus (Wikipedia, 2010m).



(a) (b)

Figure 4.4 (a) The vacuum tubes of ENIAC (Wikipedia, 2010n). (b) A die of Intel 80486DX2 microprocessor (Wikipedia, 2010f).

Up to now, it is seen that two important concepts in computing was programmability and hardware architecture (application of electromechanics, vacuum tubes, etc…) of the devices.

At this point the mathematical theories and ideas of two mathematicians gain importance. Those mathematicians namely John Von Neumann and Alan Turing were considered to be the fathers of computer science. The theoretical works of Neumann and Turing should be definitely studied by anyone scientifically interested in computer science and mathematics. Those details will not be covered here as they are out of the scope of the thesis research.

The inflexible architecture and the need for rewiring for programming were potential problems of ENIAC. The solution was John Von Neumann architecture which was a model for stored program architecture. Stored program architecture refers to a computing machine that has built-in instruction set and to a memory to write and read the program and the data for the computations. The theory and technical details of this architecture can be found in (Neumann, 1945). Although the semiconductor technology that is the base of computers has advanced from 1940s up to now, the architecture of most computers used today is exactly the von Neumann architecture or modified version of the von Neumann architecture. A die of Intel 80486DX2 which can be considered as an advanced semiconducter technology for 1990s is seen in the picture (b) of figure 4.4. In von Neumann architecture, the instructions and data are stored in the same memory unit that can be read and written. This is in contrast to Harvard architecture where the instructions and data are stored in seperate memory units. In fact, Alan Turing had previously described the stored program concept. At this point, his paper (Turing, 1936) is an important resource for the researcher. In that paper, he describes a hypothetical machine with an infinite memory in which both instructions and data are stored. In the literature, this hypothetical machine is called as Universal Turing Machine. In 1946, both Alan Turing's Automatic Computing Machine (ACE) and the other computing machine EDVAC in which John Von Neumann participated in its development process, used the stored program concept in their designs.

In addition to flexible programmability, error free programming is an important concept for computing machines. In the early days, the programs were being written directly in machine code in which each instruction was represented with an unique number namely with its opcode. Although this technique was used in early computing machines, it had high error probability especially as the complexity of the programs evolved. The next programming technique was to write the program in the computing machine's assembly language in which each instruction was given a short name identifying its function. For complex programs, assembly language was also error prone. Together with the machine language, assembly language were low level

programming languages targeted for a specific machine. What if someone wanted to port a program developed for a specific machine to another one?

At that point, high level languages and specific design patterns gained importance as a solution. High level languages like C, Fortran, C++ or Java abstract the programmer from the hardware details of the computing machine, hence the programmer could focus on the main problem to be solved. Additionally, the developments in high level languages were more error free and portable across different computing platforms. The history, technical details and related links of the preceeding paragraphs can be found in (Turing, 1936), (Neumann, 1945), (Wikipedia, 2010h), (Wikipedia, 2010f) and (Wikipedia, 2010k).

A reader might think that the preceeding paragraphs might be long for a scientific history of computing machines. But when inspected, beginning from 1970s until today it can be seen that the evolution of graphics processing units (GPUs) has correspondances more or less with the evolution of modern computing machines that are central processing units with appropriate peripheral units.

The evolution of special purpose processors for graphics began with ANTIC and CTIA chips produced for hardware control of mixed graphics and text modes on Atari 8-bit computers in 1970s. In 1984, IBM released its first 2-D/3-D graphics accelerator namely IBM Professional Graphics Controller (PGC) as seen in figure 3.5 which is taken from (Elliot, 2010). Technical details of PGC can be found in (Wikipedia, 2010o). IBM 8514 video card was one of the pioneers that implement 2-D primitives in hardware. At this time, Commodore Amiga has its own full graphics accelerator and graphics coprocessor with its own primitive set that offloads all video generation functions to hardware. Prior to this, those tasks were being handled by central processing unit (CPU).

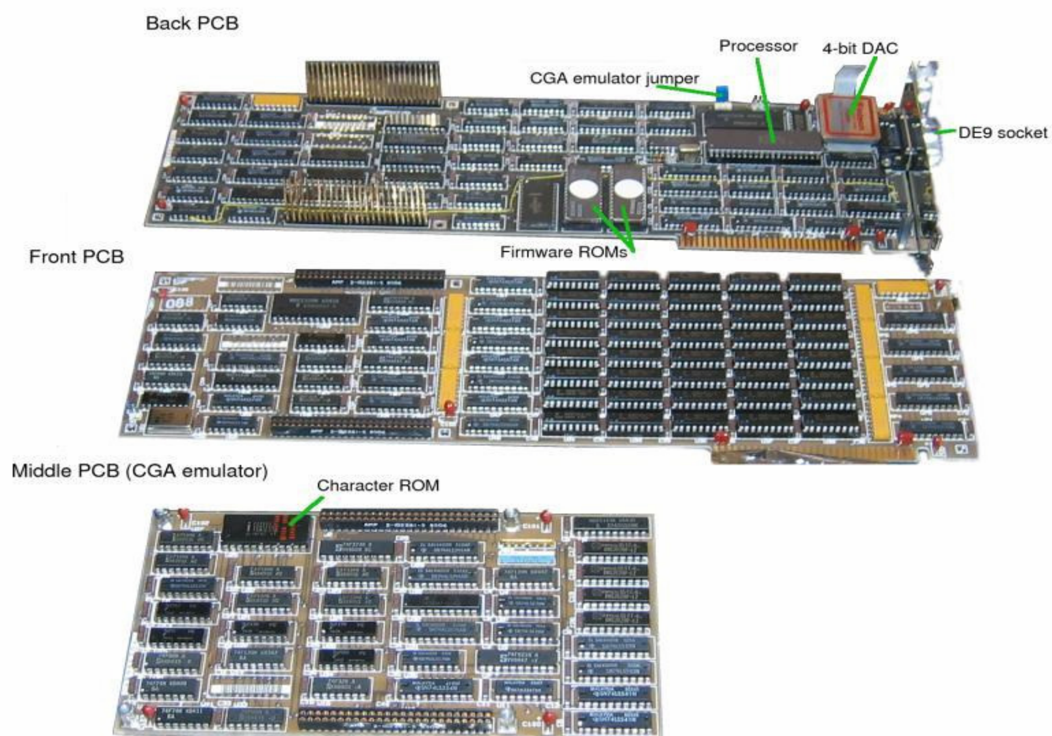


Figure 4.5 Three layers of IBM's first 2-D/3-D accelerator PGC released in 1984 (Elliot, 2010).

Beginning from 1990s, OpenGL and Microsoft DirectX became the horsepower of hardware development. OpenGL had both software and hardware implementations. The detailed development history can be found at (Wikipedia, 2010s) and (Fernando & Kilgard, 2003, chap. 1). Up to late 1990s, the GPUs in this period were capable of rasterizing pre-transformed triangles and one of two textures. GPUs were performing pixel updates instead of central processing units (CPUs). But on the other hand, they lack of adequate set of math operations for computing rasterized pixel color. Additionally, CPUs were still performing vertex transformations.

At the end of 1990s, both vertex transformations and lighting has begun to be done by GPUs instead of CPUs. OpenGL and Microsoft DirectX supported hardware vertex transformation. The hardware in this period were configurable rather than programmable. As in the mid 1990s, although the set of mathematical operations that the GPUs support in hardware improved, they were not adequate for complex texture and pixel color operations.

By the early 2000s, GPUs began to support vertex programmability. This was an important step, because rather than using only the predefined OpenGL or Microsoft DirectX transformation and lighting techniques, from there on, the developers would be able to define a program for transforming the vertices according to their needs. On the other hand, pixel programmability was still impossible. Only OpenGL and Microsoft DirectX were supporting their pre-defined pixel level configurability.

Towards the mid 2000s, GPUs were supporting not only vertex programmability but also pixel programmability. Therefore, CPUs completely released vertex transformation and pixel shading operations to GPUs. In addition to this, both OpenGL and Microsoft DirectX began to support vertex-level and pixel-level programmability.

4.2 Shaders

At this point an important technical term gains importance that is “shader”. Shaders are set of instructions that is used to program the programmable pipeline of the GPU. Technical details can be found in (Möller, & et al., 2008, chap. 2, chap. 3), and (Wikipedia, 2010p). There are three types of shader.

Vertex shaders are run for each vertex that is transferred to the GPU. The developer can code a vertex shader for transforming the vertices according to the needs. No topology change that is addition or removal of a vertex can be done in this stage. The output of the vertex shader is either transferred to the rasterizer or if exists to the geometry shader along the graphics pipeline of the GPU. Geometry shaders can perform a topology change. Geometry shaders are set of instructions that are used to generate geometry or add volumetric details to the existing geometry that will be too costly if done on the CPU. The output of a geometry shader is transferred to the rasterizer along the graphics pipeline. Finally, fragment shaders (pixel shaders can be used interchangebaly although not appropriate) are set of instructions that are used to calculate the color of each pixel. The input to the fragment shaders are from the rasterizer. The rasterization stage uses the vector graphics that are polygons to

generate a raster image that are composed of pixels to be displayed on a display. Fragment shaders are used for lighting, several graphics effects like bump mapping and for other application specific transformations for pixel color. As it is seen developer can develop necessary programs for appropriate shaders for each pixel that will be seen on the screen.

4.3 Fixed Function Graphics Pipeline and Programmable Graphics Pipeline Architecture in Detail

Geometry shaders will be shown partly connected in the figures of this section as geometry shaders were not used in the shader models prior to Shader Model 4.0 as depicted in (Möller, & et al., 2008 p. 41). The fixed function graphics pipeline is seen in figure 4.6.

In figure 4.6, the 3-D application layer can be a simulation, a game or etc… using OpenGL API or Microsoft Direct3D API high level instructions to process the scene. These APIs decompose complex meshes into triangle primitives and then send necessary low level instructions along with the data stream to the GPU via the communication bus between CPU and GPU. In the GPU front end, the vertices are transformed into a common coordinate system for further transformations and lighting. Only affine transformation is performed in this stage in order not to twist triangles into curled shapes. At the vertex transformation stage, other necessary geometric transformations are done and the vertices are transformed into the screen coordinate space for the rasterizer. Texture coordinates and vertex lighting are also completed in this stage for texturing and vertex color calculation respectively. The output of the vertex transformation stage is input to the primitive assembly stage along with the vertex indices for generating triangles, lines or points. These primitives are input to the rasterization stage.

In the rasterizer, the primitives are either clipped to the view frustrum or application defined clipping volume. This process is called as *clipping*. *View frustrum* is a pyramid that is cut beneath its apex by a near clip plane and a far clip

plane forming its base. View frustrum represents the 3-D scene that the camera in other words the viewer observes at a particular time. It can be configured by field of view angles. Additionally, the rasterizer may discard primitives according to their face orientation that is either front face, back face or none of the faces are discarded. This called *culling*. Clipping and culling is important for reducing the number of primitives that will be transferred to the later stages in the pipeline in order to decrease the work load of the following stages.

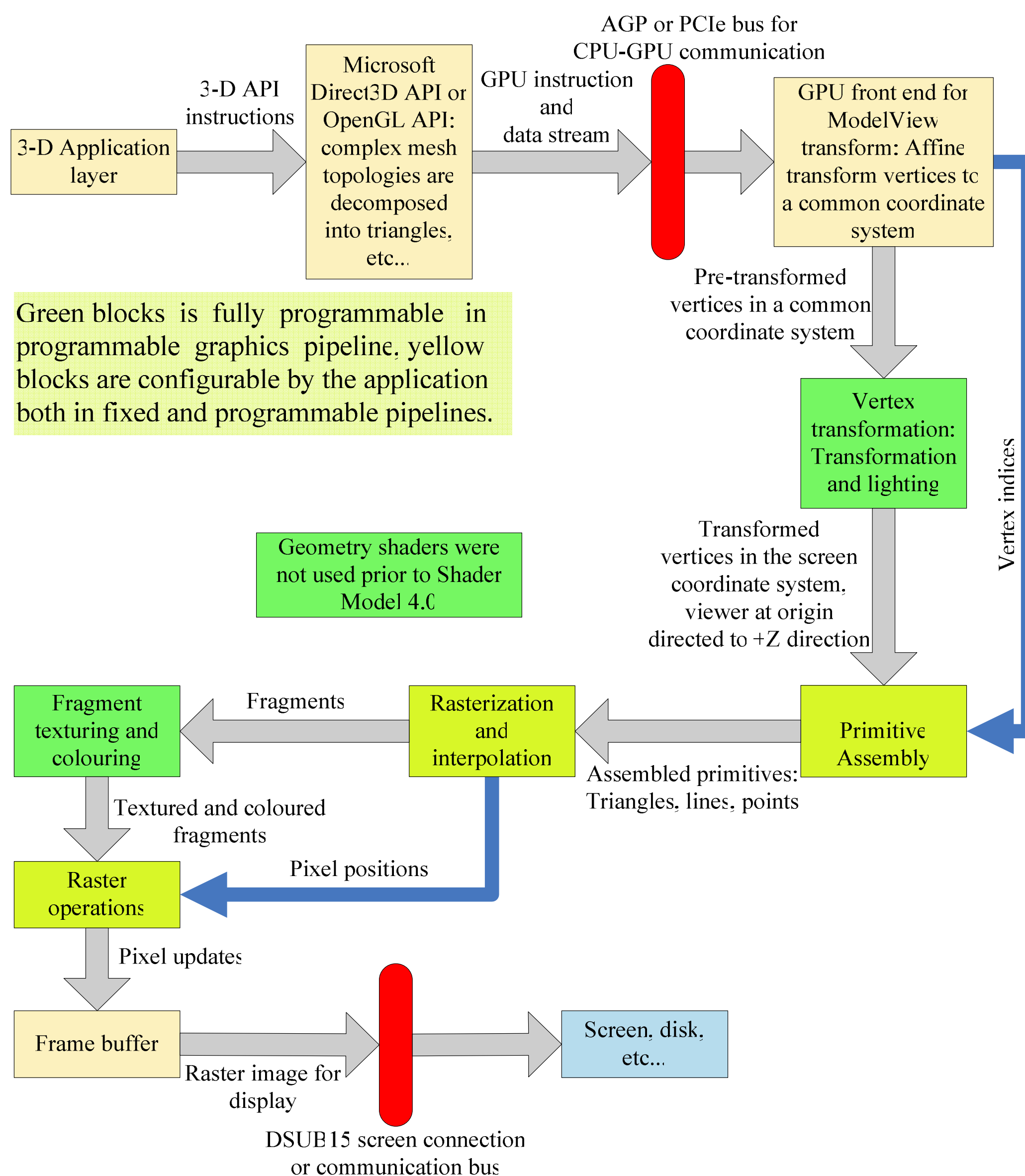


Figure 4.6 Fixed function graphics pipeline.

The rasterization stage calculates the pixels in the screen that is covered by the primitive and the fragments needed to update the pixel locations. Fragments

generated in this stage are used for pixel update decision. Consider that the pixel p_i is in the rasterizer output set W and the rasterizer input set is V in which a geometric primitive vertices are contained. Then;

$$\forall p_i \in W, \forall v_i \in V, p_i = \sum_{n=1}^{k} \alpha_n v_n \ni \sum_{n=1}^{k} \alpha_n = 1, \alpha_n \geq 0 \text{ where } k = \begin{cases} 1 & \text{, if point} \\ 2 & \text{, if line} \\ 3 & \text{, if triangle} \end{cases}$$

The fragment texturing and final colour calculation task is performed in the fragment colouring and texturing stage. A depth value may be defined, the fragment value may conditionally be discarded or not. The output of this stage is one or zero coloured fragments for each of the input fragment. These output fragments are processed by raster operations that are shown in figure 4.7. These operations are common both in Microsoft Direct3D and OpenGL APIs.

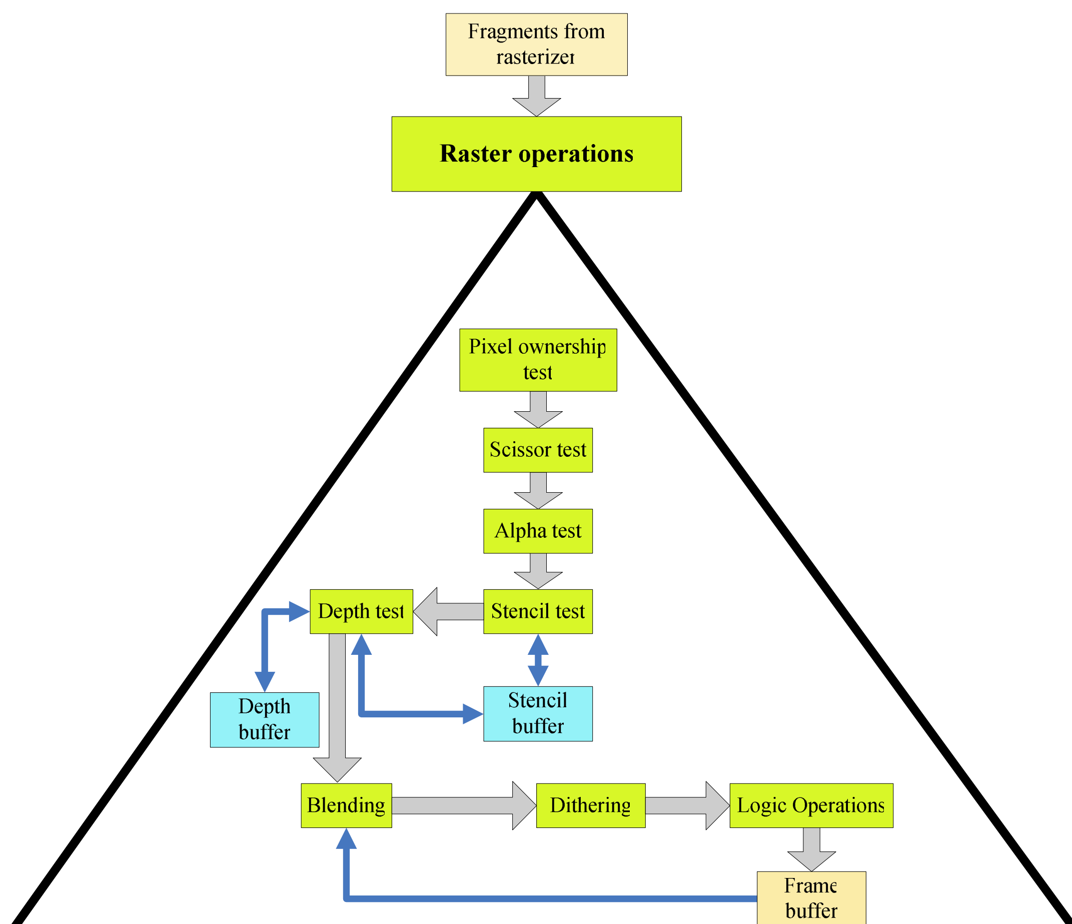


Figure 4.7 Raster operations in detail.

Raster operations stage is the last stage prior to updating the pixel value in the frame buffer. As depicted in (Luebke & Humphreys, 2007), many rasterization algorithms have been developed for this stage. All of these algorithms utilize one common observation that is each pixel can be processed independently from the others in parallel. This observation has resulted in development of massively parallel pipeline architectures in GPUs. Pixel ownership identifies whether the pixel is obscured by an overlapping window. Scissor test clips the fragments defined by the application. Alpha test discards the fragment based on its alpha value. Stencil test discards the fragment based on the comparison between the value in the stencil buffer and the reference value. Stencil buffer is composed of non-displayable bit planes that provides stencil value for every pixel. Stencil test provides extra rendering control by logical operations. Depth test discards the fragment by comparing its depth value with the corresponding depth value in the depth buffer. The depth buffer stores floating point depth values for every pixel that will be rendered. Stencil test together with depth test is used in many fundemental computer graphics techniques such as shadowing and reflections. For details in stenciling refer to (Kilgard, 1999). Blending combines the final colour of the fragment with corresponding pixel value.

Dithering is the means of noise addition to the signal to reduce the quantization errors that occurs due to the analog digital conversion of continuous data, as the resultant digitized data is just the representation with limited bits of the analog data. Similarly in computer graphics, dithering is a technique to create an effect of color depth more than actual limited colors due to the colors represented with limited number of bits i.e. 2 bits, 4 bits. The detailed examples can be found at (Wikipedia, 2010r).

At the end of the rasterizer stage, application defined logical operations are performend, and according to the cumulative result of the rasterizer stage, a write to the frame buffer is performed.

In years, GPUs evolved from fixed function graphics pipeline into fully programmable computational units. Figure 4.8 shows the programmable graphics pipeline.

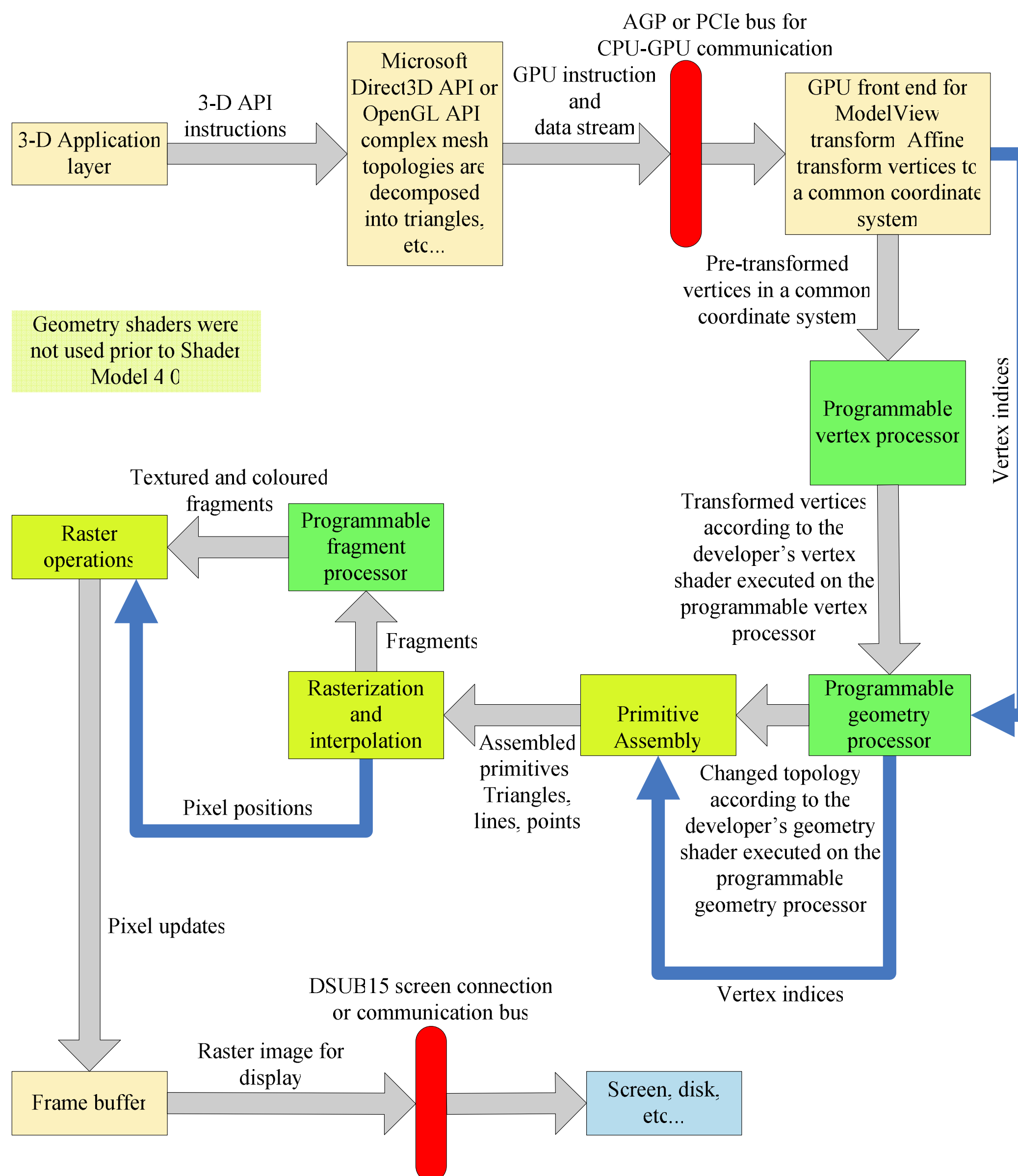


Figure 4.8 Programmable graphics pipeline.

In the programmable graphics pipeline, vertex processor, geometry processor and fragment processor are fully programmable by the application developer. The first task in the vertex processor is to load vertex specific data such as position, texture coordinates, color, and etc... to the vertex processor. Then, the next instruction in the vertex shader is fetched continuously until the vertex shader terminates. There are three types of registers that the vertex processor uses. The vertex attribute registers

contain position, normal and colour vector values which are read only and defined by the application. The temporary registers are for intermediate computation and they can be read from or written to. The write only output registers are used for output results for the transformed vertices and written by the vertex processor. This data is either sent to the geometry processor or to the rasterizer along the pipeline. The fragment processor perform texturing tasks in addition to the ability of performing the math operations that the vertex processor has. The fragment processor can access to a texture image by using texture coordinates and return a filtered sample of a texture image. The fragment shader has instructions to fetch textures. The key point for performance at this point is to use the lowest machine precision that is adequate for the application, because fragment shaders are executed until the shaders terminate for each fragment received. The read only input registers of the fragment processors contain the interpolated per fragment parameters derived from the per vertex parameters of the fragments primitive as depicted in (Fernando & Kilgard, 2003, p. 20). The temporary registers can be read from and written to for intermadiate computations. The resultant color and depth value for each fragment are written to the write only output registers of the fragment processor.

For further details, the researcher may refer to (Fernando & Kilgard, 2003, chap. 1), (Möller, & et al., 2008, chap. 2, chap. 3) and (Kirk & Hwu, 2010, chap. 1, chap. 2).

4.4 Unified Shader Architecture

The evolution of GPUs from a fixed pipeline to a programmable pipeline is an important technical process on its own. But another important technical development is “the unified shader model”. This model is also known as “Shader Model 4.0”. The first hardware examples for this architecture were ATI Xenos chip for Xbox 360 and NVIDIA 8800 chip for PCs. The unified shader architecture of NVIDIA 8800 is shown in figure 4.9.

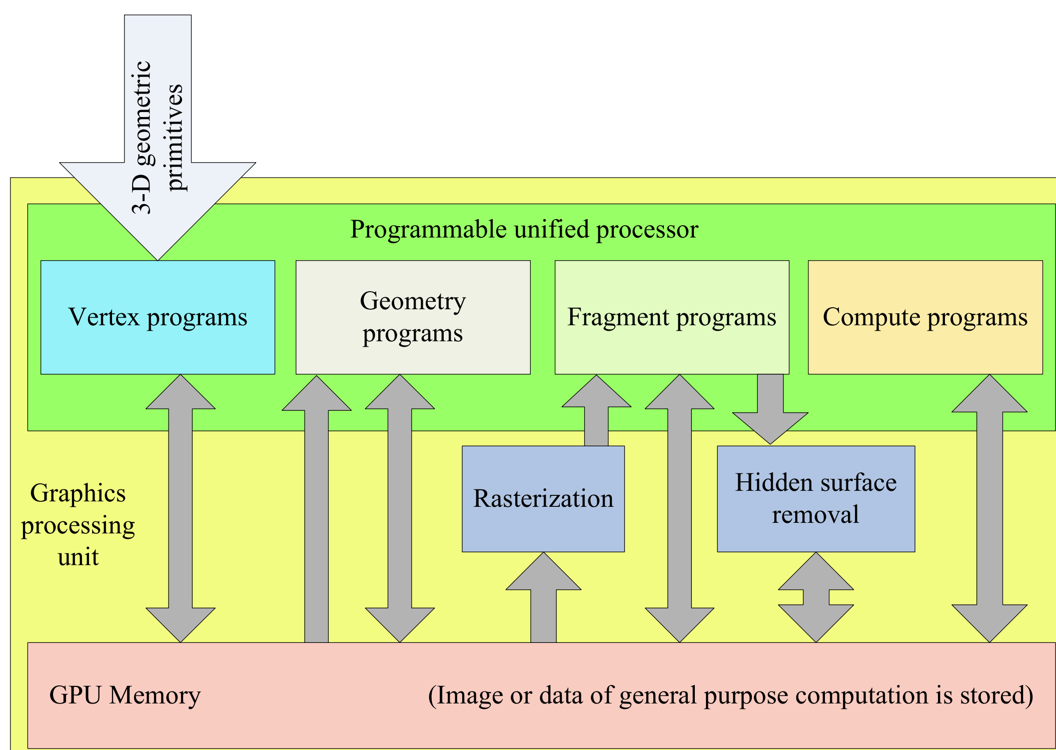


Figure 4.9 One of the first unified shader architectures belonging to NVIDIA 8800 (Inspired from Luebke & Humphreys, 2007, p. 4).

Geometry shader was not part of the hardware accelerated graphics pipeline prior to Shader Model 4.0 as depicted by (Möller, & et al., 2008, p. 41). Instead of seperate custom processors for vertex, geometry and pixel shaders, a unified shader architecture provides one large grid of data-parallel floating point processors general enough to run all these shader workloads (Luebke & Humphreys, 2007). This model uses almost the same instruction set for developing vertex, geometry or fragment shaders. During the task processing vertices, triangles and pixels pass through a set of programmable processors. The architecture that uses unified shader model is named as “the unified shader architecture”. This architecture is more flexible than the previous ones, because during the runtime of the application, the need for different types of shader processors continuously varies. For example, at one time the application may need vertex processors’ computation power more than that of geometry or fragment processors for generating a detailed scene with millions of vertices. In that case, geometry processors and fragment processors can be used as vertex processors. Otherwise, they would wait idle for the vertex processors to complete the task resulting in delayed task completition. Reversely, the application

may need a topology processing or pixel processing power for lighting or image processing more than others. Again for that case, the idle shader processors can be used for the geometry or fragment shaders respectively. As a result, in this architecture a necessary amount of processors in the processor pool can be assigned to the appropriate shader to balance the load. For further details, the researcher may refer to (Kirk & Hwu, 2010, chap. 2).

4.5 The Need for High Level Programming Languages for Computer Graphics–Cg HLSL and GLSL

As a result of evolution in the programmability of GPU hardware, developers felt a need for a programming language that will increase efficiency in development. As in the case of history of CPU development, the assembly language was the initial choice. Although assembly language enabled the programmers to use the GPU and its registers as they wanted, the code development process became error prone especially for long codes. The assembly code was not portable across different GPU platforms and the learning curve of the several GPU assembly languages slowed down the code development process severely.

The next step in developing codes for programmable graphics pipelines was the high level languages that were portable, easy to learn and less error prone. Additionally, these languages enabled the developer to focus on the problem at hand not on the hardware layer. Today three high level programming languages for programming the graphics pipeline for graphics dominated the world. These are Cg – C for Graphics - a cross platform language which can be executed with Microsoft Direct3D or OpenGL, HLSL – High Level Shading Language - from Microsoft which needs Microsoft DirectX and hence Microsoft Windows to execute and GLSL – OpenGL Shading Language - from OpenGL which needs OpenGL to execute. All of them are C like language with some restrictions and some semantic differences that allow the developer to program the vertex processors and fragment processors in the programmable graphics pipeline of the GPU. Cg and HLSL are nearly similar in programming perspectives as NVIDIA and Microsoft worked together during the

language development phase for common standards. For programming details in Cg, HLSL and GLSL the researcher should refer to (Fernando & Kilgard, 2003) for Cg, (Engel, 2004a), (Engel, 2004b) for HLSL and (Rost & Kane, 2010) for GLSL respectively. The researcher interested in development in assembly language for GPUs should refer to (Leiterman, 2004).

In the implementation case, Cg code cannot be used directly by a GPU. The code should be converted to the target machine code. The Cg compiler compiles the code that can be accepted by either Microsoft Direct3D or OpenGL API based on the choice of the developer. The API translation of the code is passed to the GPU via Microsoft Direct3D or OpenGL commands. Finally, Microsoft Direct3D or OpenGL driver produces the machine code that is accepted by the target GPU. These procedures are handled by the real time graphics engine Ogre3D that is used throughout the thesis work.

During the thesis work, Cg was mainly used for programming vertex and fragment processors for bump mapping with parallax offset. This texturing method and its modified schemes were used to texture wireframe models in the virtual environment. HLSL was used for a special lighting effect namely for light shafts implementation. Implementation of these graphics processing tasks in GPU released CPU for handling simulation logic, physical rendering and collision detection tasks.

Prior to implementation in the actual simulation, theoretical and practical study period for understanding general programming aspects of Cg and HLSL had been evaluated. The results are given in figures 9.23 – 9.25.

4.6 NVIDIA Compute Unified Device Architecture - CUDA and General Purpose Computing

As the evolution of GPUs continued towards unified processor architectures, they became more like parallel computation units. Therefore, researchers wanted to exploit the usage of these systems in performance sensitive scientific and engineering

applications. Initially, the graphics APIs were just capable of executing graphics related calls. The problem at hand should be cast in terms of these calls within a pixel shader. The input data for the computation was being stored as a texture and sent to the GPU by submitting triangles. Furthermore, the restrictive memory interface of the GPUs and limited read and write abilities made the storage of the computation results in the frame buffer much more difficult. The attempts for overcoming these technical difficulties resulted in *General-purpose computing on graphics processing units* (GPGPU). For details, the researcher should refer to (GPGPU.org, 2010). In spite of its technical problems, the researchers in several institutions developed successful applications. Stanford University's folding@home project relies on GPU based computations to study protein folding by using the spare cycles of the computers of the users that donate to the project. The researchers at University of North Carolina and Microsoft won a competition on sorting a database.

By the time, NVIDIA was designing a floating point and integer processor that could run tasks in parallel for Microsoft DirectX 10. The shader processors became fully programmable with increased instruction memory, cache and sequencing logic where each shader processor share its instruction cache and sequencing logic with others. In addition to this hardware, memory load and store instructions were added with the support of random byte addressing for compiled C programs. As a result, for non graphics applications, this GPU architecture was a generic programming model with a hierarchy of parallel thread, barrier synchronization and atomic operations to dispatch and manage parallel work load. At this point the development of CUDA C compiler (a support for C++ exist in newer versions of CUDA API for object oriented programming), libraries and runtime enabled the program developers to use this new hardware architecture. The main point was that, the application developers were no longer needed to use graphics API such as Microsoft Direct3D or OpenGL to access the GPU hardware for general purpose programming.

Similarly, ATI developed ATI Stream for general purpose computation on its GPUs. An open source API named as OpenCL exists for the same general computing purposes on both NVIDIA and ATI GPUs. But OpenCL is still under development.

CUDA enables the GPU to be accessed like a general purposes CPU. The application developer can access the virtual instruction set and memory units of parallel computation elements in the architecture. In a typical heterogeneous computing environment where CPU and GPU exists simultaneously; CPU is typically called as *a host* and GPU is called as *a device*. The function that will be executed in parallel is called *kernel*. The threads are contained in thread blocks which can be one, two or three dimensional. In a same way, thread blocks form a grid. This hierarchy is shown in figure 4.10 and figure 4.11 inspired from (NVIDIA, 2009a, p. 10, p. 11). The host executes the sequential code; on the other hand, the device executes the parallel portion of the computation. The device code is compiled by NVIDIA C Compiler *nvcc* which can be integrated into several development environments such as Microsoft Visual Studio 2005, therefore both the host code (Microsoft Visual C++ compiler for this particular case) and the device code can be compiled in a batch. The researcher who isn't convenient with multithreading and multiprocessing concepts should refer to (Deitel, H., Deitel, P., & Choffnes, 2004).

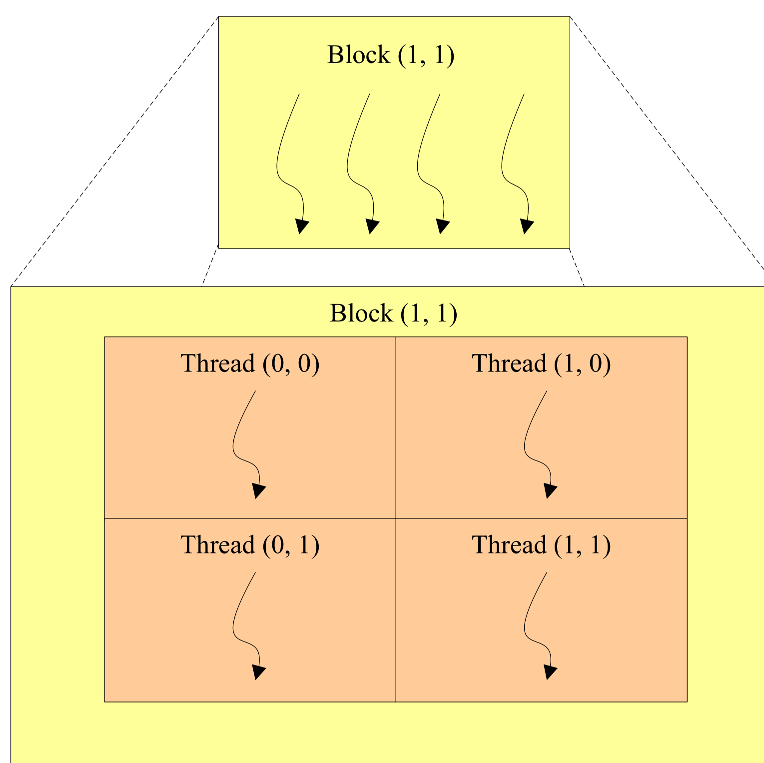


Figure 4.10 Threads inside of a thread block.

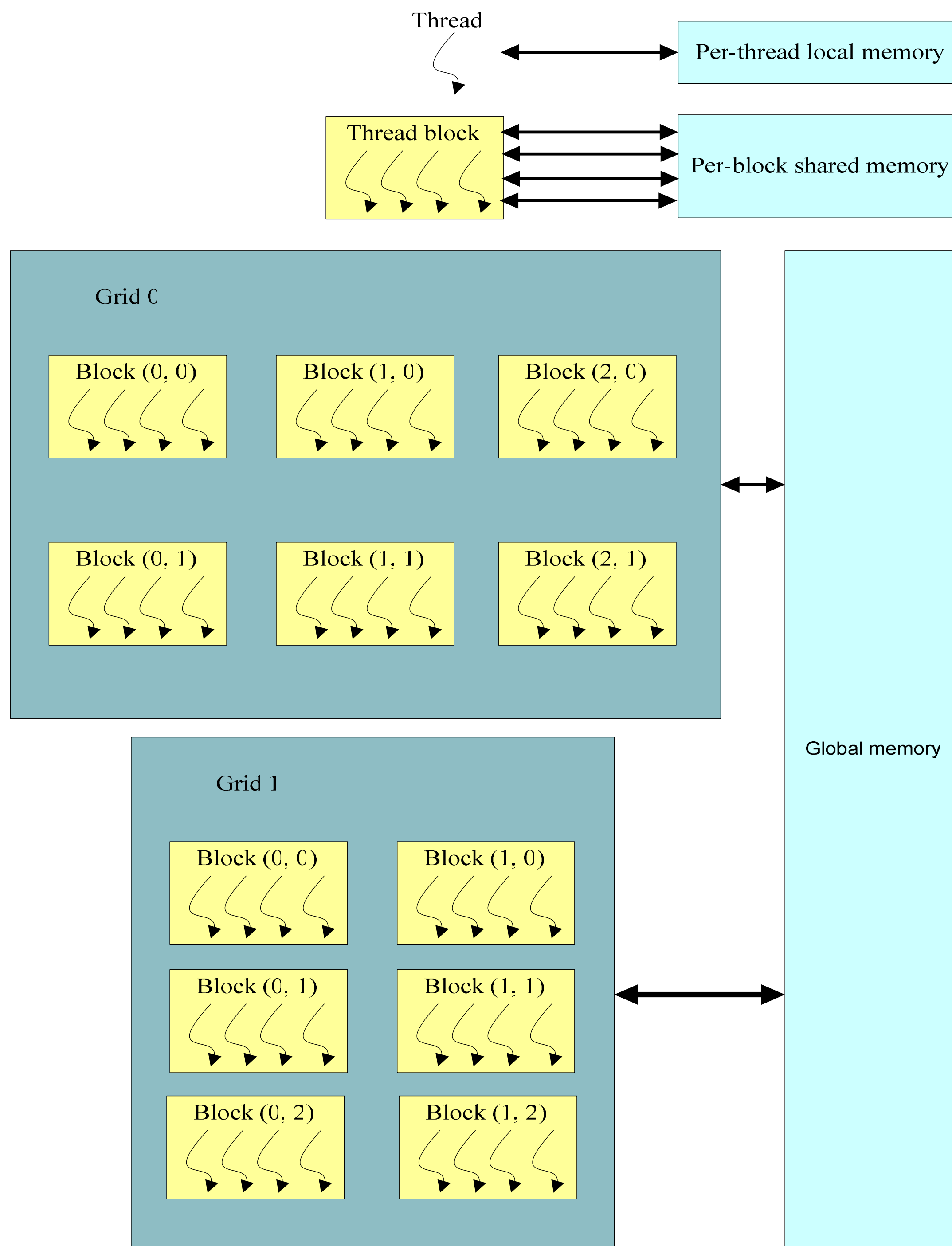
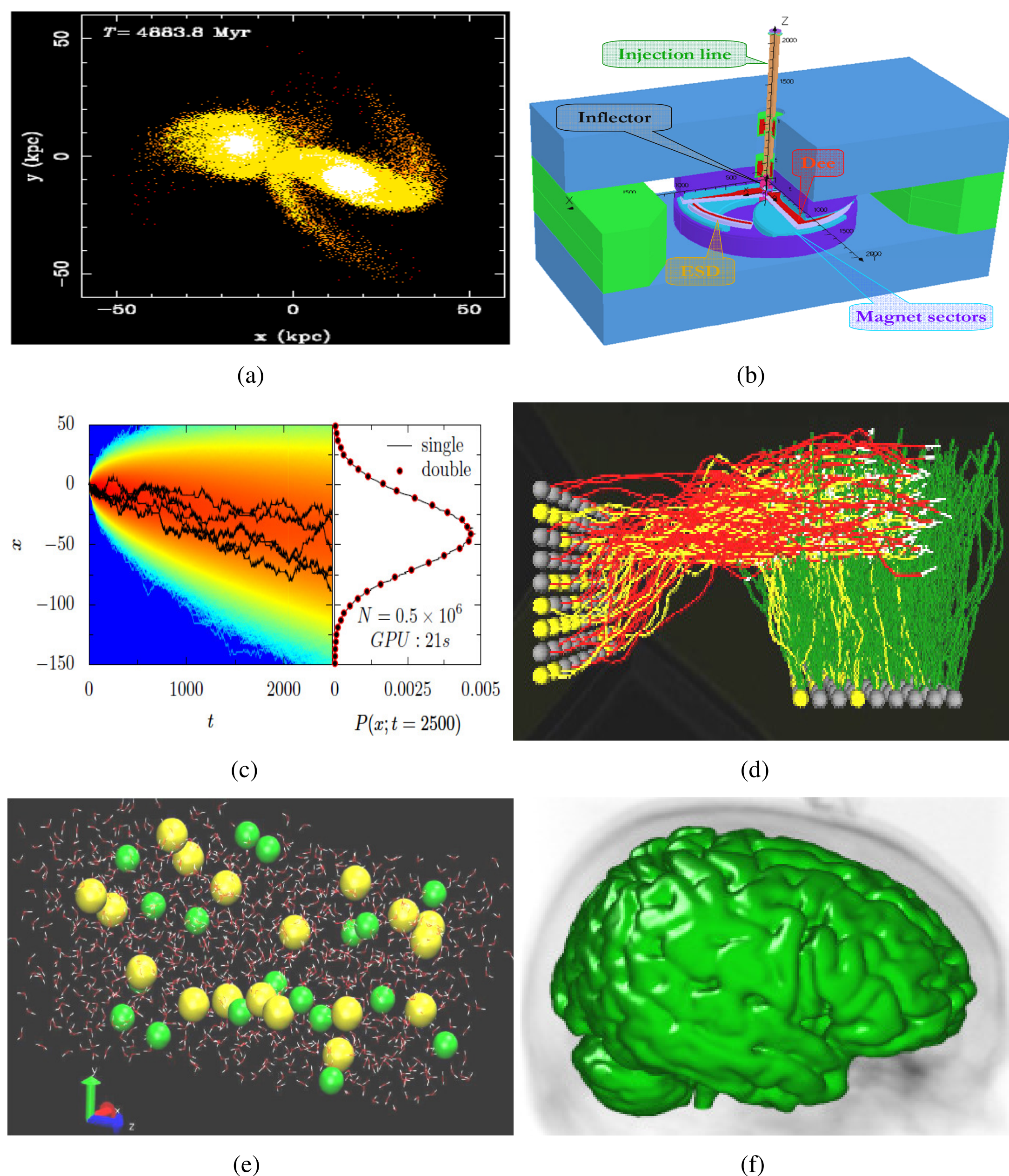


Figure 4.11 Memory hierarchy of NVIDIA CUDA.

The following characteristics are valid for the time this thesis was being written. Any frame memory area can be read from or written to. Threads can share a fast shared memory region and high bandwidth communication is possible. Reads and writes by GPU is faster. Integer and bitwise operations, integer texture looks up are faster. The language for the device code is in fact ANSI C with no recursion and no



(a) (b) (c) (d) (e) (f)

Figure 4.12 Applications using NVIDIA CUDA. (a) Simulation of approaching two galaxies with 260k particles for an initial interaction. (b) Calculation of beam dynamics of a cyclotron. (c) Numerical solution of stochastic differential equation for modeling the noisy dynamics of phase in Josephson junction. (d) Simulating brain vision and olfactory circuit. (e) Simulation of NaI solution molecular dynamics. (f) Level set segmentation with MRI.

function pointers but with addition of special keywords. The support for C++ is available in Fermi GPUs. Thousands of threads can run simultaneously in total but they should be in groups of at least 32 for optimized performance. Texture rendering is not supported. Double precision data is supported for NVIDIA GTX 260 and

newer GPUs. The bandwidth and latency between GPU and CPU is a disadvantage. CUDA is available only for NVIDIA GPUs such as NVIDIA GeForce 8 and up, Quadro and Tesla; on the other hand OpenCL can be used by many GPUs from different vendors.

A simple example of matrix multiplication will give an insight of usefulness of GPUs. Consider two matrices A_{MxN} and B_{NxP}. In a sequential matrix multiplication code executed on a CPU, the complexity of the algorithm will be $O(MNP)$, whereas if that multiplication is executed parallel on a GPU, each row-column multiplications will be completed at once in parallel. There are many more uses of computing based on GPUs in science and engineering. In figure 4.12, some examples from several research communities and academia are given. For the practices done regarding NVIDIA CUDA during the thesis period, refer to figure 9.26.

Considering figure 4.12, the top left research is from (Groen, Harfst, & Zwart, 2009); top right research is from (Perepelkin, Smirnov, & Vorozhtsov, 2009) and represents the calculation of beam dynamics of a cyclotron on two different platforms. With 1000000 particles, the computations take 2 days 4 hours and 25 minutes on a 2.5GHz CPU and the same computations take just 34 minutes and 29 seconds on NVIDIA Tesla C1060 GPU. The middle left research represents a numerical calculation of stochastic differential equation and a 675 times faster calculation with NVIDIA TESLA 1060C than a CPU is indicated in (Januszewski, Kostur, 2009). The middle right research represents brain circuitry, vision and olfactory sensory computing with GPU in which the computations are executed with 130 times faster than CPU taken form (Kirk, n.d.). The bottom left research is a molecular dynamics research with NaI solution from (Davis, Ozsoy, Patel, & Taufer, 2009) indicating 7 times speed up with GPU computing over CPU computing. The bottom right research from (Roberts, Packer, Sousa, & Mitchell, 2010) represents the level set segmentation of 256^3 MRI data with GPU. GPU provides 14 times speed up with GPU computing over CPU computing.

This evolution of GPUs will continue in the future, perhaps faster than the evolution of CPUs. GPUs have many more transistors than CPUs dedicated for computations, more streaming bandwidth for data transfer and many more processing units than CPUs. Figure 4.13 taken from (Kirk & Hwu, 2010, chap. 1, p. 3) depicts a comparison of floating point operations per second between CPUs and GPUs from year 2001 to 2009.

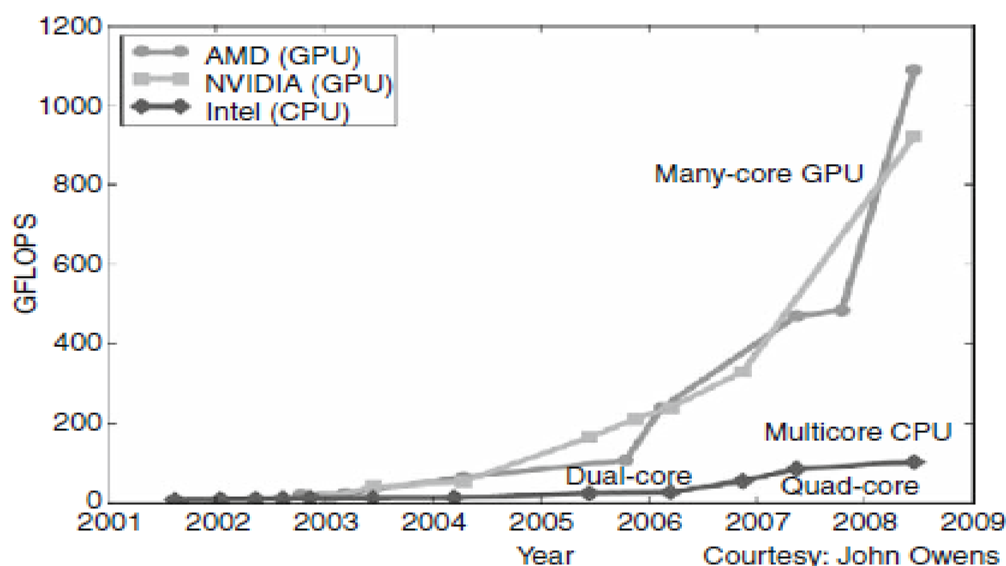


Figure 4.13 Floating point operations per second for INTEL CPU, NVIDIA GPU and ATI GPU. (Kirk & Hwu, 2010, p.3).

Although a 64 bit INTEL Core i7 980X has a memory bandwidth of 25.6 GB/sec (INTEL, 2010), NVIDIA GTX 295 has 223.8 GB/sec (NVIDIA, 2010) with 896 bit memory interface width. Although today's modern CPUs such as INTEL Q9550 and Core i7 CPUs have 4 to 6 computation cores respectively, modern GPUs such as NVIDIA GeForce GTX 295 have 480 computation cores in their architecture. This is because, over the history of their development CPUs have evolved to decrease the latency between the memory unit and the CPU (the researcher should notice the connection with the von Neumann architecture mentioned previously) and for executing sequential programs faster, but on the other hand GPUs have evolved for massive vector and matrix operations done in parallel by thousands of threads which is impossible even for the most high end CPU today. This parallelism and data processing capacity make them very suitable not only for computation power

demanding visualization applications but also for scientific and general purpose computation.

The researcher should refer to (Möller, Haines, & Hoffman, 2008), (Fernando & Kilgard, 2003), (Rost, Kane, Ginsburg, Kessenich, Lichtenbelt, Malan, & Weiblen, 2010), (Kirk & Hwu, 2010), (Fernando, 2004), (Pharr & Fernando, 2005), (Nguyen, 2007) and (NVIDIA, 2009a) for further details in real time computer graphics, GPUs, available programming languages, general purpose programming and scientific computation on GPUs and NVIDIA CUDA.

CHAPTER FIVE

ESSENTIALS OF REAL TIME GRAPHICS RENDERING

One of the two important modules of 3-D immersive virtual environment is the graphics engine. Several visual rendering tasks such as lighting, shadowing, fogging, texturing of visual synthetic objects, image based effects such as billboarding, skyboxes, volume rendering, non-photo realistic rendering, etc… and affine transformations of visuals are accomplished via graphics engine. Additionally, graphics engines handle curve and surface rendering and processing tasks in 2-D and 3-D. Each graphics rendering engine uses a tree data structure to keep visuals and rendering functions in a logically and spatially consistent hierarchy. This is already mentioned in chapter three.

In this chapter, the fundamental rendering techniques implemented, and other graphics rendering techniques that are implemented targeting the graphics processing unit (NVIDIA GTX 295 GPU) with programmable graphics pipeline will be given. One of these techniques regarding lighting is named as “Light Shafts”. The vertex shaders and fragment shaders for this technique is implemented in Microsoft HLSL shading language. The other technique regarding the texturing of 3-D objects is named as “Bump Mapping with Parallax Offset”. The vertex shaders and fragment shaders for this texturing technique are implemented in NVIDIA Cg shading language. Several experimental code studies for lighting, transformations, animation, etc… with NVIDIA Cg can be found in figures 9.23 – 9.25. Therefore the usage of vertex processors and fragment processors of the programmable graphics pipeline of the GPU offloaded CPU from these computationally demanding tasks. As it will be seen in chapter six, CPU mainly deals with the physics simulation tasks throughout the thesis work. This chapter will present brief information on a well known “Gimbal Lock Problem”. Finally, a hand rigging and skinning accomplished in the thesis work for the interaction of the user with the visual object will be presented.

5.1 Transformations, Lines, Surfaces and Rendering Techniques in Computer Graphics

Transformations, lines and surfaces are the basic building blocks of computer graphics. The mathematical details of the subjects are partly given in chapter five. For more mathematical coverage regarding graphics can be found at (Möller, & et al., 2008). In the context of the thesis work, during the absence of the necessary laboratory equipment, a stand alone application was developed targeting the creation of several line and surface types such as Coons surface, BSpline surface, etc… and performing transformations on these primitives without any use of graphics library. The aim was to understand and implement the mathematics underneath. Furthermore, it is known that the implemented surfaces in this application have a well known usage in modeling soft tissues and objects. The researcher should refer to figure 9.16 for the implementation results.

Prior to developing the software, it would be wise to practice on fundamental graphics rendering techniques regarding lighting, texturing, environment mapping and occlusions. Maintaining a solid working background on these topics would save time when problems occur in the actual software. Besides we would have a chance to observe which techniques would be usable for us in the actual software development process. For practicing rendering techniques, OpenGL is used. A well coverage of topics can be found in (Möller, & et al., 2008) and (Wright, Lipchak, & Haemel, 2007). The researcher should refer to figure 9.17 for the implementation results.

5.2 Gimbal Lock Problem – Rotation via Euler Angles and Quaternions

In computer graphics, engineering and mathematics, rotations can be represented by three forms. Briefly these representations are matrix representation, Euler angles and quaternions. Each representation has its own advantages and disadvantages. The researcher can refer to (Möller, & et al., 2008), (Bergen, 2004) and (Dunn & Parberry, 2002) for detailed coverage. What we want to mention in this section is the well known gimbal lock problem that occur with rotations via Euler Angles.

The problem occurred in the software development period when a 3-D virtual object was tried to be oriented in the virtual world coordinate system by using the Euler angles acquired from the motion tracking device. The technical details of the motion tracking device can be found in chapter eight. Although a calibration procedure of the motion tracking device had been done, the rapid rotations and movements of the virtual object could not be avoided at certain orientations. The technical and mathematical description of the gimbal lock problem that occurred can be given as follows referencing from (Wikipedia, 2010t).

Gimbals are a ring like structures that are constrained to rotate only about one axis. They are places one in another to define rotations about multiple axes. For example, inertial navigation systems are common devices where gimbals are used. In these systems, while the inner gimbal is constrained to be fixed, the outer gimbal rotates about an axis. A set of three gimbals defining the orientation of the arrow is given in figure 5.1 (a) with no gimbal lock problem. For some coordinate systems, it seems suitable to assume that there exist gimbals coincident with the coordinate axes. Therefore using the Euler angles seems feasible both mathematically and programmatically. This assumption is valid if and only if the Euler angles are constrained to an interval.

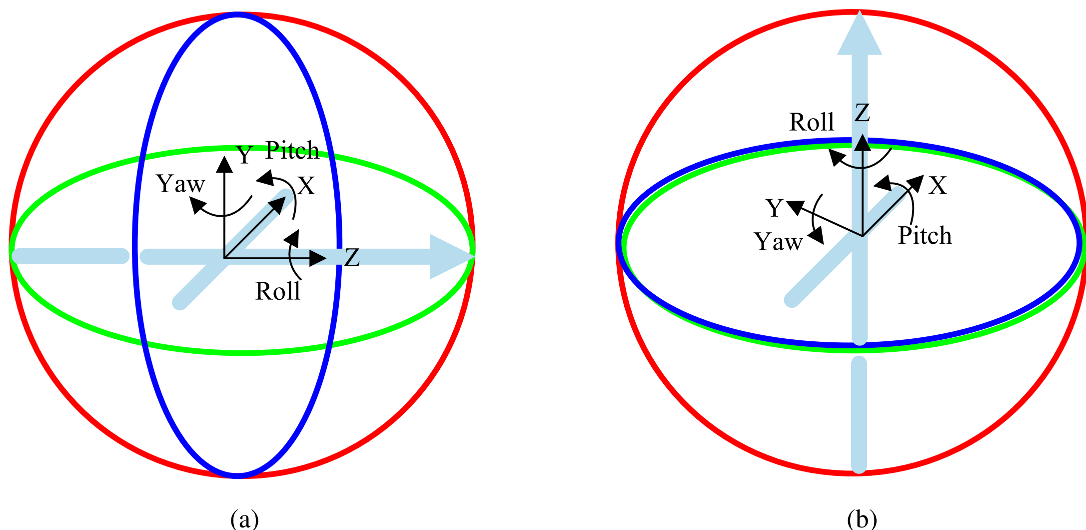


Figure 5.1 (a) Three independent gimbal set with no gimbal lock problem. (b) When the arrow pitched up 90 degrees, one degree of freedom is lost, yaw axis cannot be controlled. Yaw and roll axes are dependent resulting in a gimbal lock problem.

Mathematically, the reason is that there exists no covering map from Euler angles to rotations (topologically mapping from a torus to a 3 dimensional real projective space). Therefore, at some points the rank of the system decreases from 3 to 2. Hence, Euler angles cannot provide a unique representation at those points. The only possible solution is to use quaternion representation (topologically mapping from a sphere to 3 dimensional real projective space). Gimbal lock is exemplified in configuration of figure 5.2 (b).

Quaternions are 4-D vectors (w, x, y, z) that can be used to represent rotations if $w^2 + x^2 + y^2 + z^2 = 1$. (x, y, z) is a 3-D complex vector and represents an arbitrary axis of rotation. w is a real number that represents the angle of rotation. Therefore, in contrast to Euler angles which are made of three successive rotations, a quaternion represents a single rotation around an arbitrary axis. Hence a rotation θ around a normalized axis (x_0, y_0, z_0) is represented as follows in quaternion notation.

$$\left(\cos\left(\frac{\theta}{2}\right), x_0 \sin\left(\frac{\theta}{2}\right), y_0 \sin\left(\frac{\theta}{2}\right), z_0 \sin\left(\frac{\theta}{2}\right)\right) \tag{5.1}$$

Besides their use in rotations, quaternions are used for the interpolation between two orientations instead of Euler angles. During the thesis work, quaternions are used both for rotations and interpolations between two orientations.

5.3 Lighting and Implementation of Light Shafts

Lighting is an important factor in creating natural 3-D virtual environments. Point lights, directional lights and spotlights are the three important lighting types that can be used depending on the needs. But none of these simulate how the light scatters according to the environment it passes through. On the other hand, in a real environment, light scatters and forms shafts while passing through an environment with some particles. This effect in real world is shown in figure 5.2 which is taken from (Smith, 2004).

Figure 5.2 Light shafts in a real scene resulting from sun rays partly occluded with clouds (Smith, 2004).

In our 3-D virtual environment, the light rays from medical operation light were modeled as if the rays were forming shafts because of the scattering. This light shaft implementation was also used to highlight the anatomical parts that were touched by the user hand in order to focus attention. Our mathematical, algorithmic reference for the implementation was (Mitchell, 2004). The implementation results can be seen in figures 9.4(c) – (d), 9.8 (a), 9.9 (a) and 9.11 (a)-(b).

5.4 Texturing and Implementation of Bump Mapping with Parallax Offset

Bump mapping is another lighting technique that combines per-fragment lighting with surface normal perturbations supplied by a texture to simulate lighting interactions on bumpy surface without excessive tessellation as indicated by (Fernando & Kilgard, 2003). In 3-D real-time rendering applications, parallax offset provides a depth feel hence more realism, i.e. walls or floors seem as if there were gaps between the bricks. Simply speaking, the complexity of the scene is increased without adding new polygons.

In order to implement this technique, the texture coordinate of a point on the polygon should be displaced as a function of the view angle relative to the surface normal and of the height map value at that point of the polygon. The algorithmic details and generation of the height map can be found at (Fernando, & Kilgard, 2003) and a forum topic can be found at (Guest, 2010).

The implementation results of the technique are given at figure 9.4 and 9.5 (a).

5.5 Hand Rigging and Skinning

Hand is an important part of a body for manipulating objects, touching, mimicking, performing everyday tasks and etc … In the developed application; a 3-D hand was used as an object manipulator of the user in the virtual environment. The global translation and rotation of the 3-D hand was being performed by the second sensor of the motion tracker device that was attached to the data glove. The first sensor of the motion tracker device mounted on the HMD was being used for tracking the user head translation and rotation for walking in the virtual environment. In order to increase the realism and make the user feel as if he / she were using his / her own hand, local finger movements of the virtual hand were also modeled regarding the anatomical constraints of the user hand that is given in (Rhee, Neumann, & Lewis, 2006). The rigging and skinning of the hand was done using 3DS MAX 2008. Then the completed hand model was exported to our software for controlling the bones in order to perform various hand gestures.

In the rigging process, the hand was assumed to have 14 degrees of freedom i.e. total number of joints. Two joints were connected to each other by bones. The bone structure was setup so that it was coincident with the 3-D hand mesh. In the skinning process, each bone was assigned a cylindrical like volumetric region in which it would be able to control the vertices that were in the volumetric region. The controllability of the vertex was defined by a coefficient value between [0,1]. The vertices in the cylindrical like volumetric region had the coefficient of 1; and towards the boundaries of the region the coefficient value decreases towards zero. This means

that, the bone would be able to affect less to the vertices on the boundary of its control region. The results of rigging and skinning are given in figure 5.3.

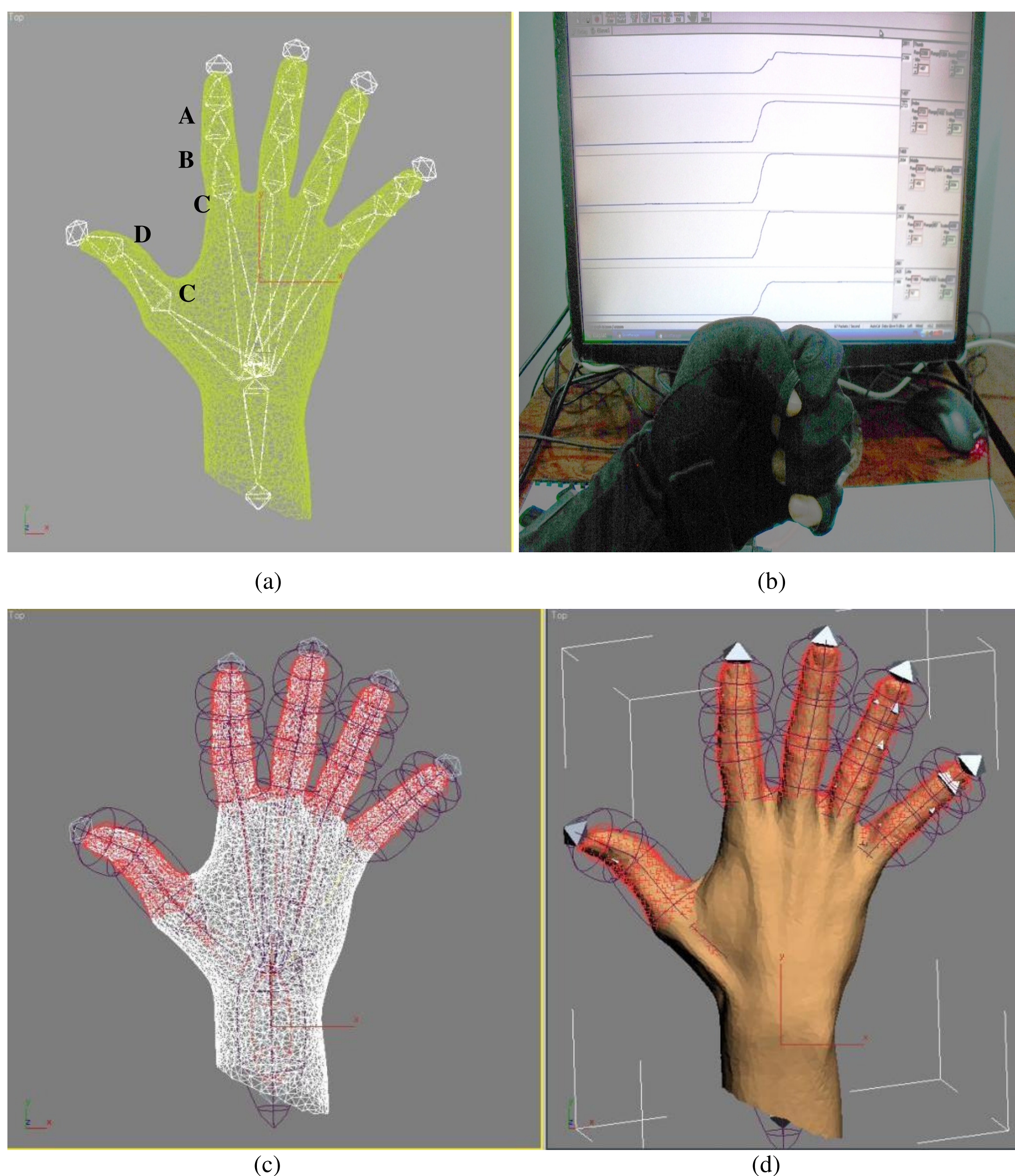


Figure 5.3 (a) The anatomical names of the hand joints, A. DIP joint, B. PIP joint, C. MCP joint, D. IP joint. Each bone is connected to joints at its ends. (b) Five bend sensor data being acquired from left hand for fist gesture. Right hand is captured for bone mapping in a similar way. (c) - (d) Each bone has an effective cylindrical like volumetric region in which it can control the corresponding vertices. The control coefficient of each bone in its region varies between 0 and 1. 1 corresponds to full control as indicated by red vertices. Towards the boundaries of the volumes the control coefficient decreases towards 0. Hence the vertices at the boundaries of the volumetric region are effected less by the rotation of the bones. Those boundary vertices are shown in white. The vertices having intermediate coefficient values have colors of yellow, green, etc… (Deformed in figure 9.4).

The following hand joint heuristics are used regarding (Cerveri, Momi, Lopomo, Baud-Bovy, Barros, & Ferrigno, 2007), where θ_{MCP}, θ_{PIP} ,θ_{IP} and θ_{DIP} are the MCP, PIP, IP and DIP joint angles respectively as shown in figure 5.3 (a).

$$
\begin{aligned}
&0° \le \theta_{MCP} \le 90° \\
&0° \le \theta_{PIP} \le 90° \\
&0° \le \theta_{IP} \le 90° \\
&\theta_{DIP} = \frac{2}{3}\theta_{PIP}
\end{aligned}
\tag{5.2}
$$

For each finger, the only data acquired was from the bend sensor for each finger found in the data glove. The incoming data for each finger was real valued between [0, 1] where 0 corresponds to no bending and 1 corresponds to full bending of the corresponding finger. This value can be thought as a linear combination of the angles of the joints such that the nth sample value $x[n]$ received from the bend sensor can be expressed as,

$$
\begin{aligned}
x[n] &= c_1 x_{MCP}[n] + c_2 x_{PIP}[n] + c_3 x_{DIP}[n] \\
&= c_1 x_{MCP}[n] + c_2 x_{PIP}[n] + c_2 \frac{2}{3} x_{PIP}[n] \\
&= c_1 x_{MCP}[n] + c_4 x_{PIP}[n] \quad ; \quad c_4 = c_2 + c_2 \frac{2}{3}
\end{aligned}
\tag{5.3}
$$

Assuming heuristically,

$$c_1 x_{MCP}[n] = 2(c_4 x_{PIP}[n]) \tag{5.4}$$

Using (5.4) with (5.3), $c_1 x_{MCP}[n]$ was found. By assumption c_1, c_2, c_3 are all equal to 1.0. $c_1 x_{MCP}[n]$ was linearly mapped to the interval [0, 90] degrees where $x_{MCP}[n] = 1.0 \rightarrow \theta_{MCP} = 90°, x_{MCP}[n] = 0.0 \rightarrow \theta_{MCP} = 0°$. Similarly the other angles are found using (5.4), (5.3) and (5.2). Observe that the calculations are valid up to a constant. The implementation results are given in figures 9.4 (c) – (d).

CHAPTER SIX

ESSENTIALS OF REAL TIME PHYSICS RENDERING AND SIMULATION OF DYNAMICAL SYSTEMS

Real time physics rendering can be divided into three main topics as linear algebra, numerical analysis and topology. Hence, at this point, the fundamental mathematical terms such as *vector space*, *linear combination*, *span*, *linear transformations* and etc… from linear system theory course are assumed to be understood. This chapter will introduce common topological definitions necessary for background. Then the chapter will go on with collision models, collision detection methods, mass-spring systems and constraint solutions. In a typical application, these concepts are implemented as software modules of a physics engine as shown in figure 6.1. For the detailed explanations related to topology and collision detection given in this chapter, the researcher should refer to (Bergen, 2004), (Ericson, 2005) and (Möller, & et al., 2008). For an additional mathematical resource, the researcher should refer to (Strang, 1986), (Rugh, 1996) and (Rogers & Adams, 1990).

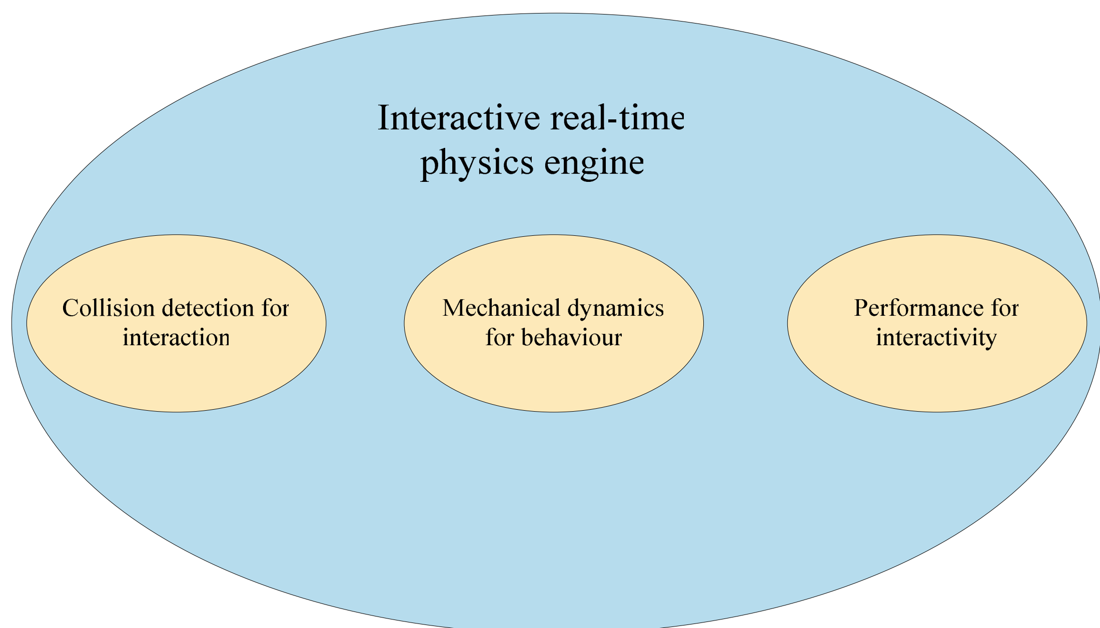


Figure 6.1 Components of an interactive real-time physics engine.

6.1 Topological Definitions

6.1.1 Affine Spaces

An affine space is defined by a set of points, an associated vector space and two operations as the addition of a point and a vector, and the subtraction of two points. The addition of a point and a vector results a point according to the following rules: (a) The addition of a point and a null vector 0, (b) The addition of a point p and vectors v and w using commutativity respectively as given below;

$$\begin{aligned} &p + 0 = p \\ &(p + v) + w = p + (v + w) \end{aligned} \tag{6.1}$$

Subtraction of two points yields a vector according to the following rule, where p and q are points.

$$p + (q - p) = q \tag{6.2}$$

A point x can be written as *the affine combination* of points $p_0,..., p_n$ can be defined as,

$$x = \sum_{i=0}^{n} \alpha_i p_i \ , \ \ni \ \sum_{i=0}^{n} \alpha_i = 1 \quad (\alpha_i \text{ are scalars.}) \tag{6.3}$$

Eliminating α_0 from the equation and arranging the equation above yields a point p as seen below.

$$\begin{aligned} &p = p_0 + \alpha_1 (p_1 - p_0) + ... + \alpha_n (p_n - p_0) = p_0 + \sum_{i=1}^{n} \alpha_i (p_i - p_0) \ , \\ &\ni \sum_{i=0}^{n} \alpha_i = 1 \end{aligned} \tag{6.4}$$

The set of affine combinations of points A is called as *the affine hull* and denoted as $aff(A)$. That is,

$$aff(A) = \left\{ \sum_{i=0}^{n} \alpha_i p_i \middle| p_i \in A, \sum_{i=0}^{n} \alpha_i = 1 \right\} \quad (6.5)$$

A set of points that is closed under affine transformations is called as an affine set. Points, lines and planes are examples of affine sets. A set of points $\{p_0,...,p_n\}$ is called *affinely independent* if the set $\{p_1 - p_0,...,p_n - p_0\}$ is linearly independent. *The dimension of the affine space* is that of associated vector space. The important result is, the number of points in an affinely independent set is the dimension of its affine hull plus one. This can be generalized to N dimensional spaces. A coordinate system is a tuple of a point and a basis. Consider the point c as origin and basis $\{b_1,...,b_n\}$, then the point p can be expressed uniquely by vector V that is

$V = \{\alpha_1,...,\alpha_n\} \in \Re^n$, where α_i are coordinates of point p

and

$$p = c + \sum_{i=0}^{n} \alpha_i b_i \quad \text{, that is the affine combination of origin and basis } b_i \quad (6.6)$$

Therefore, the coordinate system defines an affine space in which each point is defined uniquely by a vector of coordinates.

In many cases while developing a 3-D application, multiple coordinate frames are used. The same point can be defined relative to different coordinate frames, or the coordinate system can be defined relative to a parent coordinate system. *The affine transformations* are used for transforming coordinates from one coordinate frame to another. The affine transformation T that maps coordinates to coordinates can be stated as follows,

$$T(\alpha p + \beta q) = \alpha T(p) + \beta T(q) \quad , \alpha, \beta \text{ are scalars, } \alpha + \beta = 1 \quad (6.7)$$

In a same way, an affine transformation is determined by the images of the basis and the origin of the given coordinate system. Considering B as the image of the basis, and c be the image of the origin, the corresponding affine transformation T is,

$$T(x) = Bx + c \qquad (6.8)$$

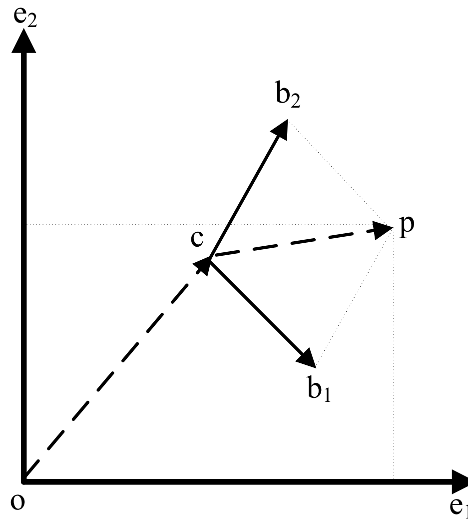


Figure 6.2 Affine transformation in $\Re^2$ (Bergen, 2004, p. 16).

Considering figure 6.2, coordinates of point p is relative to the system $(c, \{b_1, b_2\})$. Its coordinates relative to $(o, \{e_1, e_2\})$ is $Bp + c$ where basis vectors are $B = \{b_1, b_2\}$. B and c are defined relative to $(o, \{e_1, e_2\})$. The primal ancestor of all coordinate systems is named as *the world coordinate system*. In figure 6.2, the world coordinate system has origin o and basis vectors $E = \{e_1, e_2\}$. The descendent coordinate systems are named as *the local coordinate system*.

The function composition operator for the set of affine transformations from $\Re^n$ to $\Re^n$ can be defined as follows.

$$T_2 \circ T_1(x) = B_2(B_1 x + c_1) + c_2 = B_2 B_1 x + B_2 c_1 + c_2 \qquad (6.9)$$

The inverse transformation can be defined as follows.

$$T^{-1}(x) = B^{-1}(x - c) \tag{6.10}$$

The identity transformation is defined as I. The composition of affine transformations has many practical consequences. For example, in the virtual environment created during the thesis work, a camera translate node was defined relative to the world coordinate system. The yaw node was relative to the translate node, the pitch node was relative to yaw node, the roll node was relative to the pitch node and finally the pitch node held the camera from which the user viewed the virtual environment. The connections in the graph are presented in chapter 9. These relative definitions were established to overcome the gimbal lock problem due the Euler Angels acquired from the motion tracker. Each coordinate system was constrained to have only 1 degree of freedom. The other solution for gimbal lock problem was to use quaternions. In the node connection configuration described, T_1 may represent roll node coordinate system relative to pitch node coordinate system. T_2 may represent pitch node coordinate system relative yaw node coordinate system, T_3 may represent yaw node coordinate system relative to translate node coordinate system and T_4 may represent translate node coordinate system relative to the world coordinate system. Thus, $T_4 \circ T_3 \circ T_2 \circ T_1$ represents roll node coordinate frame relative to the world coordinate frame.

6.1.2 Euclidean Spaces

Euclidean space is an affine space with length and distance. At this point, it is assumed that the researcher is familiar with terms length, distance, orthogonality, orthonormality and normalization. For definitions of the terms, the researcher may refer to (Bergen, 2004, chap. 2), (Möller, & et al., 2008, chap. 4) and (Rugh, 1996). The definitions of terms normal and orientation for a hyperplane will be given. These terms have importance in many spatial transformations, lighting, shadowing calculations, culling and several numerical applications.

For $n \in \Re^m \setminus \{0\} \wedge \delta \in \Re$, the hyperplane $H(n,\delta)$ is a set of points defined by

$$H(n,\delta) = \{x \in \Re^m \mid n.x + \delta = 0\} \;,\; n \text{ is the normal, } \delta \text{ is the offset from origin} \tag{6.11}$$

The normal of the hyperplane is normalized prior to transformations. For example a distance of a point p to the hyperplane is $\|n.p + \delta\|$ if the norm of the normal is 1. Additionally, having normalized normal in lighting calculations prevents undesired distortions.

The orientation of a hyperplane is defined by the direction of the normal. The simplest application of orientation is the face culling that is to render or not to render the face viewed by the camera depending on whether back face or front face is selected for culling. That means, although $H(n,\delta)$ and $H(-n,-\delta)$ refer to the same point set, they are considered as different planes by the render system.

Simple intersection tests can be performed whether a point is on the positive, negative closed half space of a hyperplane or on the hyperplane. The positive and negative closed half spaces are defined as follows respectively.

$$\begin{aligned} H^+(n,\delta) &= \{x \in \Re^n \mid n.x + \delta \geq 0\} \\ H^-(n,\delta) &= \{x \in \Re^n \mid n.x + \delta \leq 0\} \end{aligned} \tag{6.12}$$

Referring to (Bergen, 2004, p. 21), the following definitions are valid only for 3-D Euclidean space. A coordinate system is right handed if the matrix B made up of basis vectors has the following property,

$$B = \begin{bmatrix} | & | & | \\ b_1 & b_2 & b_3 \\ | & | & | \end{bmatrix} \quad , \quad \det(B) > 0 \tag{6.13}$$

The cross product definition is important for finding a surface normal n from three affinely independent (definition is given in section 6.1.3) points p_1, p_2, p_3 such that,

$$n = (p_2 - p_1) \times (p_3 - p_1) \text{ is a normal to the plane through } \{p_i | i = 1,2,3\}. \quad (6.14)$$

The cross product of two vectors v, w is a vector $v \times w$ with the following properties,

(a) $v \times w \perp v$, $v \times w \perp w$

(b) Positively oriented: $\det[v\ w\ v \times w] > 0$, *v and w are linearly independent (*important for culling*).*

(c) $\|v \times w\| = \|v\| \|w\| \sin(\theta)$, θ is the angle between v and w

(d) For vectors relative to an orthonormal basis, the cross product is:

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} \times \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \begin{bmatrix} \alpha_2\beta_3 - \alpha_3\beta_2 \\ \alpha_3\beta_1 - \alpha_1\beta_3 \\ \alpha_1\beta_2 - \alpha_2\beta_1 \end{bmatrix}$$ where anticommutativity, bilinearity hold.

6.1.3 Affine Transformations

For all the definitions in this part, figure 6.3 will be used.

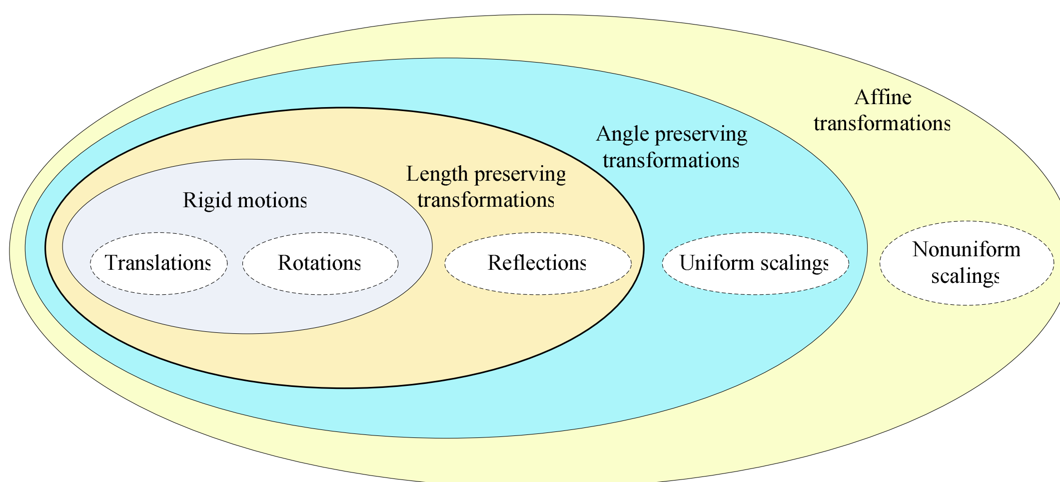


Figure 6.3 The group of affine transformations. The dashed ellipses are basic operations. Each group of transformations denoted by a solid ellipse is composed of operations inside that ellipse (Bergen, 2004, p.20).

As seen in figure 6.3, rigid motions group is composed of two subgroups: *Translations* and *rotations*. Translations have the following form,

$$T(x) = x + c \qquad (.15)$$

The rotations which are in fact linear transformations have the form,

$$R(x) = Bx \;\;, \text{ where } B^{-1} = B^{T} \wedge \det(B) = 1 \qquad (6.16)$$

As $B^{-1} = B^{T}$, this matrix is orthogonal. The important point is that, an orthonormal basis is transformed to an orthonormal basis if and only if the transformation matrix is orthogonal. The proof is as follows using the definition of dot product and orthogonality;

$$(Bb_i).(Bb_j) = (Bb_i)^T (Bb_j) = b_i^T B^T B b_j = b_i^T b_j = b_i.b_j = \delta_{ij}$$
$$= \begin{cases} 1, \text{if } i = j \\ 0, \text{if } i \neq j \end{cases}$$

$\therefore$

The length preserving transformation group is formed from rigid motion transformations and reflection transformation. A transformation in length preserving group is expressed as,

$$T(.) \in \textit{Group of length preserving transformations} \leftrightarrow \|T(x) - T(y)\| = \|x - y\|, \forall x, y$$

And for the affine transformations group to be length preserving, the following criterion should hold,

$$T(.) \in \textit{Group of affine transformations } T(x) = Bx + c \leftrightarrow B \textit{ is orthogonal}$$

This can be proved as follows using the axiom distribution of multiplication over addition, the definitions of orthogonality and dot product.

$$
\begin{aligned}
\|T(x)-T(y)\| &= \|Bx+c-By-c\| \\
&= \|Bx-By\| = \|B(x-y)\| \\
&= \sqrt{B(x-y).B(x-y)} = \sqrt{(B(x-y))^T B(x-y)} \\
&= \sqrt{(x-y)^T B^T B(x-y)} = \sqrt{(x-y)^T (x-y)} \\
&= \|x-y\|
\end{aligned}
$$

$\therefore$

A reflection transformation through a plane through origin is defined as follows,

$$T(x) = Bx,\ \ B \textit{ is orthogonal and } \det(B) = \text{-}1 \tag{6.17}$$

The group of uniform scaling about the origin is defined as follows,

$$T(x) = \alpha x\ \ ,\ \ \alpha \text{ is a } \textit{scalar and } \alpha \neq 0 \tag{6.18}$$

Notice in figure 6.3 that, the group of angle preserving transformations is the composition of length preserving transformations and uniform scaling. Therefore generalizing the property of the length preserving groups is possible as follows,

$$
\begin{aligned}
&\forall T(.) \in \textit{Group of angle preserving transformations}, \\
&\exists \alpha > 0 \ni \|T(x)-T(y)\| = \alpha\|x-y\|
\end{aligned}
$$

Finally, the group of nonuniform scaling about the origin has the following form,

$$
\begin{aligned}
&T(x) = \lfloor \alpha_{ij} \rfloor x,\ \ \alpha_{ij} \text{ are scalars}, \\
&\alpha_{ij} \neq 0 \leftrightarrow i = j
\end{aligned}
\tag{6.19}
$$

Referencing to (Bergen, 2004, p. 21), any affine transformation T can be constructed from three fundamental transformations as translation, rotation and nonuniform scaling.

One of the important points in computer graphics is the calculation of the lighting on a plane being transformed by an affine transformation. Two parameters, the normal of the plane and its distance from the origin of the reference system should be recalculated. This can be done as follows,

Let P' be an image of P under affine transformation $T(x) = Bx + c$. The definition of P' is the set of $x \in R^m$ such that,

$$P' = \left\{ x \in R^m \middle| n'.x + \delta' = 0 \right\},\ n' \textit{ is the normal,} \\ \delta \textit{ is the dis}\tan\textit{ce from reference system origin} \tag{6.20}$$

Then, by expressing P' in terms of T^{-1},

$T^{-1}(x) = B^{-1}(x - c)$

$$\forall x \in P', n.B^{-1}(x - c) + \delta = 0\ \ ,\ (\textit{equation of } P), n \text{ is normal } \delta \text{ is the distance} \\ \text{to the origin of the reference system} \tag{6.21}$$

By definition of orthogonality and dot product,

$$\begin{aligned} & n^T B^{-1}(x - c) + \delta = 0 \\ & \left(\left(B^{-1} \right)^T n \right)^T (x - c) + \delta = 0 \\ & \left(\left(B^{-1} \right)^T n \right).(x - c) + \delta = 0 \\ & \left(B^{-1} \right)^T n.(x - c) + \delta = 0 \\ & n' = \left(B^{-1} \right)^T n = Bn \\ & \delta' = \delta - n'.c \end{aligned} \tag{6.22}$$

6.2 Important Geometric Primitives for Computer Graphics and Definitions of Convex Combination and Convex Hull

A convex hull of a point set A, denoted by $conv(A)$ is the smallest object containing A. The convex hull of a finite point set $A = \{a_1,...,a_n\}$ can be expressed

as convex combinations of A (Bergen, 2004, p. 23). *A convex combination* of A is any point x defined by

$$x = \sum_{i=1}^{n} \alpha_i a_i , \ni \sum_{i=1}^{n} \alpha_{\mathrm{i}} = 1, \text{ and } \alpha_i \geq 0 \quad (6.23)$$

6.2.1 Polytopes

The convex hull of a finite point set A is defined as a convex polytope P, that is,

$$P = conv(A) \quad (6.24)$$

As stated in (Bergen, 2004, p. 24), the vertices $vert(P)$ of a polytope P is the smallest set $X \subseteq A$ such that $P = conv(X)$.

A simplex is the convex hull of an affinely independent set of points. As depicted in (Bergen, 2004, p. 24), simplices of one, two, three and four vertices are points, line segments, triangles, and tetrahedra respectively; the dimension of a polytope is the dimension of its affine hull; finally the set of two and three dimensional polytopes are the set of convex polygons and the set of convex polyhedra respectively. For detailed relations, figure 6.4 should be inspected.

As stated in (Bergen, 2004, p. 28), polytopes may be represented by a combination of half spaces instead of vertex representation. One example for this case is *the discrete-orientation polytopes (DOP)* used as bounding volume representation. A discrete-orientation polytope is the intersection of fixed number of *slabs*. A slab is a region of space bounded by a pair of parallel planes. A *k-DOP* is the intersection of k slabs.

6.2.2 *Polygons*

A closed chain of line segments that bound a region of a plane forms a polygon. The coplanar points forming the polygon are called the vertices of the polygon. A polygon is called simple if no two edges intersect other than the edges that share a vertex. For more information refer to (Bergen, 2004, p. 29).

6.2.3 *Quadrics*

A quadric is an object that has quadratic surface elements. The interior part of the object is part of a quadric. Therefore they are called as solids rather than surfaces. For more information refer to (Bergen, 2004, p. 32).

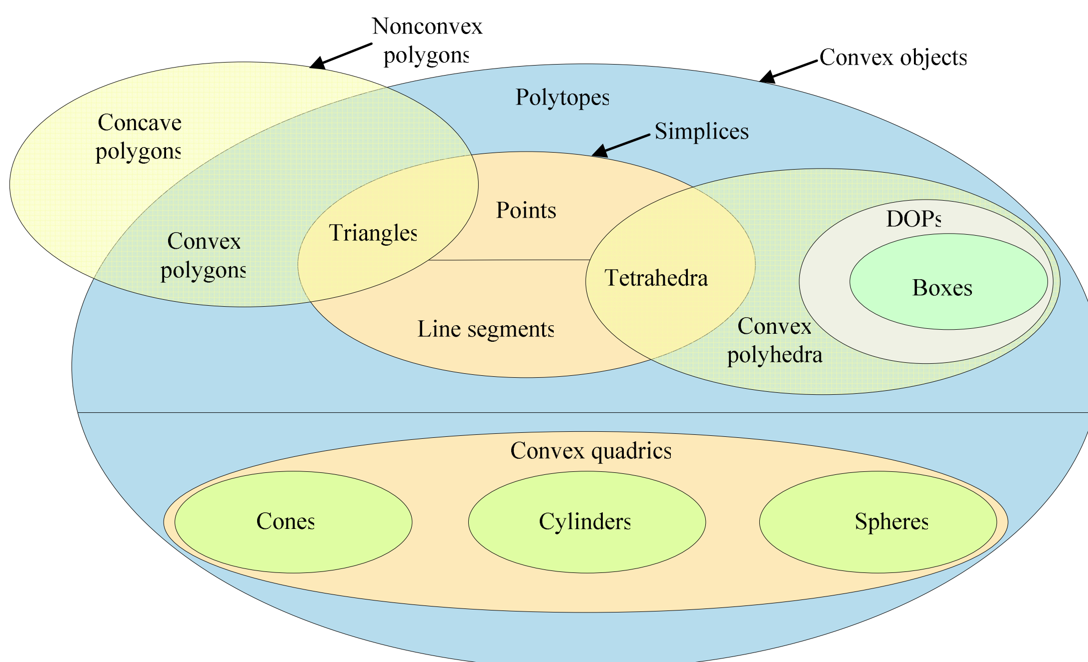


Figure 6.4 Important primitives for computer graphics (Bergen, 2004, p.24).

6.3 Minkowski Sum and Its Relation with an Intersection Test

The Minkowski sum of two objects A and B (A and B can be any primitive object such as a polygon, a polytope, quadrics and etc…) is defined as follows (Bergen, 2004, p. 33);

$$A + B = \{x + y | x \in A, y \in B\} \tag{6.25}$$

The definition should not be misunderstood, because it does not mean the addition of two points $x \in A, y \in B$. Considering the definition of affine space given in the previous section, addition of two points is not defined. Hence, according to those definitions, a point is a vector from the origin of the coordinate system to that point. The sum of two such vectors is a point that is obtained by adding the sum vector to the origin of the coordinate frame. Then the new object $A + B$ is the set of points that is covered by sweeping B's origin over all points of A as shown in figure 6.5.

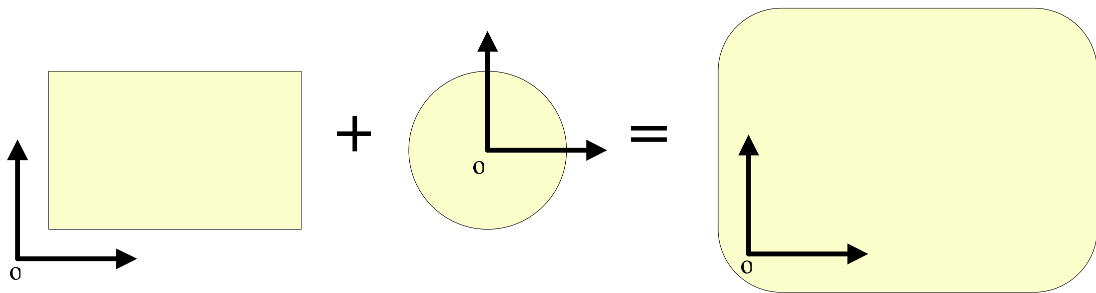


Figure 6.5 The resultant swept volume (sphere-swept volume in this case) formed by the Minkowski sum of a box and a sphere (Bergen, 2004, p. 33).

The Minkowski sum of two convex objects is convex. The Minkowski sum of two polytopes is a polytope. The proofs can be found in (Bergen, 2004, p. 34, p. 35).

Several queries on a pair of objects can be performed in terms of their *configuration space obstacle (CSO)* by using the Minkowski sum. For this purpose, the negation operation on an object is defined as follows,

$$-B = \{-y | y \in B\} \tag{6.26}$$

Then the CSO of objects A and B is the object $A + (-B)$ that is $A - B$. $A - B$ is the set of all vectors from a point of B to a point of A in the same coordinate system. *The intersection query* on a pair of objects can be expressed in terms of the CSO of the two objects such that a pair of objects intersects if and only if their CSO

contains the origin. That is, if the objects intersect, they will have a common point, that is the vector from this point to itself which is the zero vector in the CSO of these objects. This property can be presented as follows,

$$A \cap B \neq \phi \leftrightarrow 0 \in A - B \tag{6.27}$$

The distance $d(A,B)$ between two objects A and B is as follows,

$$d(A,B) = \min\{\|x - y\| \mid x \in A, y \in B\} \tag{6.28}$$

The same distance definition can be done in terms of the CSO of the objects A and B as follows,

$$d(A,B) = \min\{\|x\| \mid x \in A - B\} \tag{6.29}$$

The following property holds for two convex objects A and B,

$$\forall (A,B) \text{ pair of convex objects, } \exists! x \in A - B \ni d(A,B) = \min\{\|x\|\}$$
$$\text{that is } d(A,B) \text{ is the closest to origin } 0. \tag{6.30}$$

Referencing to (Bergen, 2004, p.23, p. 36), this can be proved by a contradiction. Assume that,

(A,B) are convex objects and expressing distance in terms of CSO of A and B,

$$d(A,B) = \min\{\|x_1\| \mid x_1 \in A - B\} = \min\{\|x_2\| \mid x_2 \in A - B\} \rightarrow$$

$$\exists x_3 = \sum_{i=1}^{2} \alpha_i x_i, \sum_{i=1}^{2} \alpha_i = 1, \alpha_i \geq 0 \;\; (\textit{convex combination of } x_i)$$

$$\ni \min\{\|x_3\|\} < d(A,B)$$

On the other hand, using the convexity of $A - B$ due to the fact that the Minkowski sum of two convex objects is a convex object as stated above,

$x_3 \in A - B$ and $\mathrm{d(A,B)} \leq \min\{\|x_3\|\}$ *which is a contradiction.*

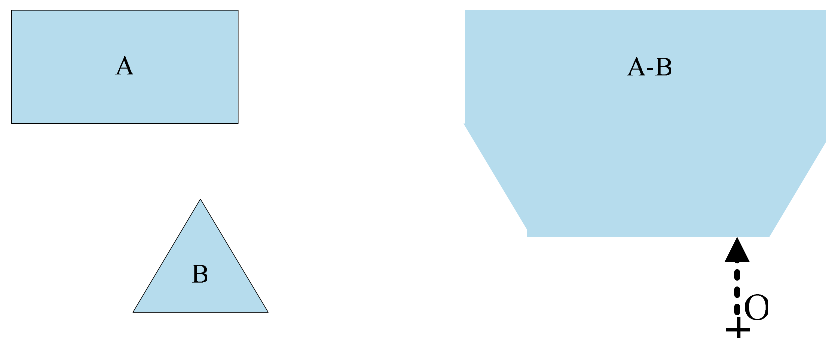
$\therefore$

The researcher should be aware that the uniqueness of the point of $A - B$ closest to the origin does not imply that the distance between two convex objects is realized by a unique pair of points. There may exist multiple $x \in A, y \in B \ni \|x - y\| = d(A,B)$. But, all the closest pairs map to the same point $x - y \in CSO(A,B)$.

The penetration depth of two intersecting objects can be expressed in terms of their CSO (Bergen, 2004, p. 36). The penetration depth of a pair of intersecting objects is the length of the shortest vector over which one of the objects needs to be translated in order to bring the pair tangent to each other. The penetration depth $p(A,B)$ can be expressed as,

$$p(A,B) = \inf\{\|x\| x \notin A - B\} \tag{6.31}$$

It should be noticed that infimum (the greatest lower bound) is used instead of minimum, because $A - B$ is a closed set meaning that it also includes its limit point. Considering figure 6.6, for a pair of penetrating objects, the penetration depth is realized by a point on the boundary of $A - B$ that is closest to the origin. As mentioned before, more than one $x \in A, y \in B$ pair in the object space may map to the origin $0 \in A - B$, hence that point on the boundary of $A - B$ is not unique.



(a)

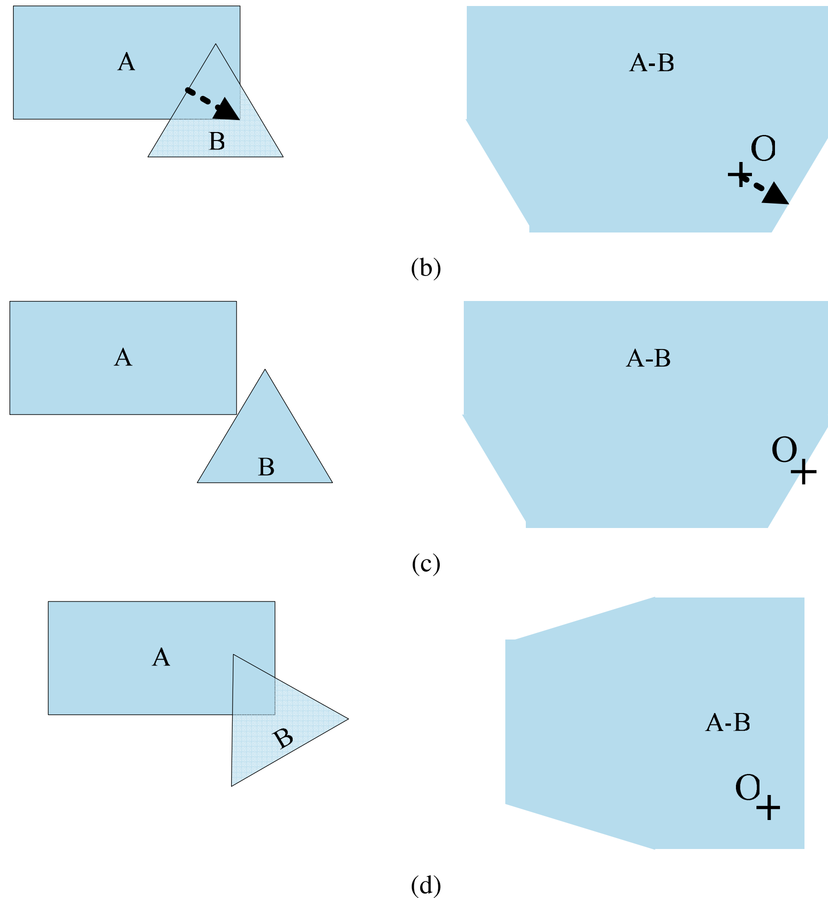


Figure 6.6 A pair of convex objects on the left and corresponding CSO on the right. (a) Nonintersecting, the origin is outside the CSO. The arrow denotes the distance. (b) Intersecting, the origin is inside the CSO. The arrow denotes the penetration depth. (c) After a translation of B over the penetration depth vector, the objects are in contact. The origin lies on the boundary of the CSO. (d) After a rotation of B, the shape of the CSO changes. (Bergen, 2004, p.38)

6.4 Separating Axis Test

Separating axis test (SAT) is an important method that is the result of the separating hyperplane theorem originating from convex analysis as stated in (Ericson, 2005, p. 156). The theorem states that, given convex objects *A* and *B* whether they intersect or there exists a separating plane *P* where *A* and *B* exist in the opposite half spaces. If such a separating plane exists, the normal *L* of that plane is called as the separating axis. Figure 6.7 depicts the theorem in 2-D. The detailed proofs can be found in (Bergen, 2004, p. 78), (Bergen, 2004, p.110).

The theorem is not valid for concave objects and in order to prove that two concave objects do not intersect, a curved surface separating those objects should be found. But the method of convex decomposition can be applied to both concave objects and SAT can be applied to the convex partitions created.

In the context of collision detection, SAT has an important role in determining whether intersection occurs or not between any convex objects such as lines, boxes, spheres or any simple polytope. Furthermore, together with the *CSO* of the objects defined in the previous section; time of collision, penetration depth, contact point and contact normal can also be computed by this method for both static and moving objects.

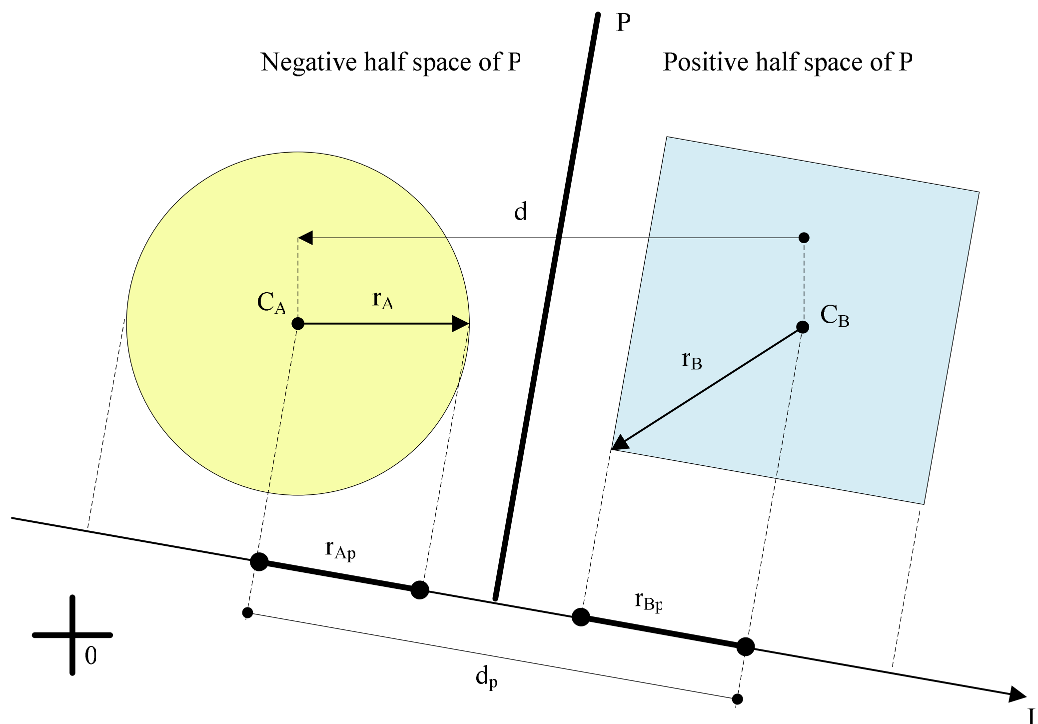


Figure 6.7 One of the separating axis tests between objects A and B. P is the separating plane for A and B; L is the normal of the plane. Inspired from (Ericson, 2005, p. 158).

Considering figure 6.7 and assuming all is valid for R^3 let,

$$d = C_A - C_B; \quad d, C_A, C_B \in R^3$$
$$r_{Ap} = L.r_A$$

$$r_{Bp} = L.r_B; \qquad r_{Ap}, r_{Bp}, d_p \in R; r_A, r_B, L \in R^3$$
$$d_p = L.d \tag{6.32}$$

Considering equations (6.32), the objects do not intersect if $r_{Ap} + r_{Bp} < d_p$ for all possible separating axis tests.

The complexity of the objects to be tested is important for the efficiency of SAT. Assuming object A has f_1 faces and e_1; object B has f_2 faces and e_2 edges, total of $f_1 + f_2 + e_1 e_2$ SATs should be performed. These tests are for the axes parallel to the face normals of object A, the axes parallel to the face normals of object B and axes parallel to the vectors formed by the cross products of all edges of object A and all edges of object B. As soon as a separating axis is found, the algorithm can terminate with no intersection. If no separating axis is found as a result of all the tests, it means that the objects are intersecting.

As stated in the following sections, SAT is used between appropriate convex bounding volumes throughout the thesis work.

6.5 Primitive Bounding Volumes for Collision Detection Used in the Software

In a typical interactive 3-D application, simple bounding volumes that can capture the actual geometry of the objects are used instead of the whole render geometry. Bounding volumes can be of several types such as axis aligned bounding boxes (AABBs), spheres, oriented bounding boxes (OBBs), convex hulls, discrete orientation polytopes (k-DOPs), and etc… In this section, the AABBs, OBBs and sphere bounding volumes will only be considered, because OBBs and sphere bounding volumes are the only bounding volume primitives used for fast collision tests in the narrow phase of the collision pipeline (see figure 6.9) apart from the more precise collision tests in the scope of the thesis work. Additionally, AABBs are used in the construction of the bounding volume hierarchy tree for the broad phase of the collision pipeline (see section 6.6). The calculation of collision parameters such as

time of impact (TOI), penetration depth, contact points in local and world coordinate frames and contact normal are performed in the narrow phase of the collision pipeline. Therefore the detailed overview of these parameters can be found in the following sections.

For detailed treatment of bounding volumes, the researcher should refer to (Ericson, 2005) and (Bergen 2004). In figure 6.8 several types of bounding volumes are shown for the same render geometry.

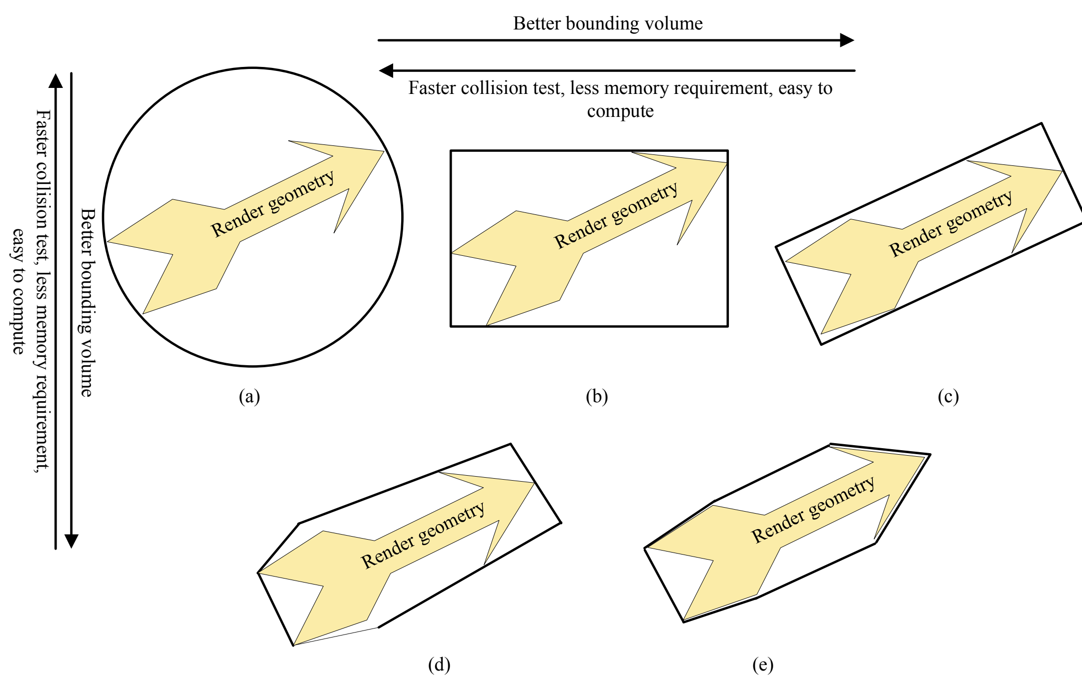


Figure 6.8 Several bounding volumes for the same render geometry. The outer thick solid lines are the bounding volumes. (a) Sphere bounding volume, (b) AABB, (c) OBB, (d) 6-DOP, (e) Convex hull. Inspired from (Ericson, 2005, p. 77).

6.5.1 Axis Aligned Bounding Boxes

Axis aligned bounding box is one of the simplest bounding volumes. It is formed from six sides that have normals always parallel to the corresponding coordinate system. In the thesis work, the AABBs are represented using minimum and maximum coordinate values along each axis of the render geometry local coordinate system. The bounding volume is the space between two opposing corners with

minimum and maximum coordinates respectively. With this representation, an AABB can be formed by defining minimum vertex coordinate p_{min} and maximum vertex coordinate p_{max} as the endpoints of one of the diagonals of the rectangular prism volume in the local coordinate system of the render geometry such that,

$$\mathrm{AABB} = \left\{ \begin{array}{l} \forall p = (p_x, p_y, p_z) \middle| (x_{\min} \le p_x \le x_{\max}) \wedge (y_{\min} \le p_y \le y_{\max}) \wedge \\ (z_{\min} \le p_z \le z_{\max}); p \in R^3, p_{\min} = (x_{\min}, y_{\min}, z_{\min}) \in R^3, \\ p_{\max} = (x_{\max}, y_{\max}, z_{\max}) \in R^3 \end{array} \right\} \tag{6.33}$$

The center c of the AABB is defined as the algebraic mean of p_{min} and p_{max}. Considering the above representation, two axis aligned bounding boxes AABB_1 and AABB_2 with minimum and maximum vertex coordinates $p_{1,min}$, $p_{2,min}$, $p_{1,max}$, $p_{2,max}$ respectively, intersect if and only if they intersect on all of the coordinate axes such that,

$$\begin{array}{l} \mathrm{AABB}_1 \cap \mathrm{AABB}_2 = \phi \leftrightarrow \\ \left(\begin{array}{l} (x_{1,\max} < x_{2,\min} \vee x_{2,\max} < x_{1,\min}) \vee \\ (y_{1,\max} < y_{2,\min} \vee y_{2,\max} < y_{1,\min}) \vee \\ (z_{1,\max} < z_{2,\min} \vee z_{1,\max} < z_{2,\min}) \end{array} \right) \\ \ni \forall i = 1,2; p_{i,\min} = (x_{i,\min}, y_{i,\min}, z_{i,\min}) \in \mathrm{AABB_i}, \\ \qquad p_{i,\max} = (x_{i,\max}, y_{i,\max}, z_{i,\max}) \in \mathrm{AABB_i} \end{array} \tag{6.34}$$

The intersection test described above should be done in the same coordinate system; that is, the AABBs should be either in the local coordinate system of AABB_1 or in the local coordinate system of AABB_2 or in the world coordinate system. Throughout the thesis work, the local frame of AABB_1 is used as the reference coordinate system; the computation and the update of the AABB are performed dynamically by finding the minimum and maximum coordinates of the local frame relative to the local origin.

6.5.2 Sphere Bounding Volumes

A sphere bounding volume S is represented by its center coordinates c and its radius r such that,

$$S = \{\forall p \| \|p - c\|^2 \leq r^2; p, c \in R^3; r \in R\} \tag{6.35}$$

Considering this representation, two sphere bounding volumes S_1, S_2 with centers c_1, c_2 respectively and radii r_1, r_2 respectively intersect if and only if the distance between their centers is less than the sum of their radii such that,

$$\begin{aligned} &\mathrm{S}_1 \cap \mathrm{S}_2 \neq \phi \leftrightarrow \\ &\left(\|c_2 - c_1\|^2 \leq (r_1 + r_2)^2\right) \ni c_1, c_2 \in R^3, r_1, r_2 \in R \end{aligned} \tag{6.36}$$

During the thesis work, the construction of the sphere bounding volume is performed by first computing the AABB of the render geometry. Then the center of the computed AABB is selected as the sphere center; the maximum extent among the three axes is selected as the sphere radius. The calculations are all relative to the local coordinate frame of the render geometry. During the collision test between two bounding spheres, the reference coordinate frame is selected as the coordinate system of the first sphere bounding volume. The update of the bounding sphere involves only translation along with the actual render geometry, because the sphere is rotation invariant. For the preceding computations more precise but at the same time more computationally demanding techniques such as gradient descent based methods or principal component analysis can be utilized.

6.5.3 Oriented Bounding Boxes

Oriented bounding boxes are similar to AABBs except that they may have arbitrary alignment. Although representation methods based on principal component analysis exist in literature; throughout the thesis work, an OBB is defined by its

center c, orientation matrix M representing local axis and positive half width extents vector $e=(e_x, e_y, e_z)$ such that,

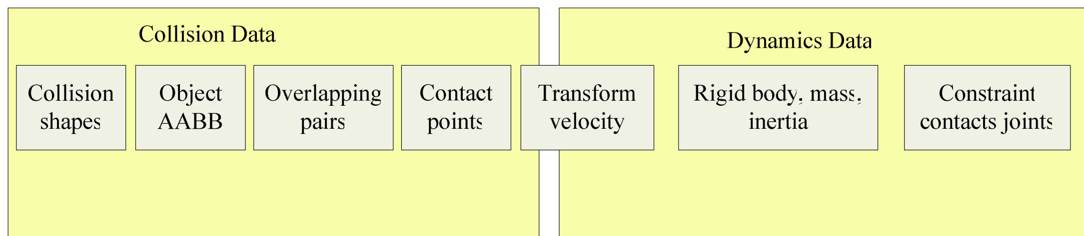
$$\text{OBB} = \left\{ \begin{array}{l} \forall p = (p_x, p_y, p_z) \middle| p = c + M \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}; p, c, e \in R^3, M \in R^{3x3}, \\ |\alpha| \in R \leq e_x, |\beta| \in R \leq e_y, |\gamma| \in R \leq e_z \end{array} \right\} \quad (6.37)$$

In order to detect intersection and to compute the collision parameters between two OBBs, the separating axis test (SAT) is used. For the details of SAT, refer to section 6.4. The collision test between two OBBs is performed relative to the local coordinate system of the first OBB under consideration throughout the thesis work.

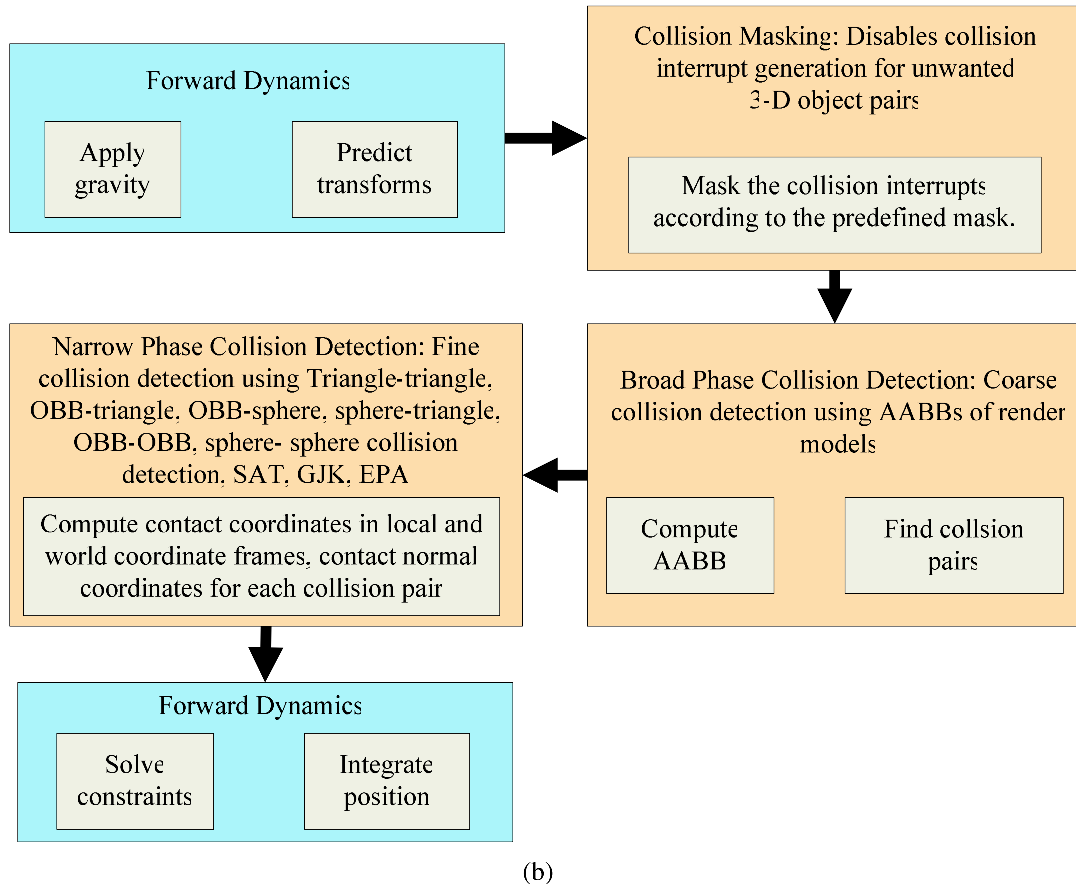
The interested researcher may refer to figures 9.18 and 9.19 for implementations of and comparison between various collision detection primitives done during the thesis work.

6.6 Collision Detection Pipeline Used in the Software

In an interactive simulation, collisions between objects are handled in several consecutive stages forming a collision pipeline. The aim of this pipeline is to decrease the computational load and memory requirements of collision detection while favoring the accuracy of collision tests, contact and penetration depth computations between the geometries of the 3-D objects. The overview of the collision detection pipeline is given in figure 6.9.



(a)



(b)

Figure 6.9 (a) An overview of how the collision data and dynamics data are stored in the developed software using Bullet. (b) The physics pipeline implemented in the developed software using Bullet. The red blocks represent how the collisions are handled in three stages.

This section will briefly explain the three important stages of the pipeline as implemented in the scope of the thesis; collision masking, broad phase and narrow phase respectively.

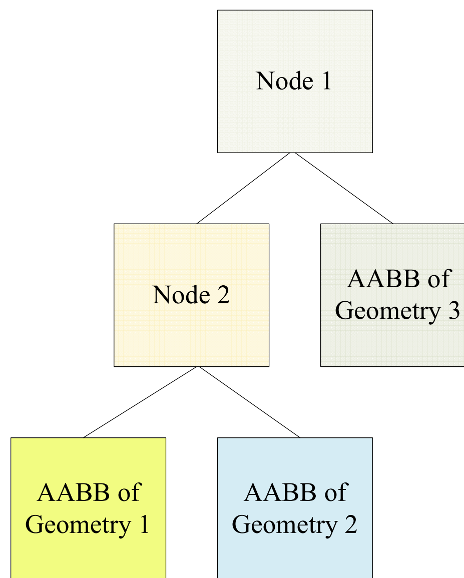
6.6.1 Collision Masking

Collision masking is a brute force collision filtering technique to define the geometries that will be considered in the collision detection process. It is the first stage of the collision detection pipeline. In the initialization stage of the simulation, each geometry is given a group and mask identity number to be used in the masking test. So, the geometries with no matching identity number will not collide to or receive collisions from other geometries. Therefore, only the geometries with

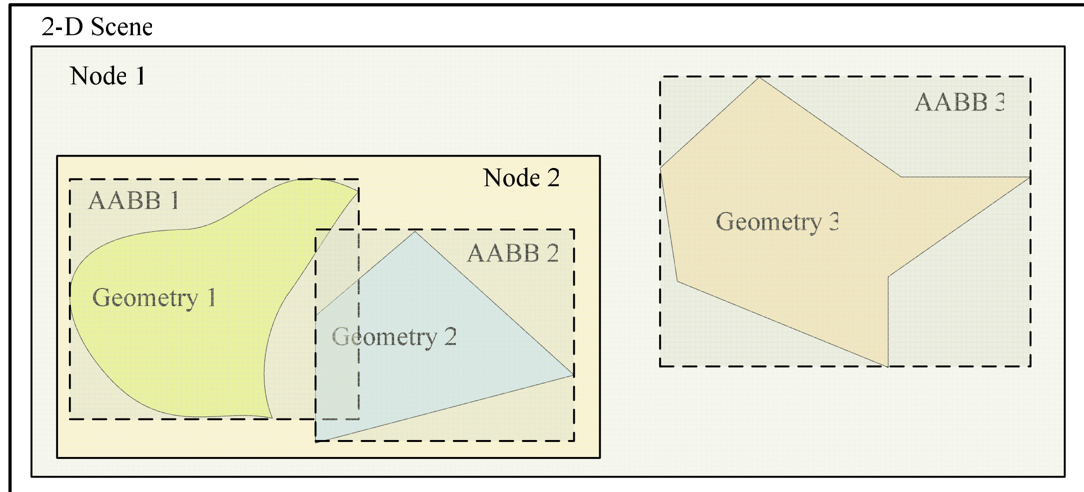
matching identity numbers will be considered in the further stages of the collision detection pipeline.

6.6.2 *Broad Phase*

Broad phase stage implemented in the software throughout the thesis work consists of a model partitioning scheme called as dynamic bounding volume hierarchy based on axis aligned bounding boxes (AABB). A bounding volume hierarchy is composed of a tree structure. Each leaf of the tree contains the bounding volume of the actual geometry. Nodes in the deeper levels of the tree are enclosed in a larger bounding volume and grouped in nodes towards the root node recursively such that each node of the tree maintains a bounding volume for a subset of the geometric primitives. The bounding volume hierarchy tree structure and the corresponding scene are seen in figure 6.10.



(a)



(b)

Figure 6.10 (a) A bounding volume hierarchy. (b) The geometries and collision models represented by the tree. In (b), the thin continuous lines represent the nodes, the dashed lines indicate the AABBs, and the outer thick line is the 2-D scene border.

The overall aim of this stage is to reduce the computational costs of the collision detection as in the collision masking stage. The arrangement of the bounding volumes of the geometries in a tree structure reduces the time complexity of the computations logarithmically in the number of tests performed whereas that time complexity is reduced by a constant factor with bounding volumes not arranged in a tree. As indicated in (Ericson, 2005, p. 235), for the latter case, although the collision detection tests are simplified by the bounding volumes, the number of collision tests to be performed remains the same so the asymptotic time complexity remains the same.

The bounding volume tree hierarchy is implemented as a preprocessing step in the developed software to increase the runtime performance. The tree structure is dynamic meaning that according to the topology changes in the meshes belonging to the scene, new nodes representing the bounding volumes of the newly created meshes can be added to or old nodes representing the unnecessary bounding volumes can be removed from the tree. For example when a soft cloth mesh is cut into many pieces, new nodes are inserted to the tree representing the bounding volumes of the mesh pieces. To increase the performance, two bounding volume tree hierarchy is used; one for the static objects and the other is for the moving objects. In the thesis

work, the objects with zero mass are defined as static. During runtime, nodes belonging to one tree can be detached and attached to the other dynamically or vice versa. The software implementation details used for bounding volume tree hierarchy construction, partitioning, node insertion and removal strategies can be found in (Ericson, 2005, chap. 6).

According to (Ericson, 2005, p. 236), the issues to be considered in order to balance the performance and the accuracy of this stage are as follows:

- The nodes in a subtree should be near to each other to favor spatial coherence
- A minimal bounding volume that will capture the topology of the objects should be used for each node to prevent false overlaps and therefore false collision test results. This also results in a minimal total bounding volume.
- Removing a node close to the root node, eliminates more bounding volumes from collision detection tests than removing a node at the deeper levels of the tree.
- The bounding volume tree should be balanced in its node structure and content so that whenever a branch is not traversed, it can be pruned to increase the performance.
- The bounding volume tree should have the minimal memory requirements.

The bounding volumes reported as colliding are directed to the narrow phase stage found further in the collision detection pipeline. The object pairs reported as colliding may be actually overlapping or not, depending on the actual geometries of the objects and their bounding volumes. Considering the figure 6.10 (b), the geometry 1 and the geometry 2 are not actually overlapping; but they will be reported as colliding due to the collision of their AABBs. On the other hand, the AABB of the geometry 3 does not overlap with the other AABBs. So it is impossible for the actual topology of the geometry 3 to collide with the topologies of the other objects in the scene; therefore no collision pairs including the geometry 3 will be reported in this stage.

Dynamic bounding volume hierarchies based on spheres, k-DOPs and oriented bounding boxes (OBB) also exist. Additionally, another broad phase technique namely “sweep and prune” can also be implemented in this stage. Spatial partitioning methods based on octree, k-d tree and binary space partitioning tree can also be considered for implementation in this stage according to the simulation needs. Only the dynamic bounding volume hierarchy based on AABB is implemented in the current software. For the other types of bounding volume hierarchies, for the “sweep and prune” scheme and for the spatial partitioning methods the researcher should refer to (Ericson, 2005) and (Bergen, 2004).

6.6.3 Narrow Phase

Narrow phase is the final part of the collision detection pipeline. Only the collision tests that pass the collision masking and broad phase are handled by this phase. In this phase, if the candidate objects for collision pass the tests performed here, it is understood that they are actually colliding. Then the collision parameters such as contact points in local coordinate frames and in world coordinate frames, contact normal and penetration depth are computed in this phase. In this phase, the collision models used for visual models can be convex hulls or the triangular element mesh of the visual object itself. It is seen that, the tests done in the narrow phase are much more costly than the previous sections. On the other hand these tests are much more precise. Triangle-triangle collision detection is performed in this phase to perform the collision check over all the triangular elements forming visual mesh of the candidate objects. For the results of implementation practices on collision detection regarding the comparison of triangle-triangle collision detection and sphere-sphere collision detection refer to figures 9.18 and 9.19. Other collision detection schemes such as sphere-triangle, OBB-triangle, ray-triangle tests are also performed in this section. In the narrow phase, the collision with the soft-soft bodies and rigid-soft bodies are performed by assuming the existence of AABBs bounding each vertex of the soft object. Therefore the previously mentioned for rigid bodies also apply for the soft body collision detection.

For collision detection between the moving objects, the Minkowski summation is used. The CSO of the moving objects are computed assuming, one of the objects static and the other moving relative to the static one. Then the intersection test mentioned in section 6.3 is applied. It is impossible to give all the mathematics beneath these tests here, therefore the interested researcher should refer to (Ericson, 2005), (Bergen, 2004) and (Möller, 1997) for mathematical theory of the implemented collision tests. Two important methods for collision detection, penetration of two convex objects; and for solving constraints between the collision object primitives are briefly mentioned below.

6.6.3.1 Gilbert-Johnson-Keerthi Algorithm (GJK) for Collision Detection between Convex Objects and Expanding Polytope Algorithm (EPA) for Penetration Depth Calculation

GJK is an iterative method for solving collision between convex objects. It can be generalized for any type of collision methods mentioned before, for application to polytopes, quadrics, Minkowski sums of convex objects and images of convex objects under affine transformations. GJK is an iterative method for approximating the point closest to the origin of *A-B, the CSO of A* and *B* convex objects (see figure 6.6). This point is approximated as follows. At each iteration, a simplex (see section 6.2.1) is constructed that is contained in *A-B* and lies nearer to the origin than the simplex constructed in the previous iteration. A simplex is constructed support mapping of *A-B*. A support mapping of a convex object *A* is a function s_A that maps a vector **v** to a point of A as follows,

$$s_A(v) \in A \ni v.s_A(v) = \max\{v.x : x \in A\} \tag{6.38}$$

The result of (6.38) is a support point. Each new support point is added to the simplex, the closest point to the origin is calculated and the farthest point is discarded at each iteration. The iteration stops, when a change in distance between newly found points decreases below a threshold. For detailed explanation refer to (Bergen, 2004, chap. 4).

As GJK algorithm was used for computing collisions, contact points and contact normals between the convex objects during the thesis work; Expanding Polytope Algorithm (EPA) was used for the penetration depth calculation between two colliding convex objects. Like GJK, EPA is an iterative algorithm depending on the CSO of two convex objects. For detailed coverage refer to (Bergen 2004, p. 147). The researcher may refer to figure 9.4 for the computed collision parameters displayed in the green overlaid box on the bottom left of the screen.

6.6.3.2 Solving the Constraints at Mechanical Joints – Linear Complementary Problem (LCP)

The virtual environment developed during the thesis work has a 3-D user interface for transforming several objects, getting information about them and etc… This user interface becomes visible when a collision between the user hand and an anatomical model is detected (see figures 9.4 (d), 9.8 (a)-(b), 9.9 (a), 9.10 (a)-(b), 9.11 (a)-(b)). As seen from the figures, the user interface contains buttons, a slider and an information box. Each of these is attached to the base of the user interface with appropriate constraints so that buttons have no degrees of freedom and the slider has only one degree of freedom. The new position of the buttons and the slider when a user collides is computed considering the applied force, contact direction and the constraints at each time step. In the context, this is formularized as a linear complementary problem (LCP). The solution is accomplished by Gauss-Siedel method. The mathematical details can be studied from (Ericson, 2005, chap. 9), (Baraff, 1989), (Bridson, 2003) and (Lacoursiére, n.d.). The end result is that, the buttons remain at their original positions when the contact ends; the slider remains at the place where the user last touches.

6.7 Mass-Spring Systems and Numerical Solutions for Governing Differential Equations

This section involves various mass spring topologies used to model the dynamical objects in the 3-D environment throughout the thesis work. Additionally, the

numerical solution techniques for the differential equations governing the dynamics of the mass spring topologies are given.

6.7.1 1-D 2-D and 3-D Mass Spring Systems and Governing Differential Equations

Mass-spring systems are preferred for modeling solid elastic 1-D, 2-D and 3-D dynamic systems in real time. 1-D, 2-D and 3-D mass-spring systems with dampers are presented in figure 6.11. Masses are placed at the vertices of the 3-D model in this particular case. The edges connecting vertices are represented by springs.

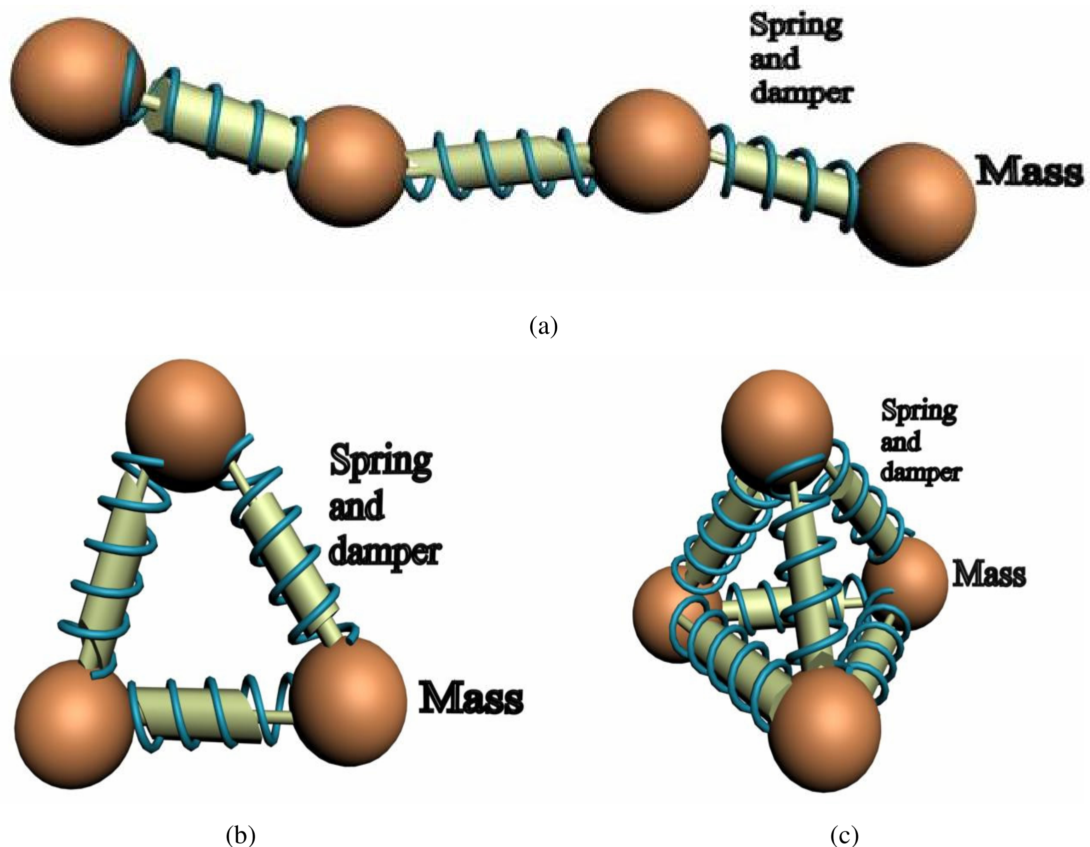


Figure 6.11 Examples of mass-spring systems. (a) 1-D mass-spring-damper system, (b) 2-D (triangular) mass-spring-damper system and (c) 3-D (tetrahedra) mass-spring-damper system.

Although simple and not computationally demanding these systems have drawbacks as stated in (Müller, Stam, & James, 2008a, p.10). The mass-spring network setup defines the behavior of the object. The spring constants are hard to be tuned for the desired behavior. Mass-spring systems cannot capture volumetric

properties directly. Because of these limitations, better models such as FEM are preferred in spite of their computational demand.

1-D mass-spring systems are used for modeling 1-D elastic objects such as hair or rope. 2-D mass-spring systems arranged as triangular elements are used to model 2-D elastic objects such as a skin, a cloth or a paper. 3-D mass-spring systems arranges as tetrahedral elements are used to model 3-D volumetric elastic objects such as human organs.

The physical formulation of the mass-spring system can be stated as follows. For a mass spring system composed of a set of N particles with masses m_i, positions x_i and velocities v_i where $i \in 1,...,N$, the masses are connected with the connection set S of springs (i, j, l_0, k_s, k_d). i, j are indices of the adjacent masses, l_0 is the rest length, k_s is the spring stiffness and k_d is the damping coefficient as stated in (Müller, & et al., 2008a, p.11). x_i, x_j are the positions and v_i, v_j are the velocities of the masses respectively. Then the spring forces on the adjacent particles of a spring are,

$$f_i = f^s(x_i, x_j) = k_s \frac{x_j - x_i}{x_j - x_i}\left(\left\|x_j - x_i\right\| - l_0\right) \qquad (6.39)$$

$$f_j = -f^s(x_i, x_j)$$

The forces are proportional to the elongation of the spring $\left\|x_j - x_i\right\| - l_0$ from its equilibrium state.

The damping forces are proportional to the velocity difference projected onto the spring. That is,

$$f_i = f^d(x_i, v_i, x_j, v_j) = k_d(v_j - v_i).\frac{x_j - x_i}{\left\|x_j - x_i\right\|} \qquad (6.40)$$

$$f_j = f^d(x_j, v_j, x_i, v_i) = -f_i \qquad (6.41)$$

Notice that the conservation of momentum holds, therefore $f_i + f_j = 0$. The combination of the forces is,

$$f(x_i, v_i, x_j, v_j) = f^s(x_i, x_j) + f^d(x_i, v_i, x_j, v_j) \tag{6.42}$$

Then, considering the second law of Newton, $F = m\ddot{x}$, this ordinary differential equation should be solved for the acceleration $\ddot{x}$ of particles that is the 2nd derivative of the position with respect to time. That is,

$$\ddot{x} = \frac{d^2 x}{dt^2} = \frac{F}{m} \tag{6.43}$$

A Nth order ordinary differential equation can be written in terms of N coupled 1st order ordinary differential equations. So, $F = m\ddot{x}$ can be written as two coupled ordinary differential equations as follows,

$$\begin{aligned} \dot{v} &= \frac{f(x,v)}{m} \\ \dot{x} &= v \end{aligned} \tag{6.44}$$

The analytical solutions of these equations are respectively,

$$\begin{aligned} v(t) &= v_0 + \int_{t_0}^{t} \frac{f(t)}{m} dt \\ x(t) &= x_0 + \int_{t_0}^{t} v(t) dt \end{aligned} \quad \text{and the initial conditions are } v(t_0) = v_0 \;\;,\;\; x(t_0) = x_0 \tag{6.45}$$

As it is seen from the above analytic solutions, simulation is in fact time integration. In the following sections the numerical integration methods to solve the initial value problem with *the initial value* $x(t_0)$ will be given. Therefore the problem at hand is to find a function satisfying the relation described by the ordinary differential equation $\dot{x} = f(x,t)$ where f is a known function, x is the state of the system and $\dot{x}$ is the derivative of x with respect to time. Theoretical background can be found at (Khalil, 2002), (Müller, & et al., 2008a) and (Witkin & Baraff, 2001).

6.7.2 Explicit Euler Integration

Assume that x is continuously differentiable function. Then, consider the Taylor Series Expansion of x at point t_0 with a small perturbation Δt from t_0 as follows,

$$x(t_0+\Delta t) = x(t_0) + \frac{1}{1!}\frac{dx(t)}{dt}\Delta t + \frac{1}{2!}\frac{d^2x(t)}{dt^2}\Delta t^2 + ... + \frac{1}{n!}\frac{d^nx(t)}{dt^n}\Delta t^n \tag{6.46}$$

Linearizing the function at t_0 yields,

$$x(t_0+\Delta t) = x(t_0) + \frac{1}{1!}\frac{dx(t)}{dt}\Delta t + O(\Delta t^2) \tag{6.47}$$

Eliminating 2nd order error term $O(\Delta t^2)$ results in the linearized x as follows,

$$x(t_0+\Delta t) = x(t_0) + \frac{dx(t)}{dt}\Delta t = x(t_0) + \dot{x}\Delta t \tag{6.48}$$

The discreet step size is Δt, $\dot{x}$ defines the norm of the step along the Δt direction. That is, it is used to calculate the change Δx in x corresponding to Δt. Consider that for N dimensional case, Δt is a N dimensional vector specifying the step directions.

For this linearization case, x should at least be C^1 continuous. Then the integral equations in (6.45) can be solved numerically by linearizing $v(t)$ and $x(t)$ about t_0 and then iterating the following equalities by neglecting the 2nd order error terms respectively.

$$v_{t+1} = v_t + \dot{v}\Delta t + O(\Delta t^2) \cong v_t + \dot{v}\Delta t \tag{6.49}$$

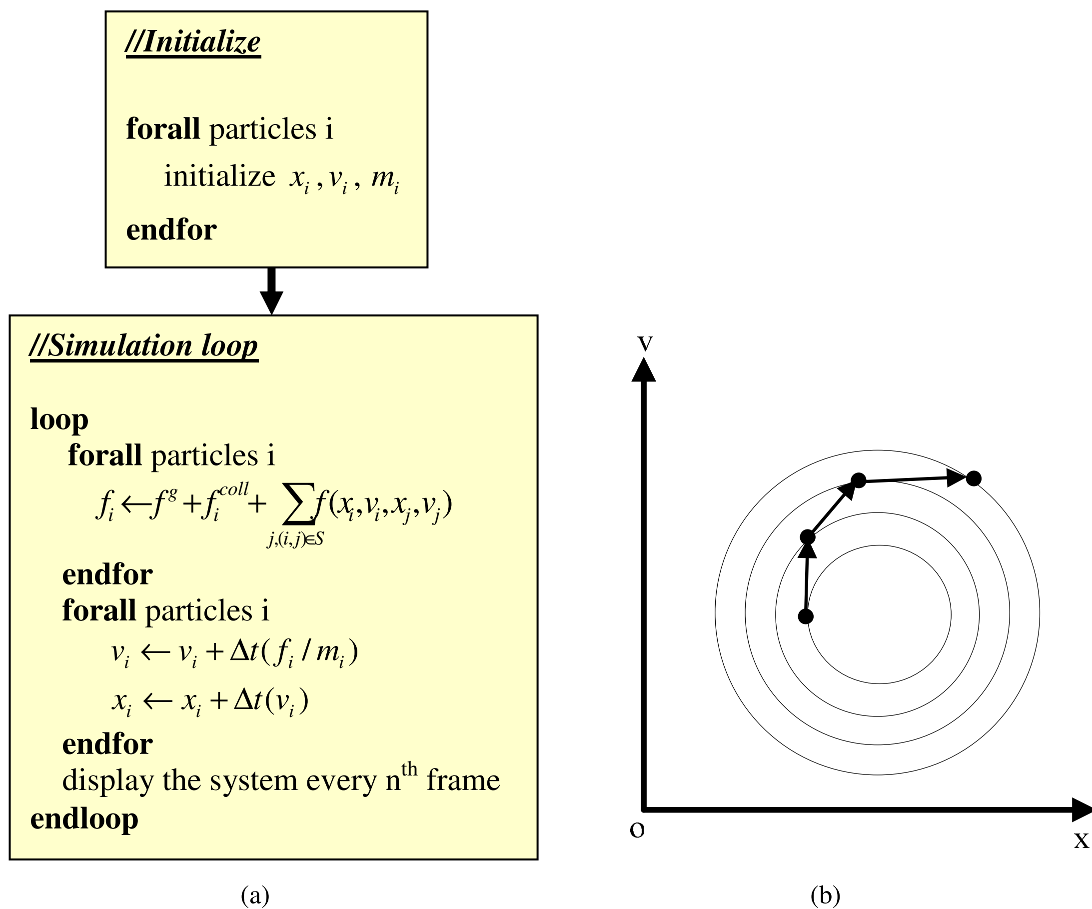
$$x_{t+1} = x_t + \dot{x}\Delta t + O(\Delta t^2) \cong x_t + \dot{x}\Delta t \tag{6.50}$$

Notice that, t is the frame number and Δt is the time interval between two consecutive frames for the case of real time graphics rendering. Plugging equalities in (6.44) into (6.49) and (6.50) yields respectively,

$$v_{t+1} = v_t + \frac{f(x_t, v_t)}{m} \Delta t \tag{6.51}$$

$$x_{t+1} = x_t + v_t \Delta t \tag{6.52}$$

Calculation of (6.51) and (6.52) are *Explicit Euler Integration Method*, that is the values of v_{t+1} and x_{t+1} are calculated using the values v_t and x_t of the current time step by explicit formulas as also stated in (Müller, & et al., 2008a). The following three pseudocodes of algorithms can be found at (Müller, & et al., 2008a) and (Press, & et al., 2007).



(a) (b)

Figure 6.12 (a) Pseudocode for Explicit Euler Integration. f^g is the gravity force, f^{coll} is the forces due to the collisions. (b)The phase space representation of differential equation for the mass-spring system. The actual solution for differential equations form a concentric circles, but due to the linearization in Explicit Euler Integration, the particle velocity and position overshoots. Smaller time steps only makes this process occur in longer time but is not a complete solution.

Although Explicit Euler Integration is a simple method, it is unstable for large time steps. The phase space representation is shown in figure 6.12 (b). Therefore, during simulation, several time steps should be performed per each frame and damping is necessary otherwise the velocity will overshoot.

6.7.3 Second and Fourth Order Runge Kutta Integration

Considering the Taylor Series Expansion in (6.46), if the second order term is retained, the expansion of the function will have error terms starting with $O(\Delta t^3)$ hence the result will be 2nd order accurate. That is,

$$x(t_0 + \Delta t) = x(t_0) + \frac{1}{1!}\frac{dx(t)}{dt}\Delta t + \frac{1}{2!}\frac{d^2x(t)}{dt^2}\Delta t^2 + O\left(\Delta t^3\right) \tag{6.53}$$

Considering $\dot{x} = f(x(t),t)$, assume that f implicitly depends on time t that is $\dot{x} = f(x(t))$. Using chain rule,

$$\ddot{x} = \frac{\partial f}{\partial x}\dot{x} = \dot{f}f \tag{6.54}$$

Approximate $\dot{f}$ in terms of f by using Taylor Expansion of f as follows,

$$f(x_0 + \Delta x) = f(x_0) + \frac{1}{1!}\dot{f}(x_0)\Delta x + O(\Delta x^2) \tag{6.55}$$

Let $\Delta x = \frac{\Delta t}{2} f(x_0)$, then (5.54) can be written as,

$$\begin{aligned} f\left(x_0 + \frac{\Delta t}{2} f(x_0)\right) &= f(x_0) + \frac{\Delta t}{2} f(x_0)\dot{f}(x_0) + O(\Delta t^2) \\ &= f(x_0) + \frac{\Delta t}{2}\ddot{x}(t_0) + O(\Delta t^2)\ ,\ x_0 = x(t_0) \end{aligned} \tag{6.56}$$

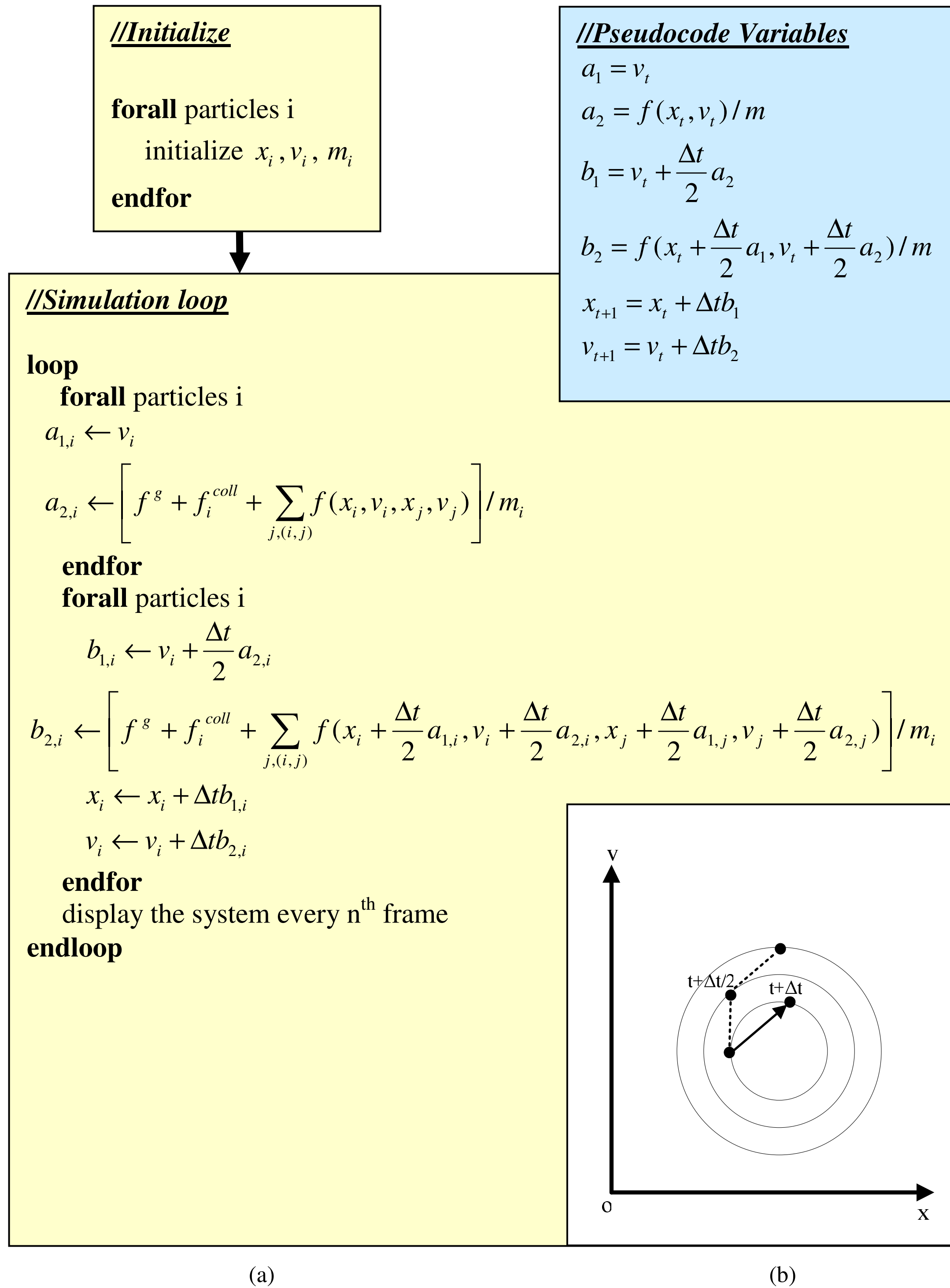
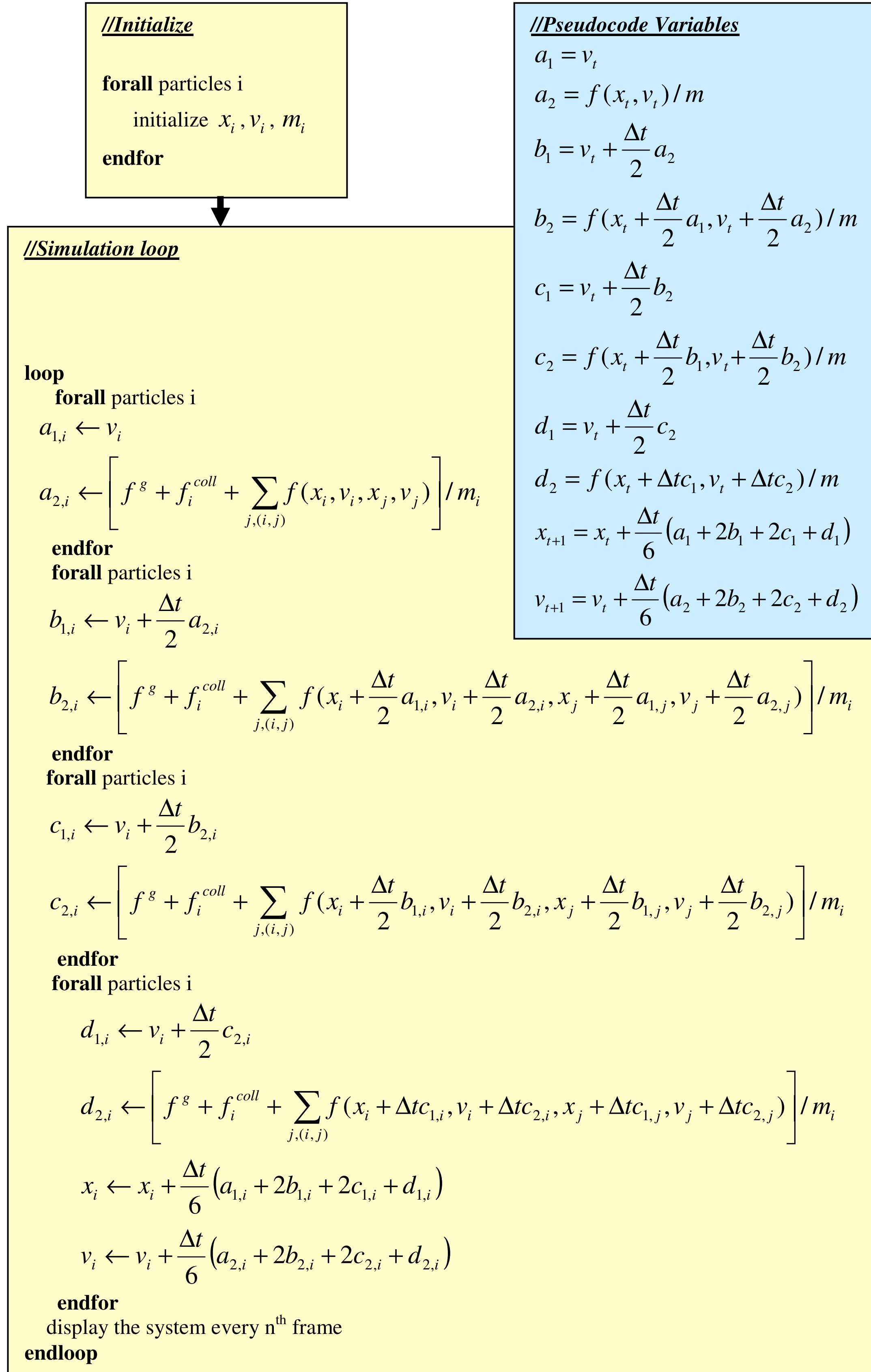
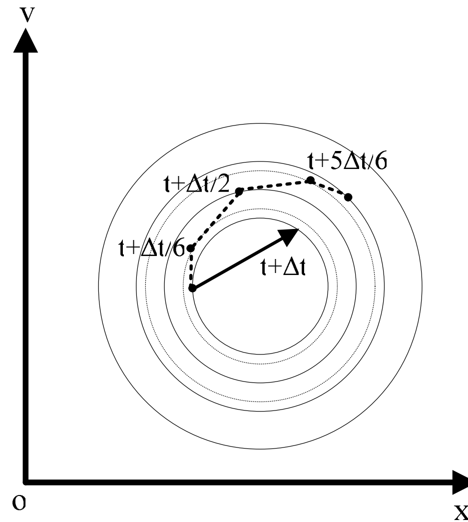


Figure 6.13 (a) Pseudocode for 2nd order Runge Kutta Integration. (b) The phase space representation of the differential equation. First, an Euler step is performed and then at the half of the step size, the second derivative is evaluated to update x at each frame.



(a)



(b)

Figure 6.14(a) Pseudocode for 4th order Runge Kutta Method. (b) Phase space representation.

Multiply both sides of (6.56) by Δt,

$$(\Delta t)f\left(x_0 + \frac{\Delta t}{2} f(x_0)\right) = (\Delta t)f(x_0) + \frac{\Delta t^2}{2}\ddot{x}(t_0) + O(\Delta t^3) \tag{6.57}$$

Use (6.57) in (6.53) considering $\dot{x} = f(x(t))$,

$$x(t_0 + \Delta t) = x(t_0) + (\Delta t)\left(f(x_0) + \frac{\Delta t}{2} f(x_0)\right) \tag{6.58}$$

Equation (6.58) indicates that, an Explicit Euler scheme is performed up to the half of the step size, then a second derivative is evaluated at the half of the step size to update x at each frame. Therefore this method is 2nd order accurate and more precise than the Explicit Euler scheme. The two Euler step evaluation brings a computational cost. Additionally the 2nd order Runge Kutta still lacks instability problems. 2nd order Runge Kutta is also an explicit numerical integration method. The pseudocode is given in figure 6.13 (a) and the phase space representation is given in figure 6.13 (b).

4th order Runge Kutta integration is similar, but it is 4th order accurate as opposed to 2nd order accuracy. This costs four times the computational load that of the Explicit Euler Integration. The pseudocode is given in figure 6.14 (a) and the phase space representation is given in 6.14 (b).

6.7.4 Verlet Integration

This method uses the values evaluated at the past steps to increase the stability and accuracy of the prediction at the current step. The method is accurate up to the 4th order. Consider the forward and backward Taylor Expansion of x as follows respectively,

$$\begin{aligned} x(t+\Delta t) &= x(t) + \dot{x}(t)\Delta t + \frac{1}{2}\ddot{x}(t)\Delta t^2 + \frac{1}{6}\dddot{x}(t)\Delta t^3 + O(\Delta t^4) \\ x(t-\Delta t) &= x(t) - \dot{x}(t)\Delta t + \frac{1}{2}\ddot{x}(t)\Delta t^2 - \frac{1}{6}\dddot{x}(t)\Delta t^3 + O(\Delta t^4) \end{aligned} \tag{6.59}$$

Sum the expansions in (5.58) as follows,

$$\begin{aligned} x(t+\Delta t) &= 2x(t) - x(t-\Delta t) + \ddot{x}(t)\Delta t^2 + O(\Delta t^4) \\ &= x(t) + [x(t) - x(t-\Delta t)] + \frac{f(t)}{m}\Delta t^2 + O(\Delta t^4) \end{aligned} \tag{6.60}$$

Then by letting $v(t) = [x(t) - x(t-\Delta t)]/\Delta t$, the followings are obtained,

$$\begin{aligned} x_{t+1} &= x_t + v_t\Delta t + \frac{f(x_t)}{m}\Delta t^2 \\ v_{t+1} &= \frac{(x_{t+1} - x_t)}{\Delta t} \end{aligned} \tag{6.61}$$

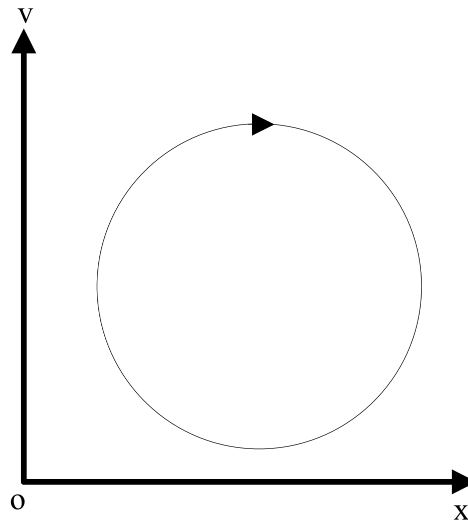


Figure 6.13 Phase space representation for Verlet Method. The energy remains constant with sufficiently small steps.

All the above methods were explicit integration methods. Those methods are stable for a limited range of time steps which depends on the stiffness of the springs; hence they are conditionally stable as stated in (Müller, & et al., 2008a). Smaller time steps should be used to maintain the stability of the simulation as the springs get stiffer. The real time simulation applications require unconditional stability for any value of time steps. This requirement can be satisfied by using implicit integration methods. An implicit integration uses the new of values of x and v for computing the followings,

$$\begin{aligned} v_{t+1} &= v_t + \frac{f(x_{t+1})}{m}\Delta t \\ x_{t+1} &= x_t + v_{t+1}\Delta t \end{aligned} \tag{6.62}$$

Equation (6.62) cannot be evaluated explicitly. Instead, the system can be solved for velocities by linearizing this nonlinear system at each time step using Newton-Raphson method. Then the linearized system can be solved using iterative methods such as Conjugate Gradients. As stated in (Müller, & et al. 2008a), although this integration scheme is unconditionally stable, it is slow so large time steps should be

performed, and additionally temporal details disappear due to the numerical damping.

6.8 Mesh Topology Processing and Mesh Refinement – An Example to Mesh Cutting

The method used in the developed software to process the topology of the meshes is referenced from (Coumans, 2009). The method works as follows: A ray is casted from the camera along the forward direction of the user hand towards the soft object. Then an intersection query is performed whether the raycasting is resulted in an intersection with an efficient closure to any of the constraints (lines) connecting the AABBs of the vertices of the geometric topology of the soft object (see section 6.6.3). If this is the case, create a *sphere s* with unit radius of 1 centered at the collision coordinates of the ray and the constraint. Then the distance of the collision point to the supporting vertices along the constraint are calculated numerically. The new vertices with appropriate velocity, position and mass are added to topology of the soft object and the connections between the cut part of the topology and the rest of the topology is broken. The position, velocity and the mass of the new vertex are calculated as the linear interpolation of the positions, velocities and the masses of two supporting vertices of the constraint (line) on which a new vertex is placed. Considering *the vertices a, b and the sphere s* given above, the problem is formularized as follows:

$$\min_{\|v\|<\varepsilon}(t) \tag{6.63}$$

ε is the user controlled value controlling the minimum distance from *sphere s* that can be considered as the surface of the sphere. $\|v\|$ is the distance to the collision point that is the center of *sphere s*. $t = t^*$ is the value at which the iteration goes into the ε - neighborhood of the surface of the *sphere s*. If no such t is found in a given step number than $t = -1$ is returned resulting in no topology process, else $t = t^*$ is

returned. The position x_c and the velocity v_c of *the new vertex c* are computed as follows:

$$
\begin{aligned}
x_c &= x_a + (x_b - x_a)t^* \\
v_c &= v_a + (v_b - v_a)t^*
\end{aligned}
\tag{6.64}
$$

The mass m_c of the new vertex c is computed as follows:

$$
\begin{aligned}
m &= m_a + (m_b - m_a)t^* \\
f &= \frac{m_a + m_b}{m_a + m_b + m} \\
m_c &= mf
\end{aligned}
\tag{6.65}
$$

If one of the supporting vertices is static that is it has zero mass. The mass of the supporting vertex with the positive mass is assigned to the new vertex, and the mass of that supporting vertex is doubled. If both supporting vertices have zero mass, the newly created vertex is assigned a zero mass. The process is repeated for all *the (a, b) vertex pairs* in the $\|v\|$ neighborhood of *the collision point v*. The geometric representation of the process is given in figure 6.14.

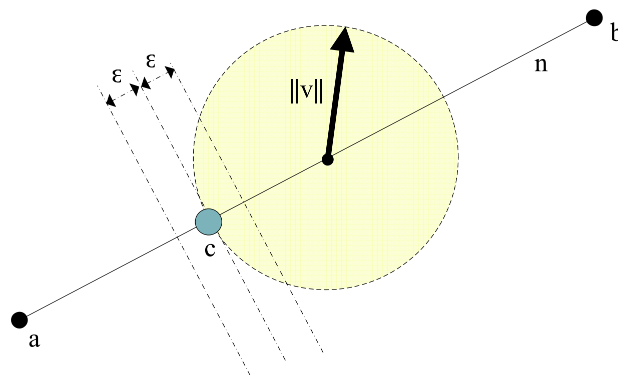


Figure 6.14 The geometric representation of the cutting topological operation. v is the collision point of the ray with the constraint n; a and b are the existing supporting vertices of the constraint n; c is the newly added vertex. This process is done for all the *(a, b)* vertex pairs in the ||v|| neighborhood of the collision point v.

Finally, the node hierarchy data structure representing the soft object is updated accordingly. The topology processes were all done just for the surface models, no volumetric model was evaluated.

The research may refer to figures 9.12, 9.13 and 9.28 for the implementations done during the thesis period. Figure 9.27 is a stand-alone application compiled with NVIDIA PhysX for test purposes.

6.9 Haptic Rendering with Rigid and Deformable Models

Haptic senses provide important cues for getting information about the geometry and the structure of the object. Therefore, in order to provide a haptic feedback to the user in the virtual environment, haptic device was used. The technical details of the device are given in section 8.3.

Haptic rendering module of the software was at its development stage at the time this thesis was written. The main task of the module was to model the anatomical parts to create a haptic perception for the user when a tissue or organ was touched. Additionally, when the user applied a force over a threshold, a topology of the mesh would be altered, for example a fracturing of a rigid bone or cutting of a deformable organ would be performed. Initial haptic rendering module developed, was tested using the same dynamic modeling principles – namely mass spring model – and collision detection techniques mentioned in the previous sections. OpenHaptics API was used for the implementation. The instability of typical numerical integration method namely Explicit Euler Method used in this module was also observed for large time steps and also for the high forces loading that makes the mesh system diverge from its equilibrium point in these tests. The haptic rendering module initial results can be seen in figures 9.29.

CHAPTER SEVEN

FEATURE SEGMENTATION TRACKING AND POSE ESTIMATION METHODS USED FOR AUGMENTED REALITY APPLICATION DEVELOPMENT DURING THE THESIS WORK

This chapter presents a basic application created to fulfill the need of tracking a human hand and head motions without any motion tracking device precisely. The tracking results was planned to be used for controlling a 3-D virtual object and to perform pan movements for the user head in a synthetic environment. These were necessary to be immersed in a 3-D virtual environment because it was not possible to get the motion tracking device till the end of the second year of the thesis work.

The preferred way for tracking a user hand holding a known marker was tracking from real time video frames taken from a calibrated stationary single camera. The reference for this method was (Kato & Billinghurst, 2006). For in depth understanding, the researcher should refer to (Tekalp, 1995) and (Forsyth & Ponce, 2003). This method was in fact a registration meaning to estimate the rotation and translation parameters of the tracked features over the video frames taken from a calibrated camera. Therefore it was also used for registering a 3-D virtual object with the tracked object in real time. This type of application is called as an augmented reality application in the literature. In a same way, the user head was tracked using facial features such as the structures of eyes, nose and mouth. This method was referenced from (Viola & Jones, 2004) and (Bradsky & Kaehler, 2008). The explanations relating to this chapter will be kept relatively brief, because the focus of the thesis work was the real time computer graphics and physics simulation.

7.1 Feature Segmentation

Considering the human hand tracking, a known, rectangular planar marker is used. The assumption that all the features lie on the same plane decreases the number of unknowns in the rotation and translation matrix. The features segmented were the

corners of the black rectangle. The researcher can refer to (Bradsky & Kaehler, 2008) for corner extraction in sub pixel accuracies.

For the 2-D face tracking task, the Haar features of the face are used. A threshold is applied to the sums and differences of rectangular image regions. An integral image technique is used for rapid computation of the value of rectangular regions or 45 degree rotated versions. Then a statistical boosting technique is used to create face and non-face classification nodes characterized by high detection and weak detection. Then the algorithm organizes the weak classifier nodes of a rejection cascade. Meaning that, the first group of classifiers is selected that best detects image regions containing a face while allowing mistaken detections; the next classifier group is the second-best at detection with weak rejection, etc… In test mode, a face is detected if and only if it makes through the entire cascade. The details are given in (Bradsky & Kaehler, 2008, p. 508).

7.2 Feature Tracking and Pose Estimation

Considering the case of hand tracking, in every video frame, the corners of the rectangle is segmented and their coordinates are tracked. The rotation matrix and translation vector for the tracked rectangle were computed using homography matrix computation. Hence from there on, the tracked features in the consecutive frames are related. The computed rotation and translation parameters of the rectangle in 3-D space were then used to control the orientation and translation of a 3-D virtual user interface tablet in the synthetic environment.

Considering the case of the head tracking, the center coordinate of the rectangular area including the detected human face was tracked. As the tracking was performed in 2-D space, only the pan movements of the head corresponding to the pan movements of the camera of the synthetic scene were possible in the virtual environment. The algorithm flows of the tracking schemes are given in figures 7.1 and 7.2 respectively. The implementation results can be seen in figures 9.1 (a)-(f).

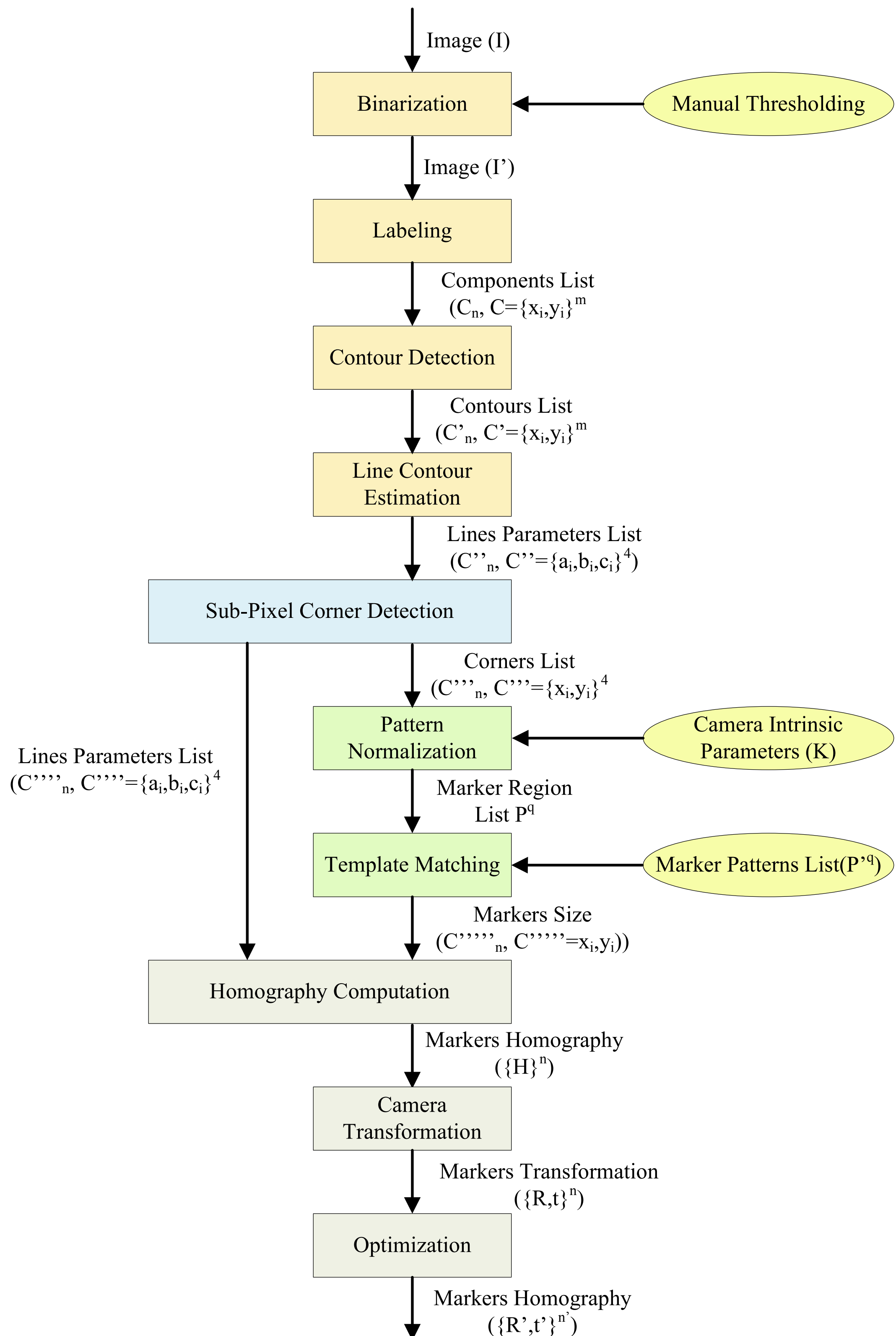
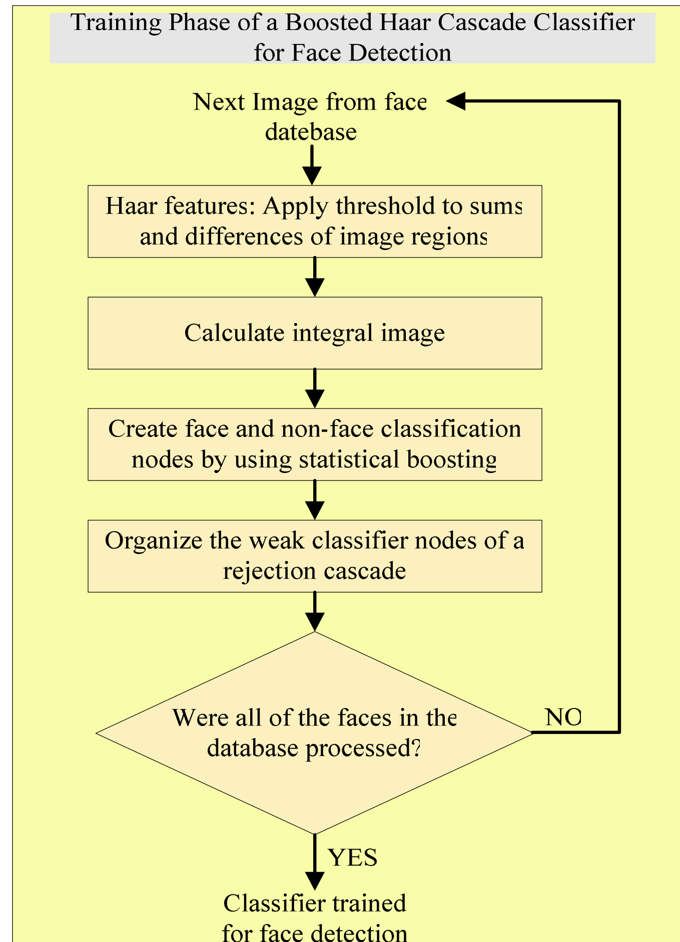
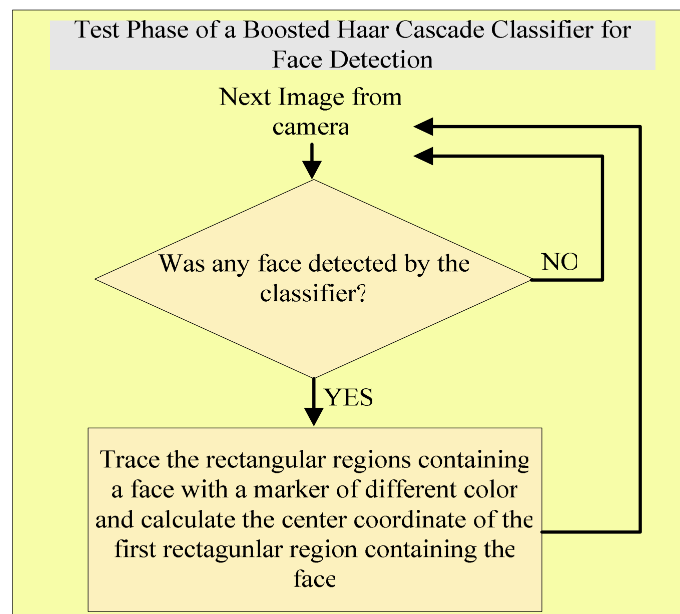


Figure 7.1 Tracking algorithm for a planar object held by the user hand. This algorithm was both used for controlling a virtual user interface tablet in the synthetic environment and also for registering a 3-D virtual object with a feature in real-time (Kato & Billinghurst, 2006).



(a)



(b)

Figure 7.2 (a) Training phase of Haar Classifier for face detection. (b) Test phase of the classifier.

CHAPTER EIGHT

ESTABLISHMENT AND CURRENT SETUP OF COMPUTER GRAPHICS AND VIRTUAL REALITY LABORATORY

This chapter will explain the establishment process of the computer graphics and virtual reality laboratory in Dokuz Eylül University.

A new research laboratory has been established in the scope of this thesis work. The laboratory is located in Dokuz Eylül University Electrical and Electronics Engineering Department (DEU EEE Department). The aim of the laboratory is to provide necessary equipment and development environment for undergraduate and graduate level researches on computer graphics, scientific simulation and visualization, computer vision, virtual environments and augmented reality. Laboratory establishment period and current setup of the laboratory will be explained in this chapter. Then the technical details of fundamental laboratory equipments that a researcher should know will be given briefly.

The thesis project and the laboratory establishment are supported in the scope of Dokuz Eylül University Scientific Research Project (BAP) with the support code of 2008.KB.FEN.027. VESTEL Electronics supported the establishment process with a LCD panel. The establishment process of the laboratory and the important technical details of several equipments are as follows.

Table 8.1 lists the equipments that the computer graphics and virtual reality laboratory in DEU EEE Department has. The establishment progress of the computer graphics and virtual reality laboratory in DEU EEE Department can be followed in date order can be followed from figures 8.1 to 8.3.

Table 8.1 DEU EEE Department Computer Graphics and Virtual Reality Lab. Equipment List.

Equipment No.	Equipment Name
1	VESTEL 102" Full HD LCD
2	Polhemus Fastrak 6 DOF motion tracker
3	Sensable Phantom Omni haptic device
4	5DT Data Glove 5 Ultra USB (left and right pairs)
5	5DT HMD 800-26 3-D head mounted display
6	Logitech QuickCam Pro 9000 webcams (2 pieces for augmented reality and computer vision applications)
7	Intel Quad Core and Core i7 based computers using Microsoft Windows XP
8	ATI X1550 based graphics card
9	NVIDIA GeForce GTX 295 based graphics card
10	Sanyo data projector

(a) (b)

Figures 8.1 (a) and (b) are two views from the laboratory by the end of October 2008.

(c) (d)

Figures 8.2 (a) and (b) are two views from the laboratory by the end of December 2009.

(e) (f)

Figures 8.3 (a) and (b) are two views from the laboratory by the end of August 2010.

8.1 VESTEL LCD Panel

VESTEL Electronics supported the laboratory establishment with a LCD panel. It is a 102” Full-HD 1080p model with composite, PC, YPbPr, HDMI inputs. The LCD is used as a primary or secondary display device together with the HMD.

8.2 Polhemus Fastrak Motion Tracking System

Polhemus Fastrak motion tracker is a six degrees of freedom (6 DOF) motion tracking system. It is capable of tracking both position and orientation using electromagnetic fields. The near field, low frequency magnetic field vectors are generated via three concentric, stationary antennas in the transmitter. The generated magnetic field vectors are detected by three concentric, stationary antennas in the receiver. The position and the orientation of the receiver relative to the transmitter is calculated by using the sensed signals as the input arguments of a mathematical formulation. Figures 8.4 and 8.5 show the main components of Polhemus Fastrak motion tracking system. The details of technical specifications of the tracking system and the software development kit can be found in (Polhemus, 2009).

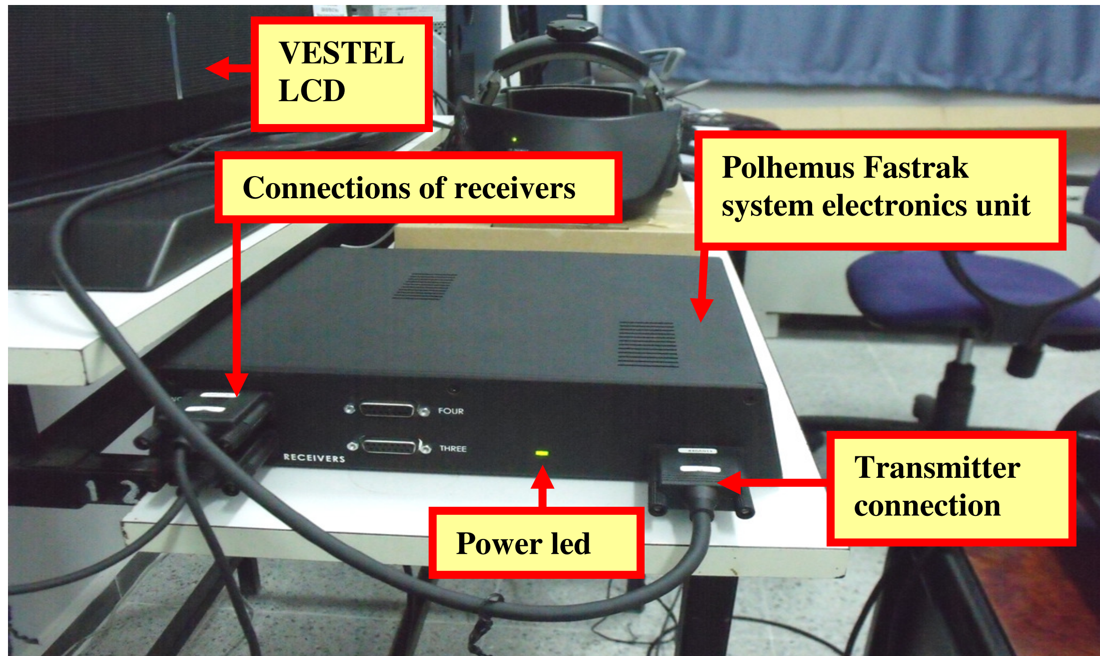


Figure 8.4 Polhemus Fastrak motion tracking system front side connections.

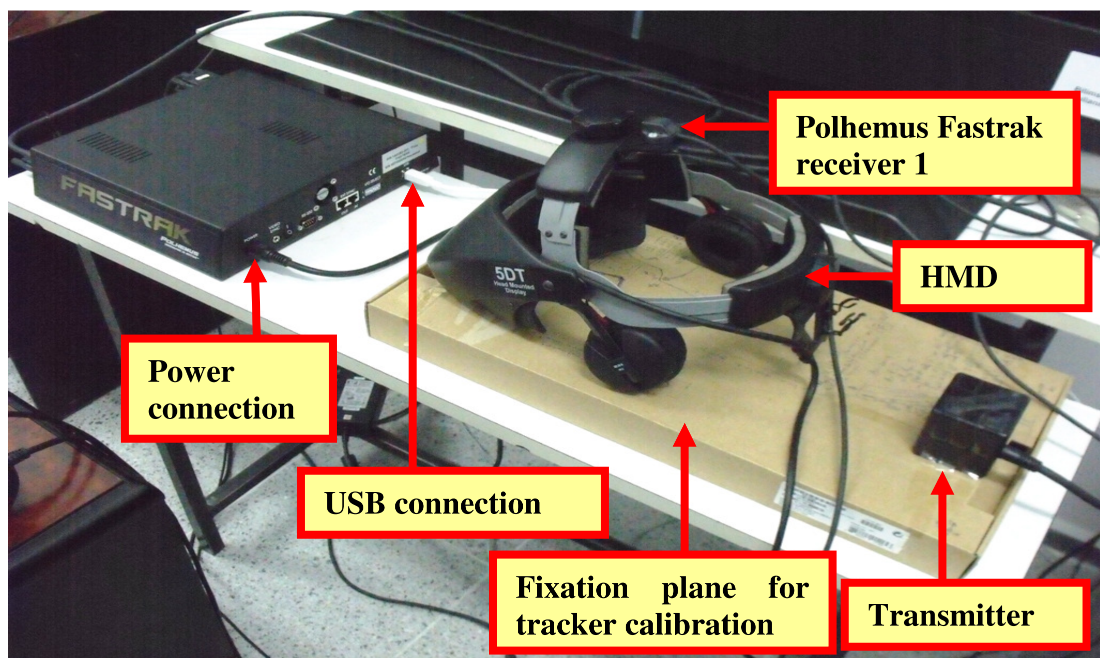


Figure 8.5 Polhemus Fastrak motion tracking system rear side connections.

The tracking system includes a system electronics unit, a power adapter, a transmitter and up to four receivers. The sampling frequency of the receivers is dependent on the number of receivers physically connected to the system electronics unit. Therefore, a single receiver is sampled at 120 Hz which is the maximum sampling rate. Two receivers are sampled at 60 Hz each. Three receivers and four

receivers configurations operate at 40 Hz and 30 Hz sampling frequency for each of the receivers respectively. The interface with the host computer can either be RS – 232 interface or USB interface.

The tracking system is designed to provide the optimum accuracy when the standard receivers are within the 76 cm of the standard transmitter. The receivers are all – attitude. The static accuracy of the system is 0.08 cm RMS for X, Y or Z receiver position, and 0.15° RMS for receiver orientation. The positional and angular resolutions are 0.0005 cms/cm of range, and 0.025° respectively. The latency of the system from the center of receiver measurement period to beginning of transfer from output port is 4.0 ms.

By default, the output position data is X, Y, Z position (cm or inch) in Cartesian Coordinate System considering the reference point on the transmitter as the origin of the system. By default, the output orientation data is azimuth, elevation and roll in Euler Angles considering the reference point on the transmitter as the origin of the system. If needed, the application developer can select direction cosines or quaternion as an output data. The metric unit can be selected as inch or metric units. The type of output data can be ASCII or binary.

Prior to using Polhemus Fastrak, it should be calibrated for all the connected receivers. In order to understand the calibration procedure, the developer should know where the origins and reference frames on the transmitter and on the receivers are located respectively. The related origins and reference frames are given in figure 7.6. Additionally, working knowledge on boresighting, reference frame alignment, hemisphere tracking should be gathered.

8.2.1 Reference Frame Alignment

Reference frame alignment is needed as the first part of the calibration procedure prior to each use.

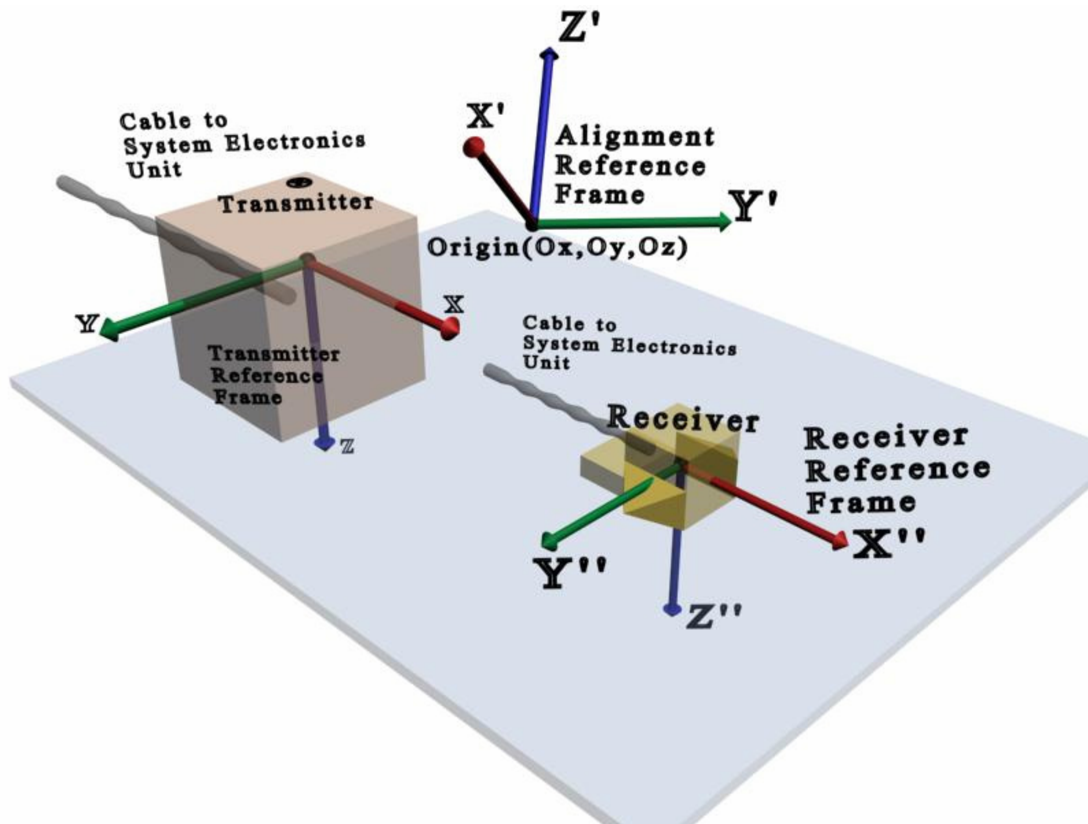


Figure 8.6 The coordinate systems of the transmitter and the receiver.

The alignment creates a reference frame where the position and orientation data gathered from the receiver are referenced to. The receiver is placed on the desired location within the limits of the transmitter and the origin $O(Ox, Oy, Oz)$ of the transmitter – receiver pair is defined. That location will be the origin for only that particular transmitter – receiver pair. Then, two more spatial points (Xx, Xy, Xz) and (Yx, Yy, Yz) are defined in the positive X' and Y' directions respectively so that rays OX' and OY' are orthogonal. The norms of these rays are defined by the user. By default they are defined as 200cm. Finally, using three points O, X' and Y', a 2-D space that is a plane can be defined. The normal of the plane is calculated by the cross product of the rays OX' and OY'. Hence a Cartesian Reference Frame is built.

8.2.2 Boresighting

Boresighting is not mandatory in the calibration procedure. Boresighting aligns the orientation of the receiver with the user coordinate system. This means that, when

boresighting is applied, the azimuth, elevation and roll at that moment will be referenced as zero values. Then the further orientation measurements will be performed relative to this reference orientation.

8.2.3 Hemisphere Tracking

The magnetic fields generated by the transmitter are symmetric. Therefore there are two mathematical solutions to each set of receiver data as depicted by (Polhemus, 2009, p. 48). To provide a unique solution to the equations, only one hemisphere named as the current hemisphere is used during the tracking. Outside the current hemisphere, mathematical ambiguities i.e. sign flips occur. These ambiguities will result in positioning and orienting the 3-D virtual model (for the application in the scope of the thesis) inappropriately. Therefore the tracking system provides a hemisphere tracking feature to track the current hemisphere that the receiver in. But to enable this option, the hemisphere tracking should be enabled when the receiver is in a known initial condition. This means that, the receiver should be in the $+X$ direction relative to the transmitter initially, prior to turning on the tracking system and enabling this feature.

8.2.4 Output Data

The needed output data can be acquired from the motion tracking system by software configuration prior to the operation. For most of the time X, Y and Z Cartesian coordinates of position, azimuth, elevation and roll Euler orientation angles and orientation quaternion will be adequate. Additionally, configuring the output data format as binary instead of ASCII will reduce the data packet size.

8.2.5 Angular Operational Envelope

If needed, the azimuth, elevation and roll angles can be constraint to intervals. If the receiver is outside these intervals, the user is notified.

8.2.6 Position Operational Envelope

If needed, the positions along $\pm X$, $\pm Y$, $\pm Z$ directions can be constraint to intervals. If the receiver is outside these intervals, the user is notified.

In most of the operation purposes, configuring the parameters mentioned in section 8.2.1 to 8.2.6 will be adequate for tracking.

8.3 Sensable Phantom Omni Haptic Device

When a human touches an object in real world, a tactile stimulus is generated due to the forces that are generated between the object and the point of contact. The stimulus signal is transmitted to the brain via the nervous system. Then the transmitted signal is interpreted appropriately by the brain. This leads to the haptic perception. Then necessary reaction signals are generated accordingly by the brain and transmitted to the motor system. The reaction signals are transformed into an action by the motor system of the human.

Similarly, haptic devices aim to simulate the tactile stimulus generated when a contact occurs between the human body and an object in virtual or real teleoperated environment. The ultimate goal of the device is to make the human user perceive as if he or she is really touching an object in the virtual or teleoperated environment although the object is in fact virtual or far away. A haptic device performs this simulation by applying appropriate forces along the appropriate linear or radial axes. Thus a physical resistance is applied to the human holding the end-effector of the haptic device. The strength and direction of the resistance depend on the material composition of the object being touched. A good coverage of haptics can be found in (Srinivasan, n.d.).

Sensable Phantom Omni has recently arrived at the laboratory by the time this thesis was written. It has 6 degrees of freedom positional sensing with IEEE 1394a interface for host computer connection. For further technical details the researcher

may refer to (SensAble Technologies, Inc., 2008). Figure 8.7 shows Sensable Phantom Omni haptic device in the laboratory.

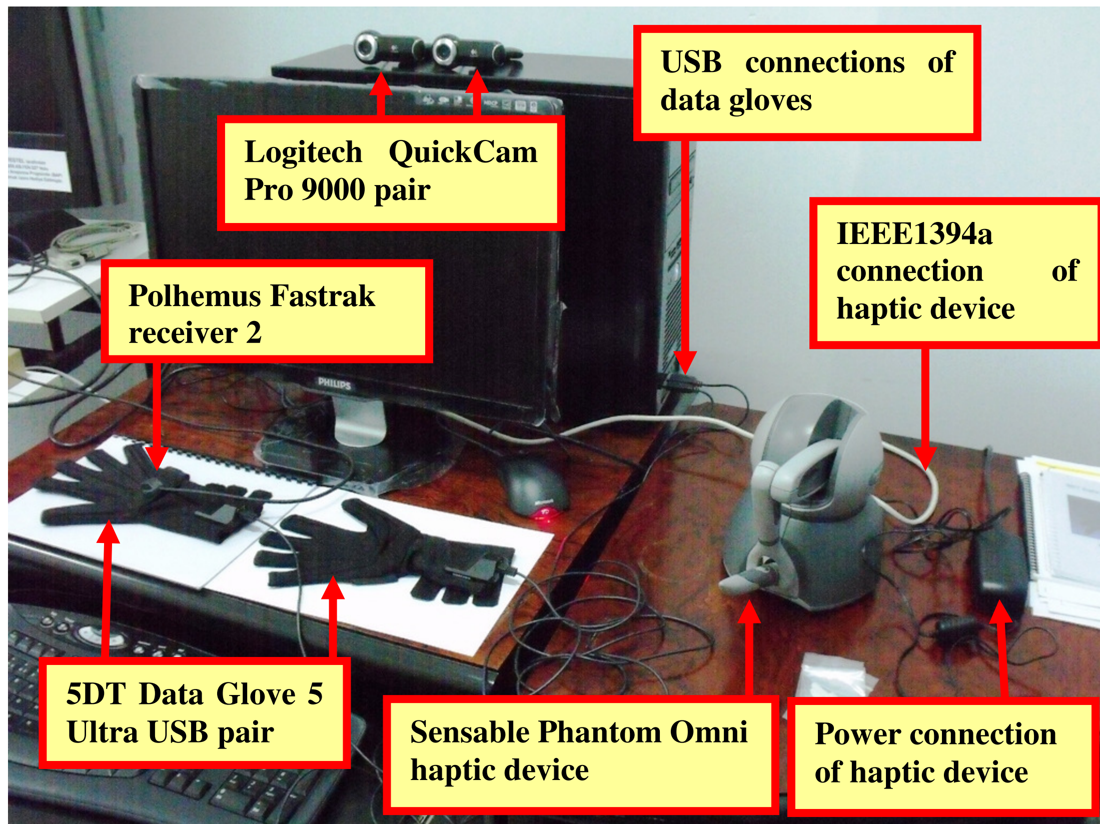


Figure 8.7 The haptic device, the data glove pair with the receiver 2 and the webcam pair.

8.4 5DT Data Glove 5 Ultra USB Left and Right Pairs

5DT Data Glove 5 Ultra USB is a device for hand motion data capture. The laboratory has both left and right data glove pairs. The data gloves have a USB interface for host computer connection. Each data glove has five bend sensors to measure the flexure of each finger. The flexure is measured as an average of first joint and second joint on each finger. Each bend sensor analog output is digitized with 12 bit analog digital converter. All the bend sensors are sampled at least at 60 Hz. The figure 8.7 shows each pair of data gloves that the laboratory has.

The data gloves can be integrated to a development environment via its bundled software development kit. The data acquired from the sensors can either be raw or

scaled data. A total of 16 simple gestures can be defined for a single hand via the software development kit. A calibration procedure is necessary. Auto-calibration procedure is the simplest way to go. Technically, in auto-calibration mode, the raw value x acquired from the bend sensor is compared to the current boundary values $x_{\min}$ and $x_{\max}$. If the read raw value is outside this inclusive interval, the boundary values are updated. The corresponding sensor data observed from the application is calculated as follows;

$$x_{out} = \frac{x - x_{\min}}{x_{\max} - x_{\min}} MaxVal \quad , \; MaxVal \text{ is set by the user.}$$

For further information, the researcher should refer to (Fifth Dimension Technologies [5DT], 2004a).

8.5 5DT HMD 800 – 26 3-D Head Mounted Display

5DT HMD 800 – 26 3-D head mounted display is a stereoscopic SVGA device that has 26° viewing angle and 44 inch virtual image size at 2 meters. Each LCD panel of the device can generate 800x600 image plane for each red, green an blue colors resulting in 1.44 million pixels. The device in the laboratory is shown in figure 8.8. For further technical details the researcher may refer to (5DT, 2004b).

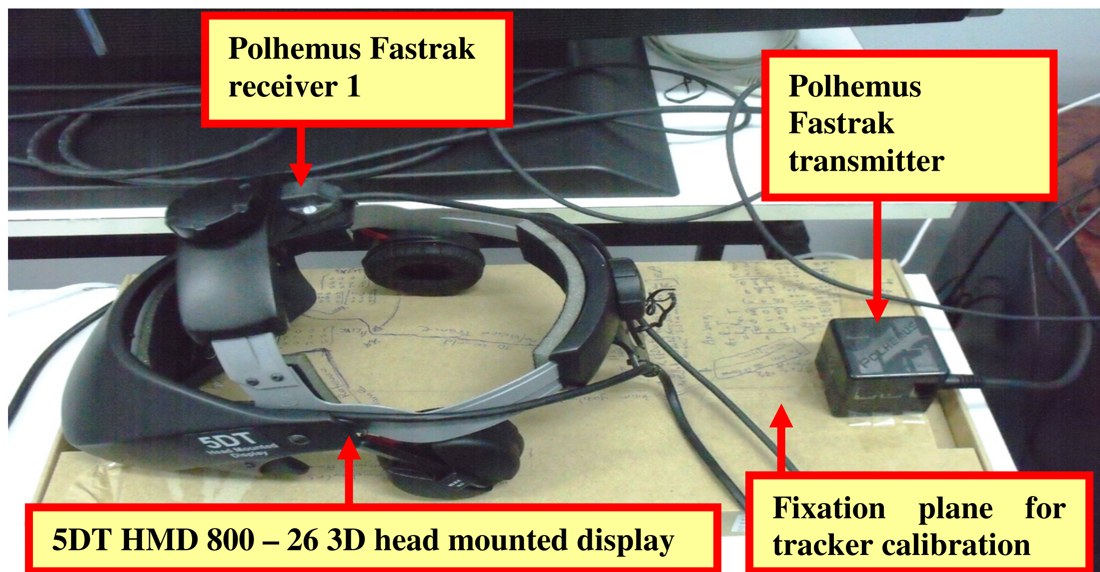


Figure 8.8 5DT HMD 800 – 26 3D head mounted display, mounted tracking system receiver and transmitter.

8.6 Logitech QuickCam Pro 9000 Webcams

The laboratory has two Logitech QuickCam Pro 9000 webcams for augmented reality and video processing applications. The webcams have Carl Zeiss optics with 2-MP HD sensor that can produce high definition video up to (1600 x 1200). The devices in the laboratory are shown in the figure 8.7.

8.7 ATI X1550 and NVIDIA GeForce GTX295

ATI X1550 is an ATI RV515 core based graphics card with 105 million transistors on 90nm fabrication process. It has 4 fragment (pixel) shaders, 2 vertex shaders, 4 raster operation pipelines (ROPs) and 4 texture units. A support for DirectX 9.0c, OpenGL 2.0, multiple render target and render to vertex buffer is provided. It supports Shader Model 3.0 programmable vertex and fragment shaders in hardware and up to 128 simultaneous pixel thread. The card has PCIe x16 bus interface for host computer communication. For more technical details, the researcher should refer to (Advanced Micro Devices, Inc. [AMD], 2010) and (Wikipedia, 2010s).

NVIDIA GeForce GTX 295 is a 2nd generation NVIDIA® Unified Architecture and 10th generation NVIDIA GeForce series. It has two graphics processing unit. Each graphics processing unit has 1.4 billion transistors on 55 nm fabrication process. It supports NVIDIA® CUDA™ technology for general purpose computing. Each graphics processing unit has 240 CUDA cores resulting in total of 480 CUDA cores. It supports programmable graphics pipeline and hence programmable vertex, geometry and fragment processors in hardware. The card supports NVIDIA® PhysX™ for graphics processing unit based physics tasks for complex rigid body, soft body, particle system, character control, ray-cast and articulated vehicle dynamics, volumetric fluid simulation, cloths and volumetric force fields. NVIDIA® PhysX™ is also multithreaded, multi platform and physics processing unit enabled. Additionally a support for NVIDIA 3D Vision is provided for stereoscopic 3-D applications. The card has NVIDIA Quad SLI® support for multi-graphics card

utilization, SLI multi monitor support and GigaThread^{TM} technology that is a massively multi threaded architecture for running thousands of independent threads simultaneously. A support for Direct3D 10.0, OpenGL 3.3 and Shader Model 4.0 programmable vertex and fragment shaders in hardware is provided. The card uses PCIe x16 2.0 for communication with the host computer. For more technical information, the researcher should refer to (NVIDIA, 2010) and (Wikipedia, 2010t).

CHAPTER NINE

SOFTWARE DEVELOPMENT AND HARDWARE INTEGRATION RESULTS

In this chapter, all the software implementation results from the beginning of the thesis to the end will be presented.

9.1 Software Development Tools Used During the Thesis Work

Microsoft Visual C++ 2005 Development Environment was used throughout the thesis work. Mathworks MATLAB 2008a was used for numerical verification of the methods implemented. The configuration graphical user interface (GUI) of the software was developed using Qt GUI Development Kit which is platform independent. Qt has its own meta object compiler and thus cannot be compiled directly with Microsoft Visual C++ or other compilers. For technical details, software design patterns and implementation considerations regarding Qt, the researcher should refer to (Blanchette & Summerfield, 2008). Ogre3D was used as a real time graphics rendering engine. For the technical and implementation considerations, the researcher should refer to (Junker, 2006). Prior to Ogre3D, OpenSceneGraph was the choice as a graphics rendering engine. It is used during the augmented reality (AR) application development together with osgART. osgART is the ARToolkit plug-in for OpenSceneGraph. INTEL OpenCV was also used for the development of AR application. For more information about INTEL OpenCV, the researcher should refer to (Bradsky & Kaehler, 2008). Ogre3D was the choice over OpenSceneGraph during the virtual reality (VR) application development because of its flexibility, ease of integration, support of shaders and most importantly the availability of learning resources. The researcher interested in OpenSceneGraph should refer to (Martz, 2007). Bullet was used as a physics engine during the thesis work. A good learning resource for Bullet is (Coumans, 2009). For code development tests accomplished using NVIDIA CUDA were compiled using NVIDIA C Compiler that can be executed under Microsoft Visual C++ 2005. CUDA

API 2.2 was used during these tests. Microsoft DirectX 9.0c and Cg 2.0 were used during the development. Apart from the actual software development, in order to practice GPU based physics rendering and to test whether NVIDIA PhysX could be useful for the development target or not, NVIDIA PhysX SDK 2.8.1 was studied theoretically and practically. For details of NVIDIA PhysX, the researcher should refer to (NVIDIA, 2008).

9.2 Implementations Completed during Augmented Reality (AR) Application Research

In the beginning of the thesis work period, the necessary tracking and data glove equipment could not be purchased. That meant a slow down for the laboratory establishment process and also for the application development during that period. Because, the user would not be able to interact with the virtual environment without the tracker and the data gloves. Then for the period, the main target was defined to be a development of a virtual environment in which the user can pan in the environment with the head movements and control a virtual graphical user interface tablet for several actions. At the result of this period, a virtual environment in which an user can control a virtual graphical user interface tablet via real time video tracking was developed. The user head could successfully tracked for movements that are not very fast so that linearity conditions satisfy. The Z distance that is the distance of the user from the camera could not be calculated at that moment. To calculate the Z distance, either a stereo rig should have been setup, or the affine relations of feature points between two consecutive frames should be tracked. The real time tracking study was not carried so far as the main goal of the thesis work was not computer vision. The necessary theoretical background was given in the previous chapters. The implementation results will be given in figure 9.1. For necessary camera calibration, ARToolkit was used. For 2-D head tracking was performed using INTEL OpenCV Library. For development of the virtual environment OpenSceneGraph was used. Finally, the integration of OpenSceneGraph with ARToolkit was established via osgART plug-in.

Calibration result

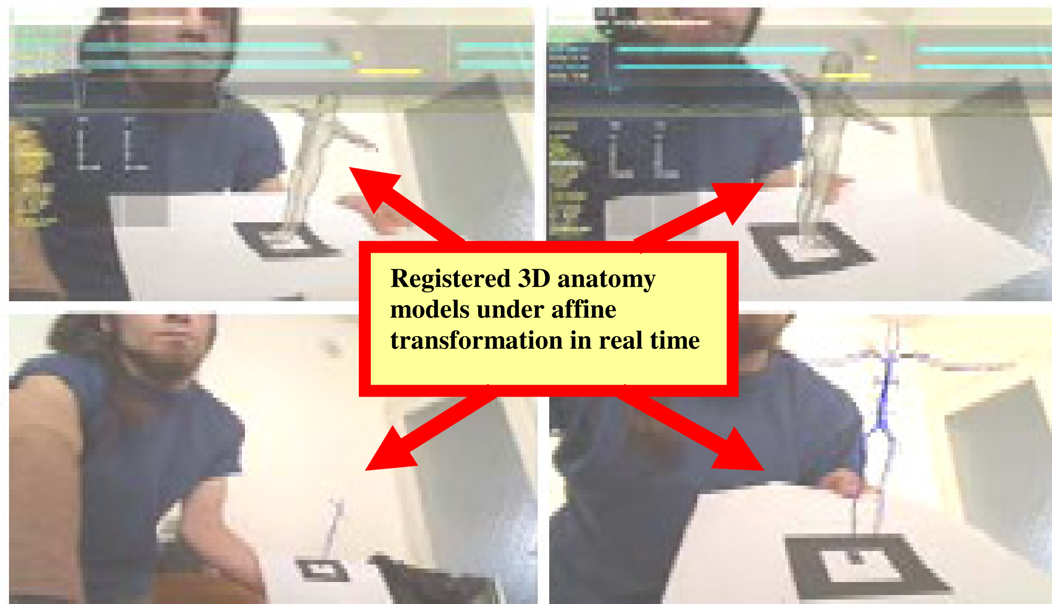
(a)

Registered 3D virtual cube in real time

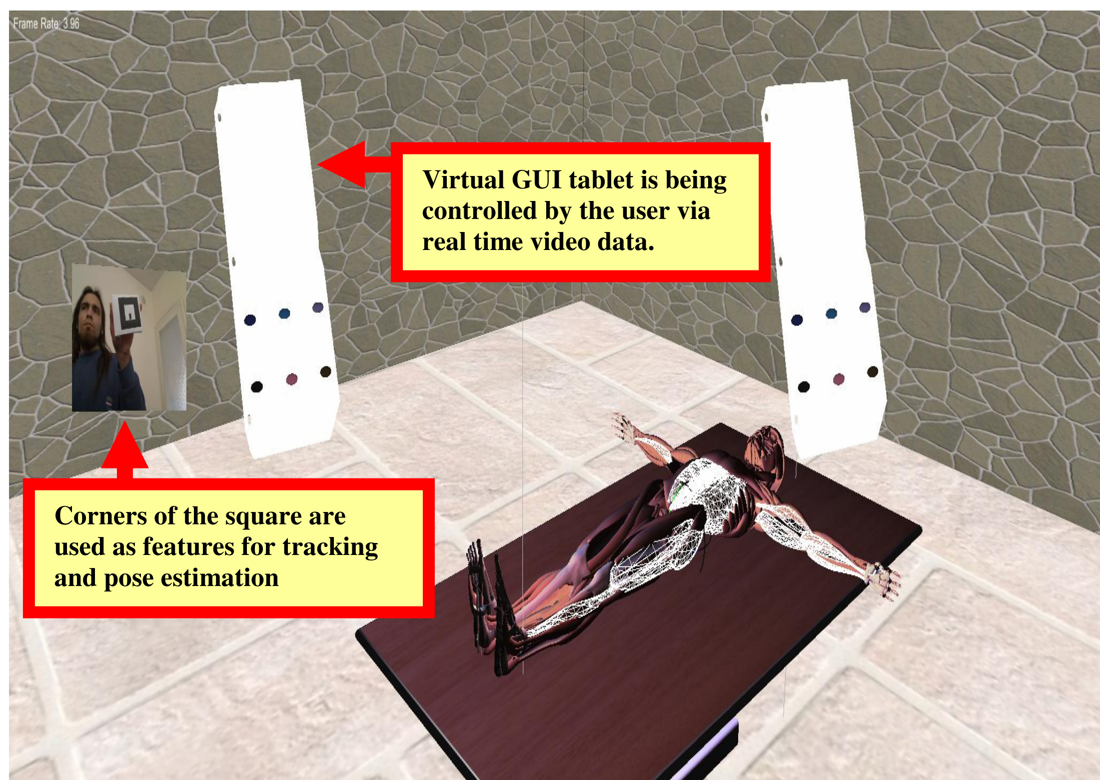
Registered 3D virtual cube real time

(b)

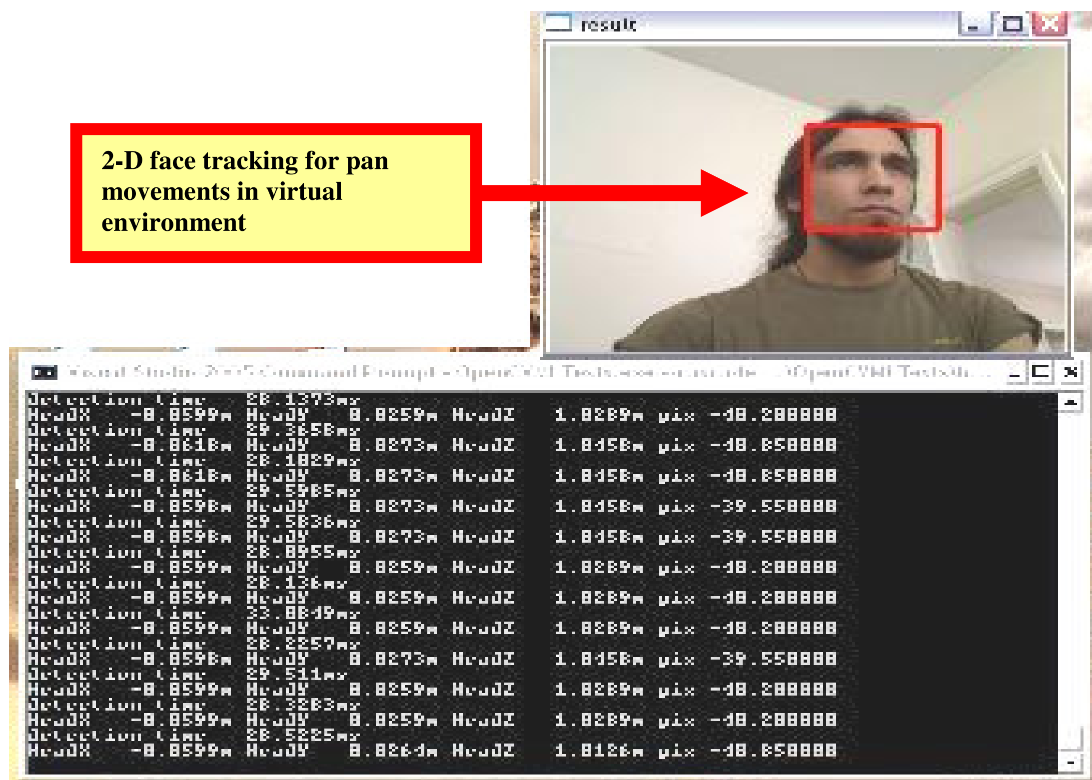
(c)



(d)



(e)



(f)

Figure 9.1 (a) Camera calibration. (b), (c) Initial implementations for registering 3-D models with the video in real time. (d) Application of affine transformations to 3-D model in real time by tracking the corner features of the square on the planar paper in the video data. (e) User interaction with the virtual environment and the control of a graphical user interface tablet via tracking the same features in (d); white wireframe overlays indicate the model part selection. (f) Face tracking for pan movements in the virtual environment.

The face tracking implementation remained as a separate module, because at that time the necessary equipments arrived.

9.3 Development Result of the Immersive Interactive Virtual Environment for Collaborative Anatomy Inspections in Medical Education

The final development results are presented in this part. The software layers and scene graph hierarchy are developed as given in figures 9.2 and 9.3 respectively. Then the views from the real time interactive environment will be presented. Following this part, the preliminary implementation experiences and study results leading to the final software development will be presented.

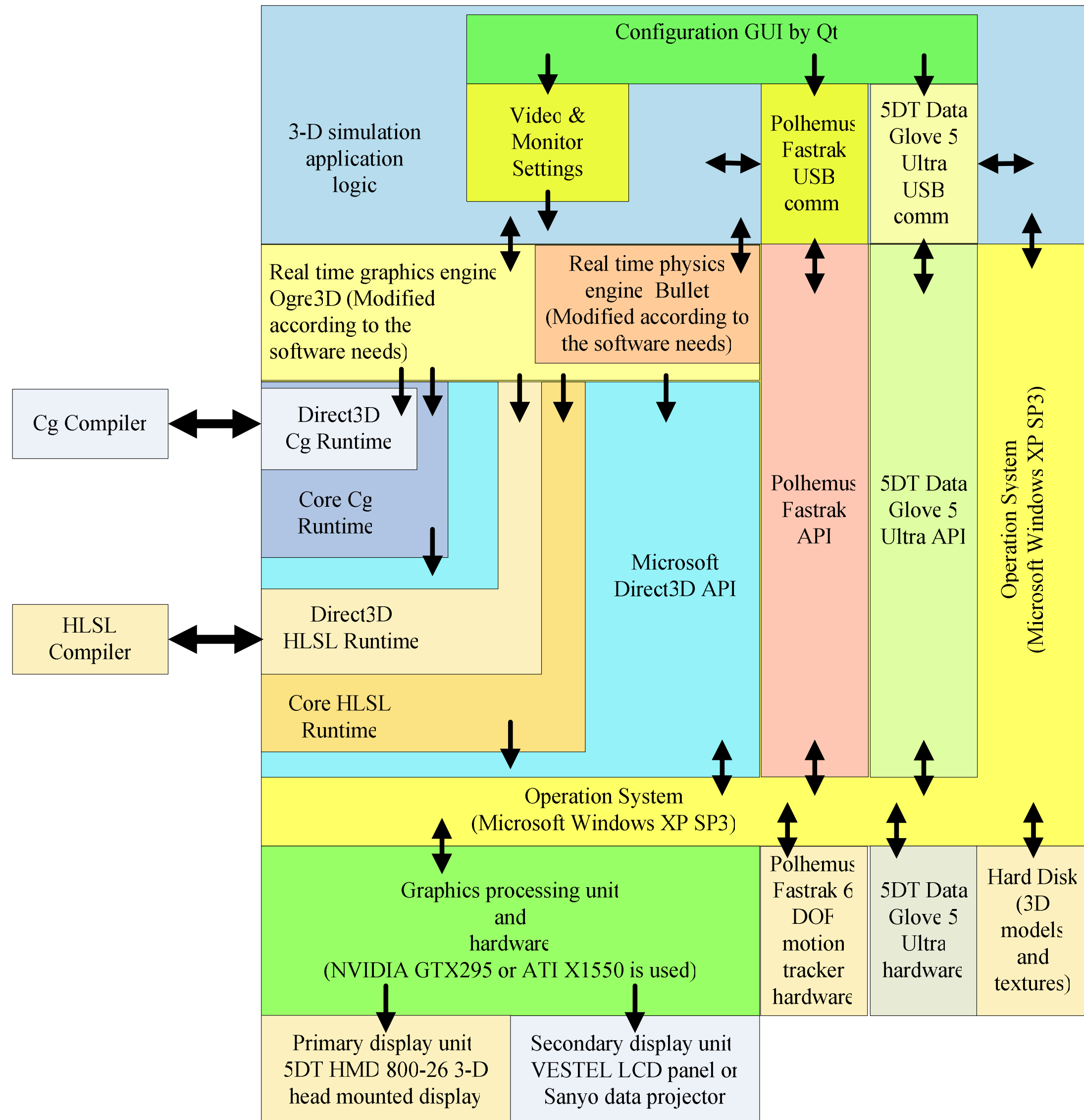


Figure 9.2 Functional layers of the developed software.

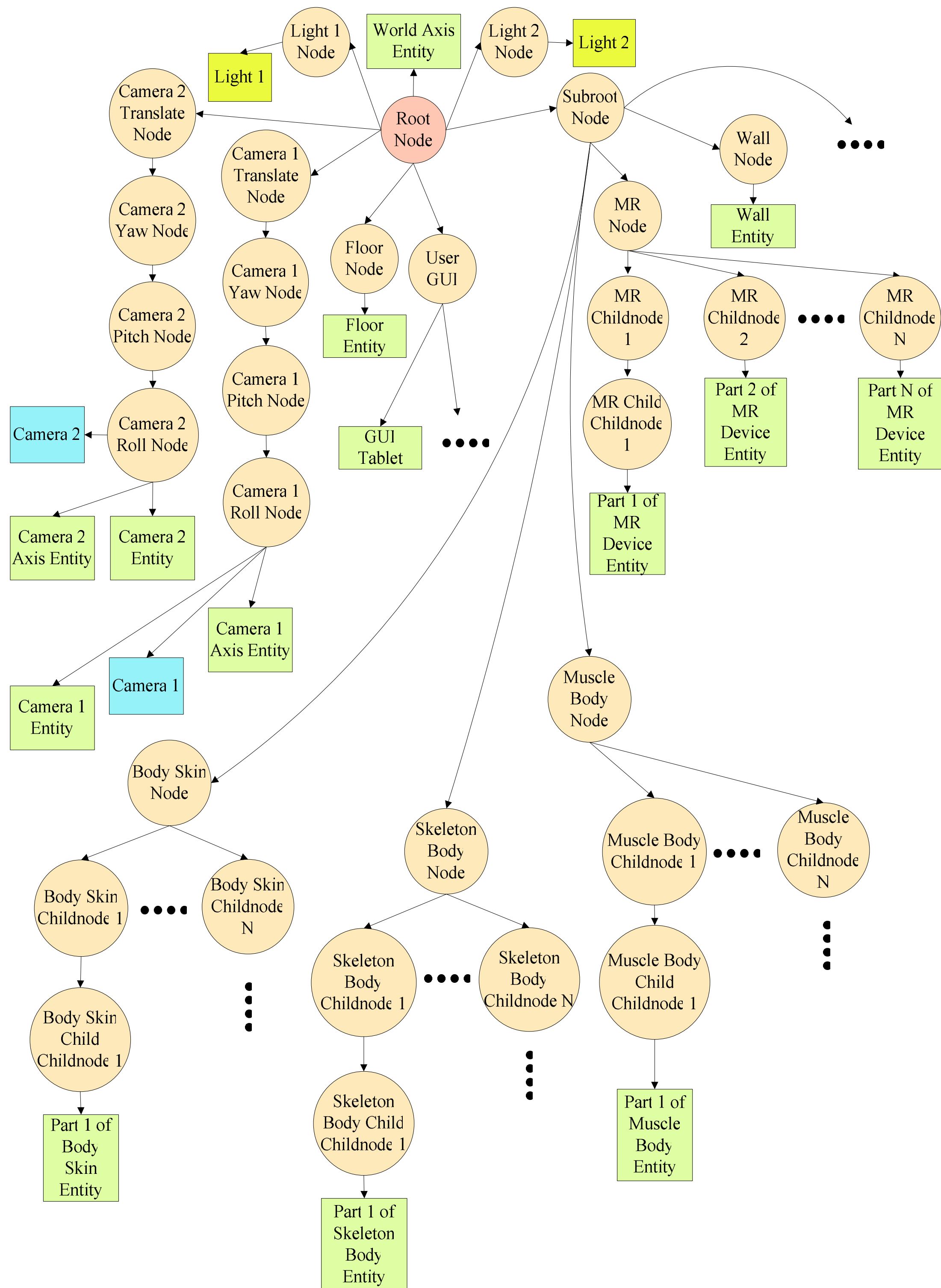


Figure 9.3 Directed acyclic graph representing a part of the virtual environment developed. Only a representative portion of the whole graph is given because of the page size constraint. Black dots represent remaining node connections in the graph.

(a)

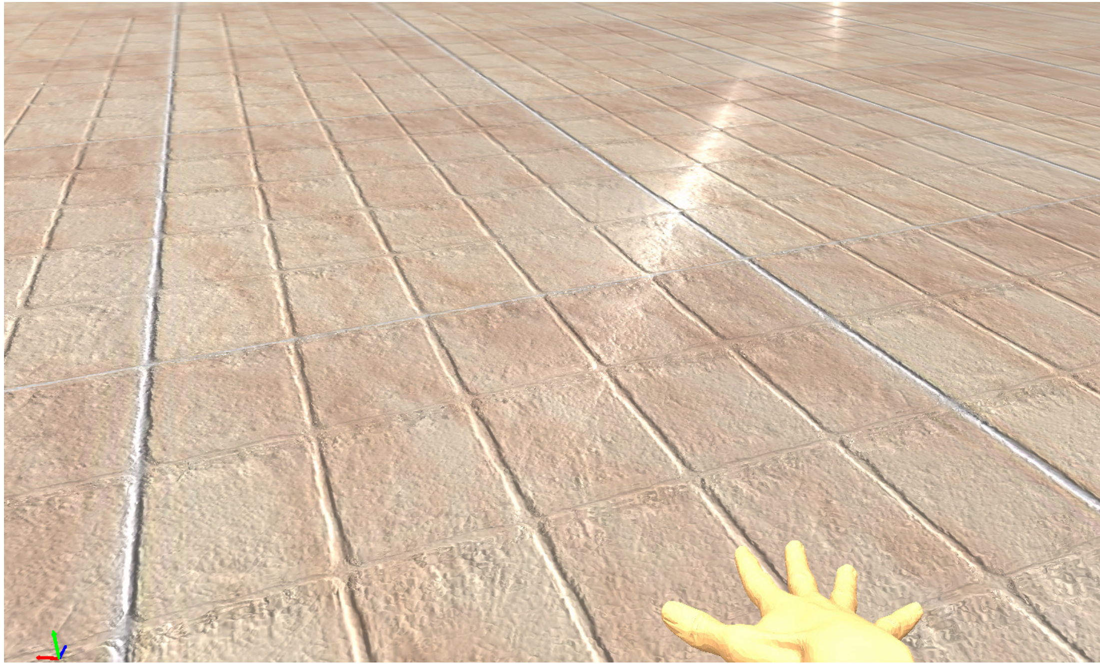
(b)

(c)

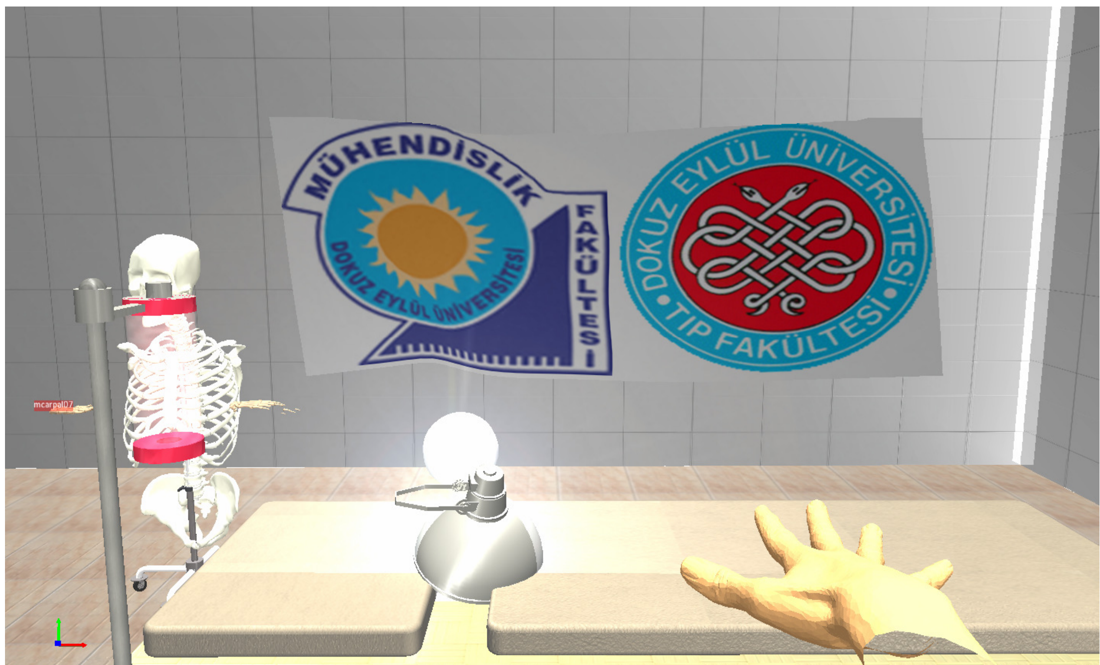
(d)

(e)

Figure 9.4 (a) and (b) represent two views from the virtual environment in which two users are present. The uses are indicated by blue pyramids together with local coordinate axis. Each user has a local coordinate axis in the bottom left of the view to see his or her orientation in the virtual environment (See section 5.2 for gimbal lock problem). Object manipulations are done by hand. The frame rate is 18 fps at average. The texturing and lightning are performed using programmable graphics pipeline by using Cg language (See section 5.4). (c) and (d) represent the rigged and skinned hand deforming in accordance with the user's hand gestures (See section 5.5). A light shaft is rendered at the position of the medical light aimed towards the human body (See section 5.3). (e) Represents the deformable cloths and tissues. The green info overlay at the lower left corner of the screen informs the user about the penetration depth, collision contact point and contact normal and the applied impulse when collision occurs between the user's hand and between any virtual object (See chapter 6 and sections 6.6.3.1, 6.6.3.2). White wireframes represent the collision models used for related render models.



(a)

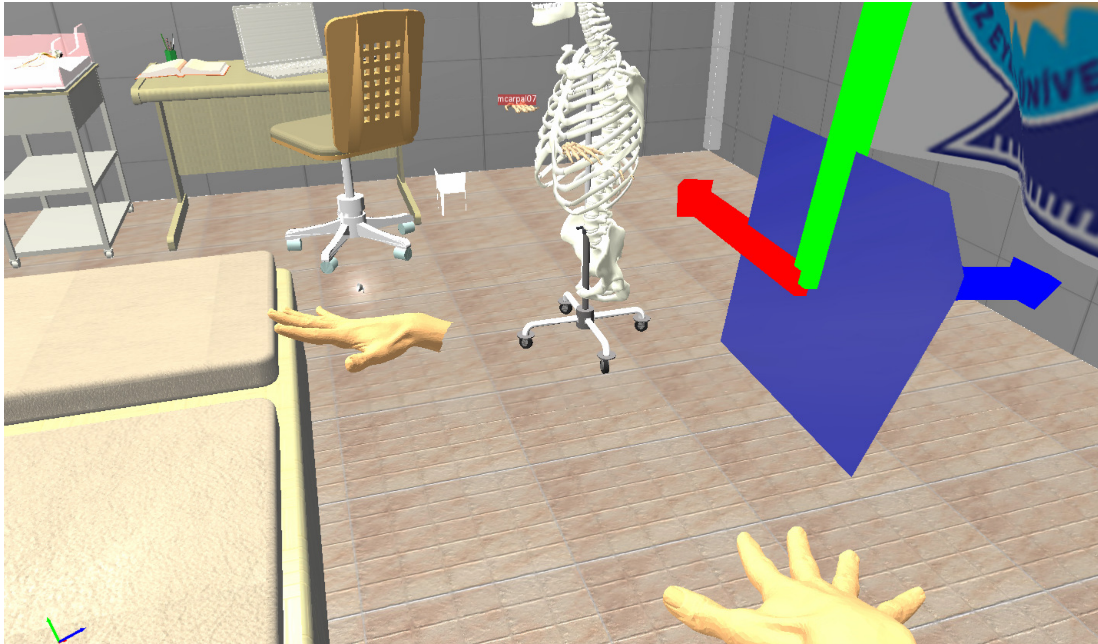


(b)

Figure 9.5 (a) A close look at bump mapping with parallax offset technique used for texturing the environment. The model-view matrix and light parameters are continuously passed to the GPU as the vertex code input arguments to update the lighting effects such as reflection power and its direction. This technique is applied by using programmable graphics pipeline with the help of vertex processor and fragment processor codes written in Cg (See section 5.4). (b) A 2-D dynamic mass-spring topology namely – the cloth simulation - on which two logos present. This dynamic topology is used

to study the tradeoffs of numerical integration methods between their stability and accuracy in conditions where time steps are changed and to implement integration methods such as explicit Euler integration, second order Runge-Kutta integration, fourth order Runge-Kutta integration and Verlet integration are considered. The other particular importance of that scene is that, one of the first collision detections are implement by using the white sphere standing in front of the 2-D mass-spring topology. The collision between the sphere and the 2-D mass-spring topology is solved by implementing fitting a sphere around the mesh of the white sphere and detecting collisions between this sphere and the vertices of the 2-D mass-spring topology to which masses are bound. The governing differential equation of the 2-D mass spring topology, applied forces and collision scheme are independent from the physics properties of rest of the virtual environment (See chapter 6).

(a)



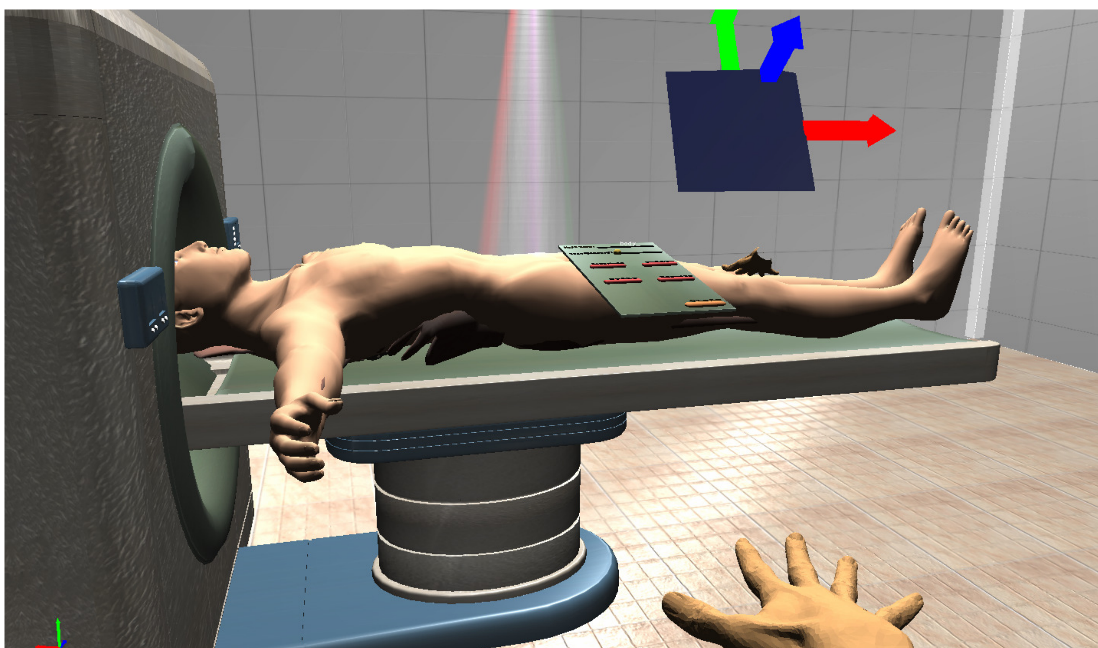
(b)

Figure 9.6 Interaction is possible with the 3-D models in the virtual environment. (a) One of the users has taken the light standing on the bed by touching and holding with his or her hand. (b) The other user is looking at the user holding the light. The light can be notices on the hand of the user (See chapter 6).

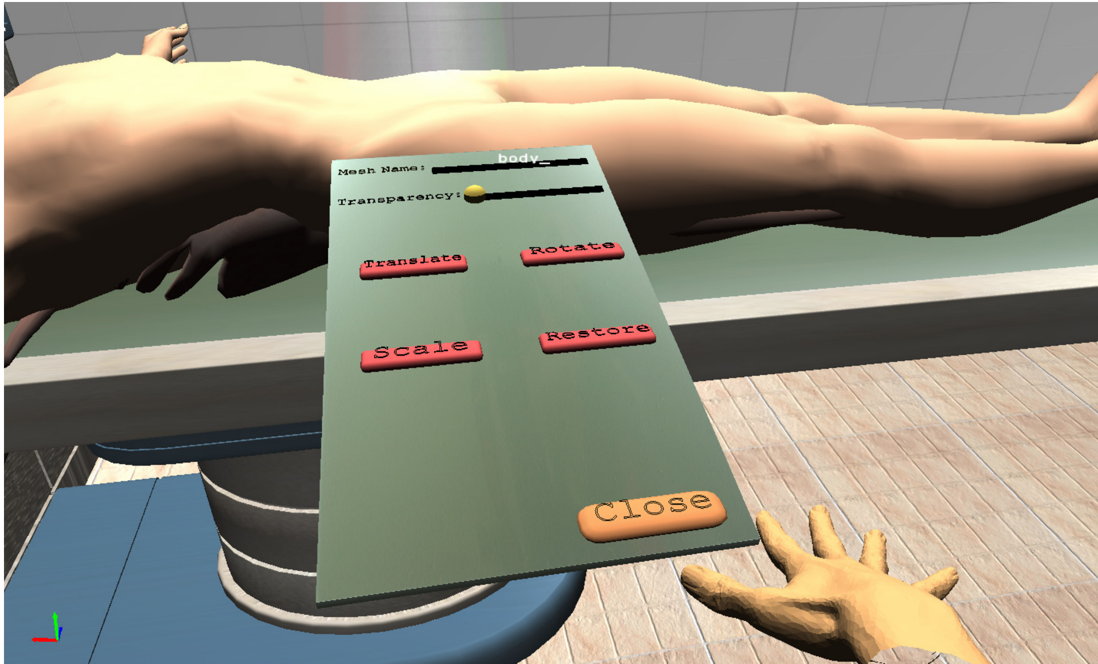
(a)

(b)

Figure 9.7 (a) One of the users is looking at the other user who has left the light to the ground. (b) The view of the user who has left the light near the wall. Leaving the 3-D model can be done by colliding it with the bed or according to the bending data of the fingers of the user retrieved from the data glove (See chapter 5 and chapter 6).

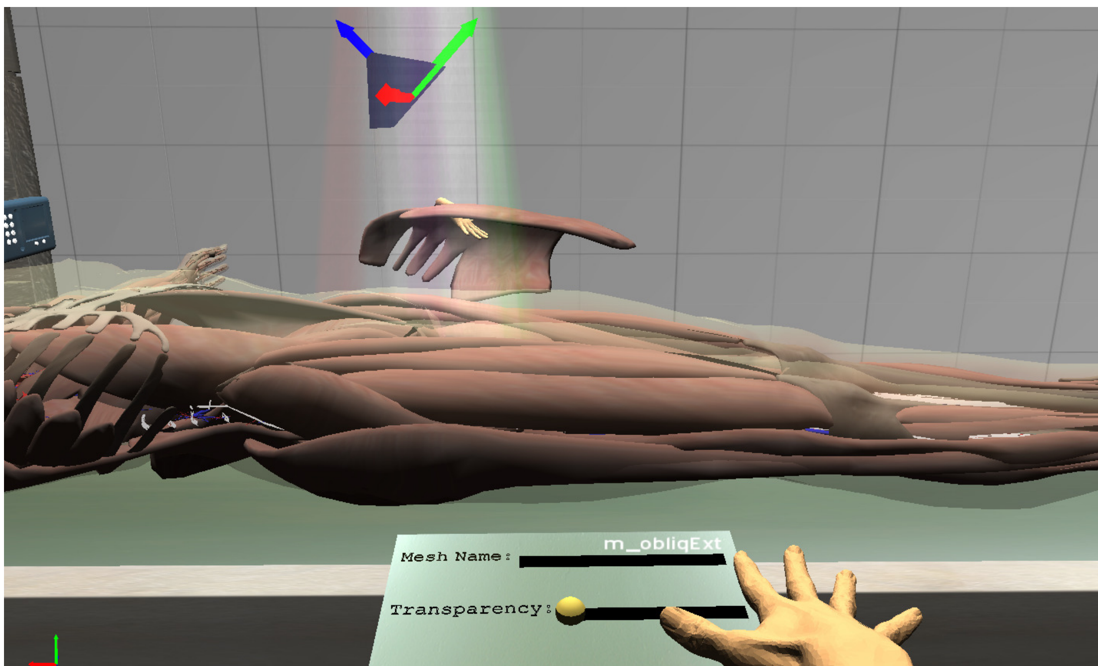


(a)

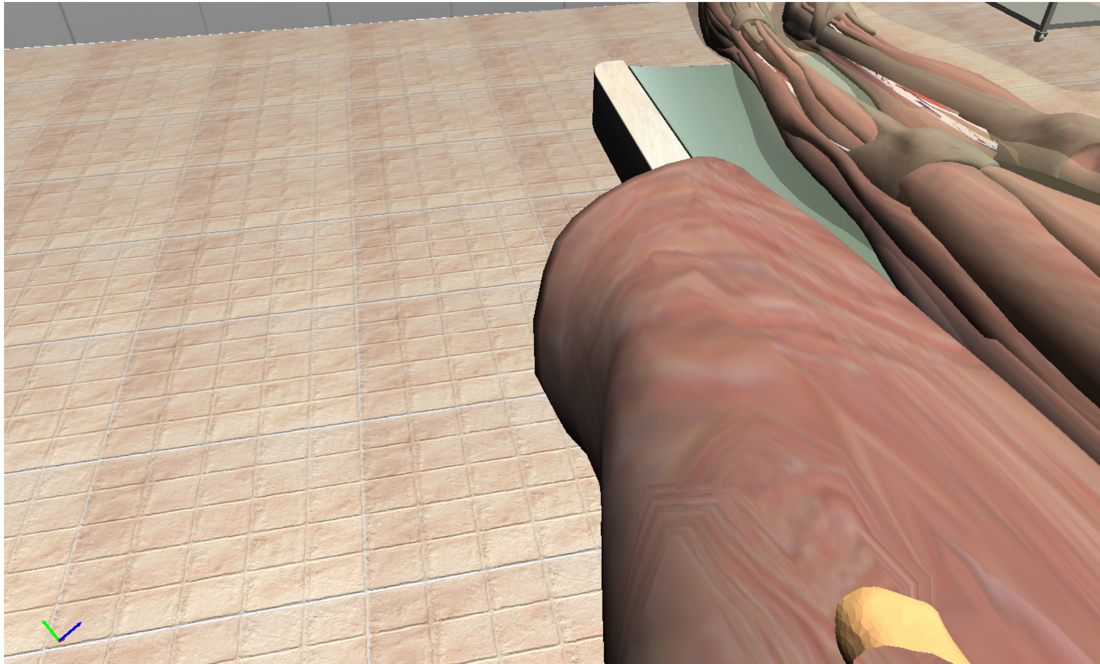


(b)

Figure 9.8 A user interface with several controls is displayed when a user wants to manipulate the parts of the anatomy model. (a) and (b) presents the views of two users. Lights are dimmed if wanted, during inspection (See section 5.3 for light shafts rendering).

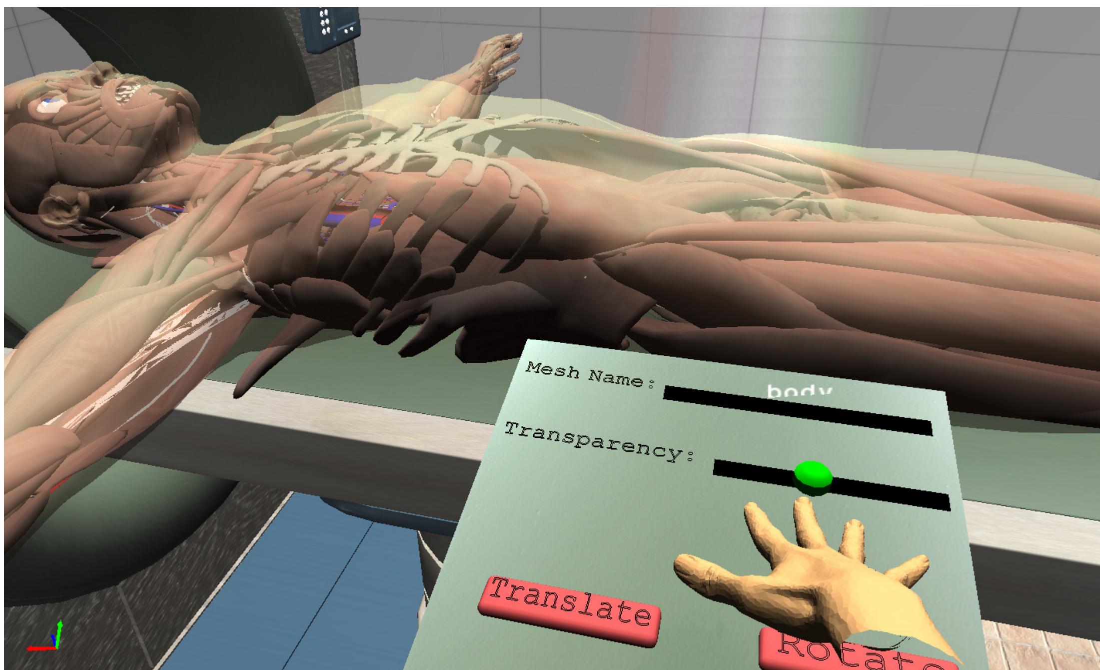


(a)

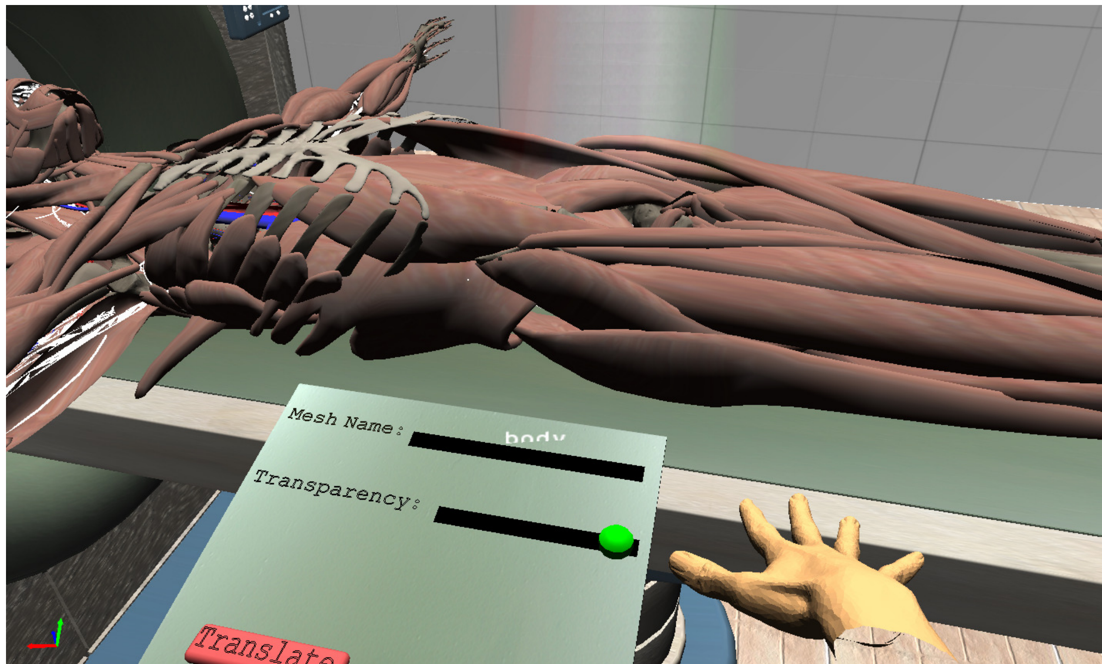


(b)

Figure 9.9 (a) One of the users looks at the user interface. Notice that information can be displayed on the user interface. (b) A user in the same scene holding the part of an anatomy model. A light shaft is located over the model part of interest (See section 5.3 for light shafts rendering).

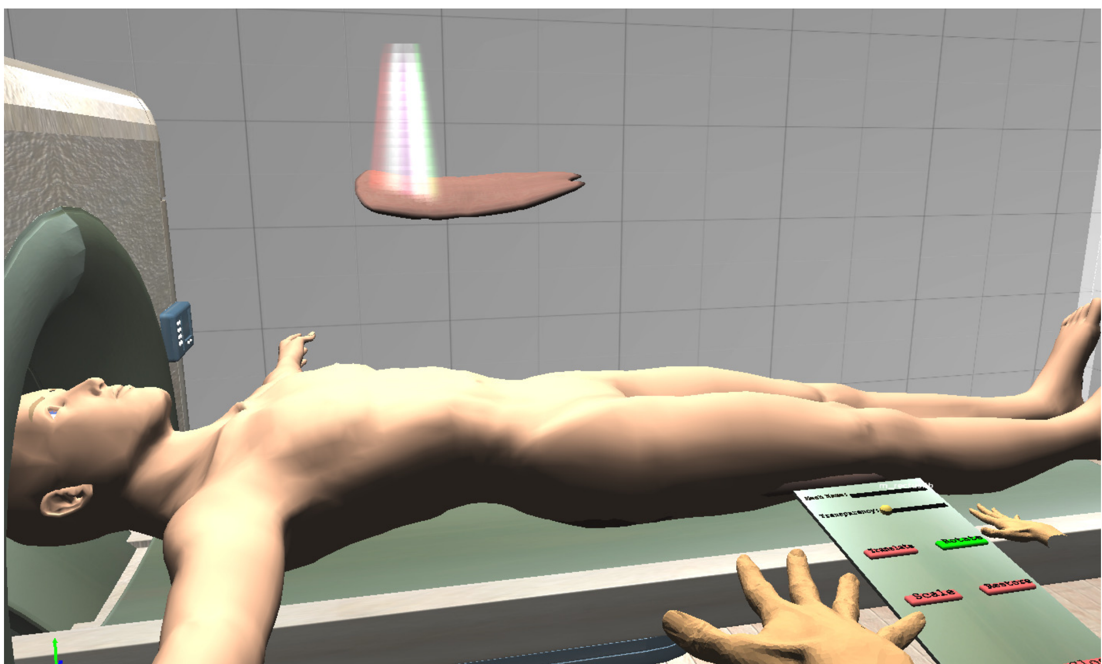


(a)

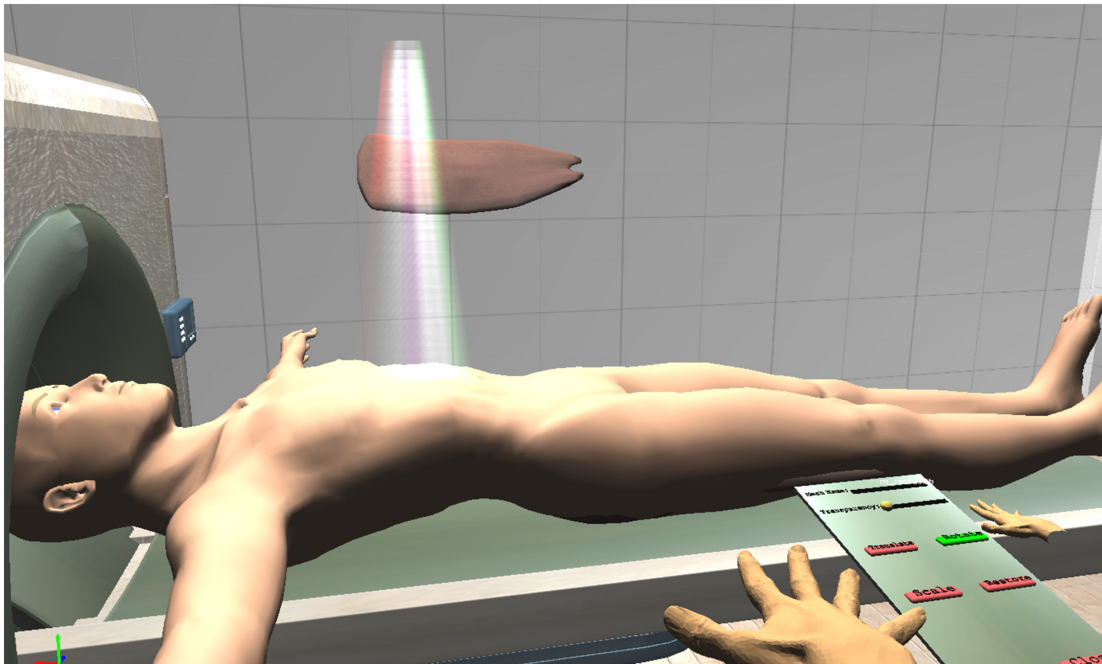


(b)

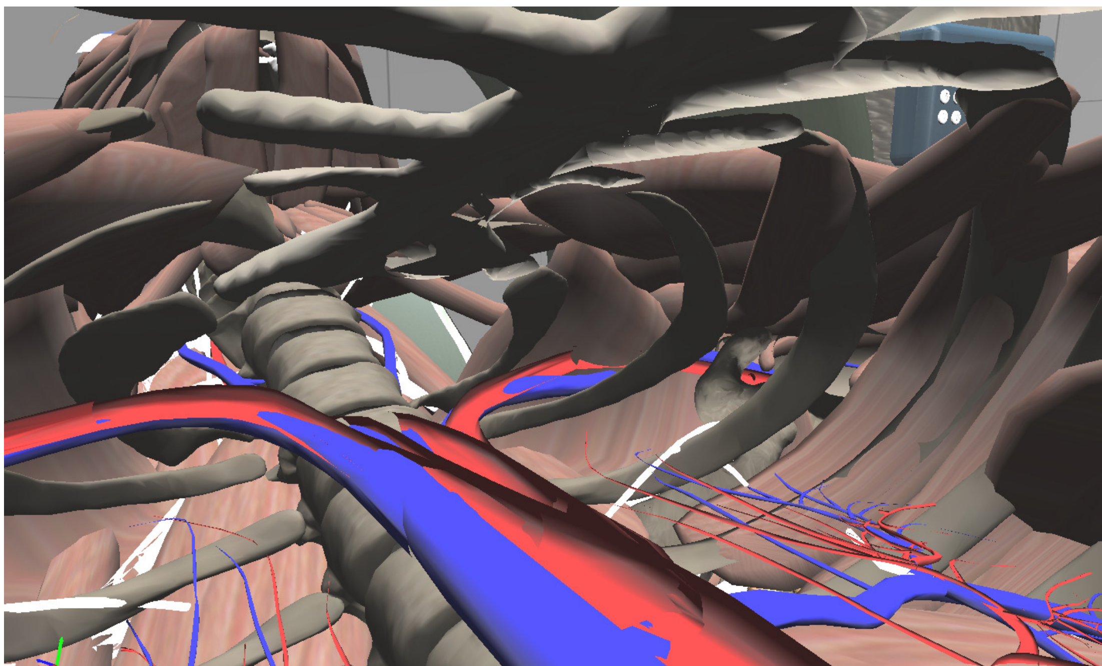
Figure 9.10 Several properties of the 3-D models can be controlled via the user interface. Constraints defined for the controls of the user interface define their behaviour. When a user touches or grabs the user control, the color of the related control goes to green. When the user leaves the user control its color returns to the original color (See chapter 6 for the collision detection and constraint solution methods). These are presented in (a) and (b). Anatomic parts can be attached, detached and manipulated freely by the user via the virtual hand.



(a)

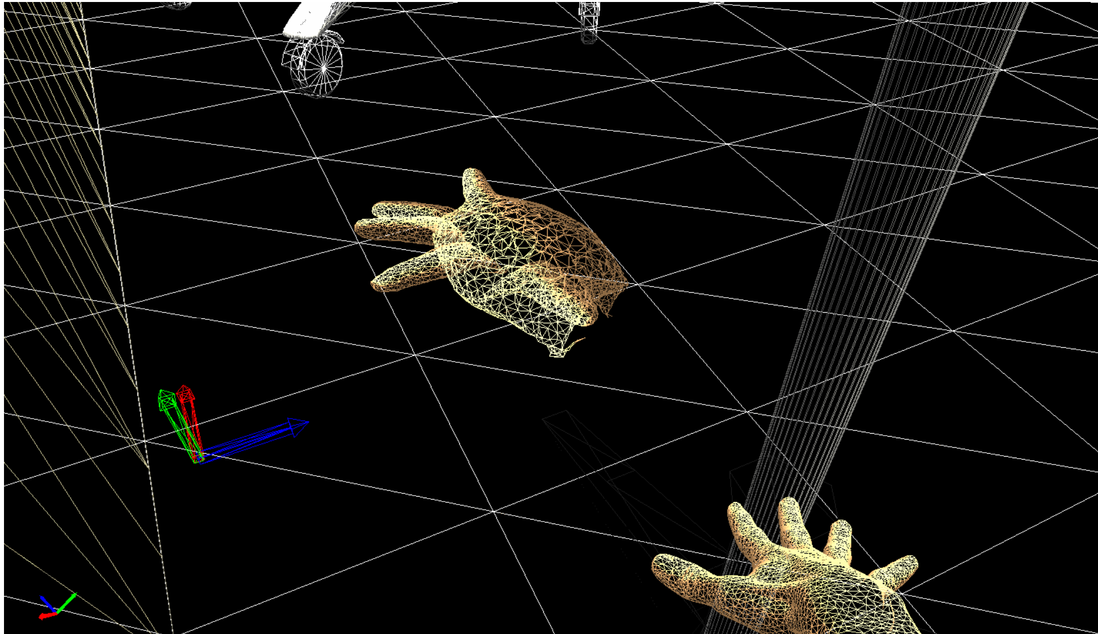


(b)

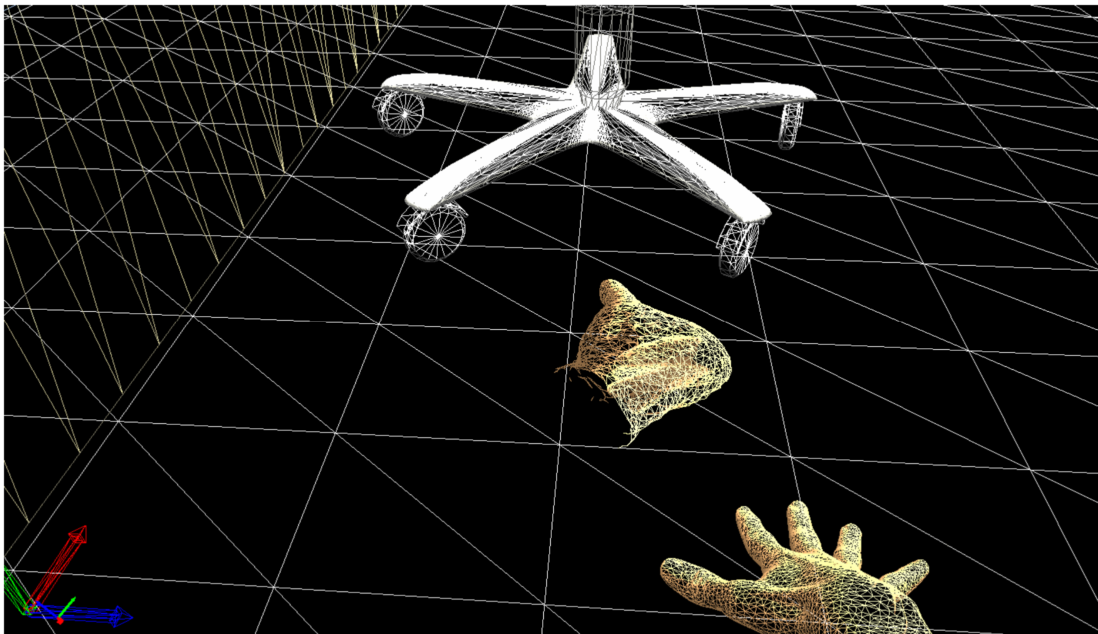


(c)

Figure 9.11 One of the users decides the inspect one of the parts. Any affine transform can be applied for this purpose. A light shaft is seen over the model of interest. These are presented in (a) and (b) from the views of two users. In (c), a user inspects inside of the anatomy models and if wants interacts with the anatomy model parts.

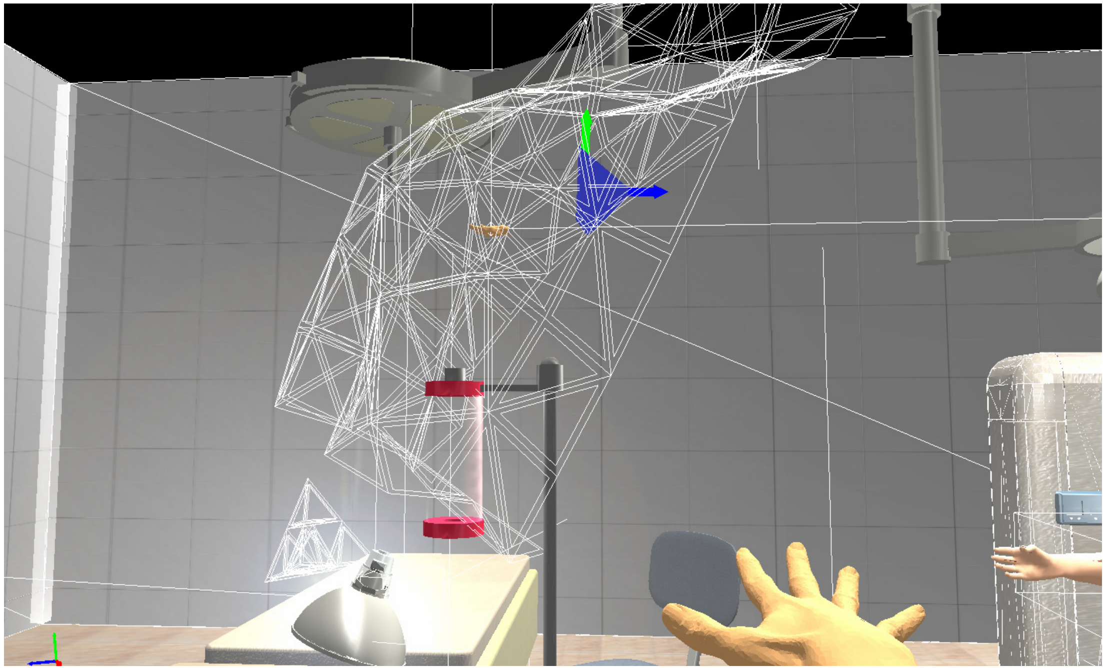


(a)

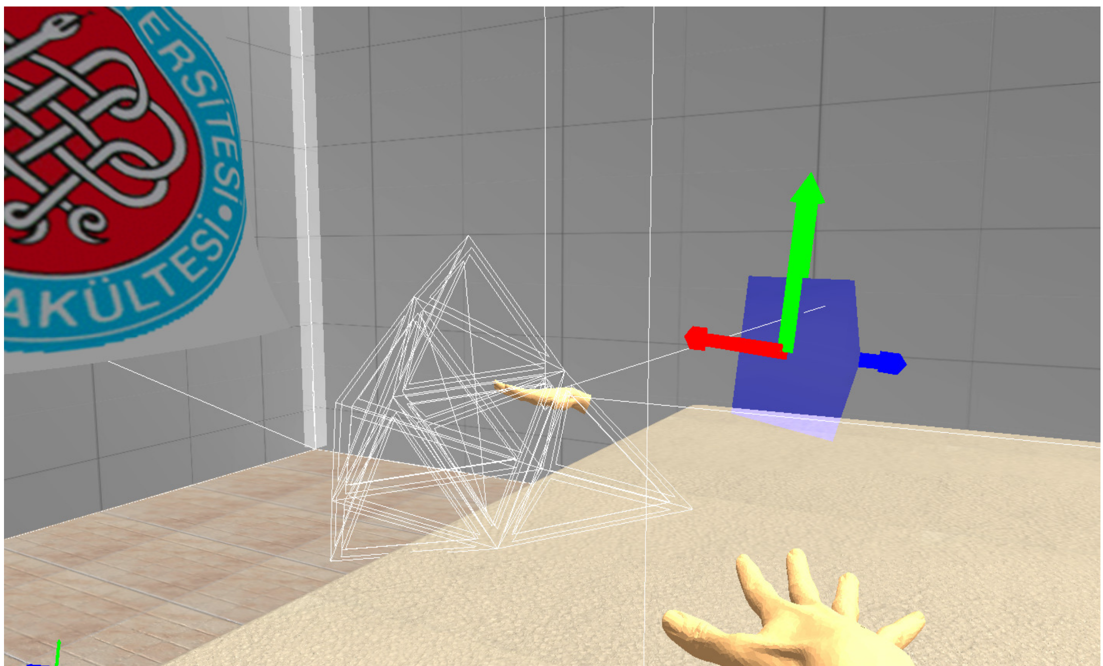


(b)

Figure 9.12 Initial tests for using 2-D mass spring systems with volumetric 3-D models. The implementation should be tuned by appropriate spring constants and volume preservation constraints. But the correct approach is to use tetrahedral mass-spring ssystem and solve those models numerically for modeling states of dynamic 3-D topologies i.e deformation, due to the applied force, because using 2-D mass-spring system can capture the surface of the 3-D model with lack of information relating volume of the 3-D model (the hand in this particular case). (a) and (b) presents two states of a deformable volumetric hand tried to be modeled by 2-D mass-spring system. Notice that the applied force is due to the gravity and due to the collisions from the ground. Due to the lack of necessary constraints on the 2-D mass-spring system, hand behaves in an inconsistent manner (See chapter 6).

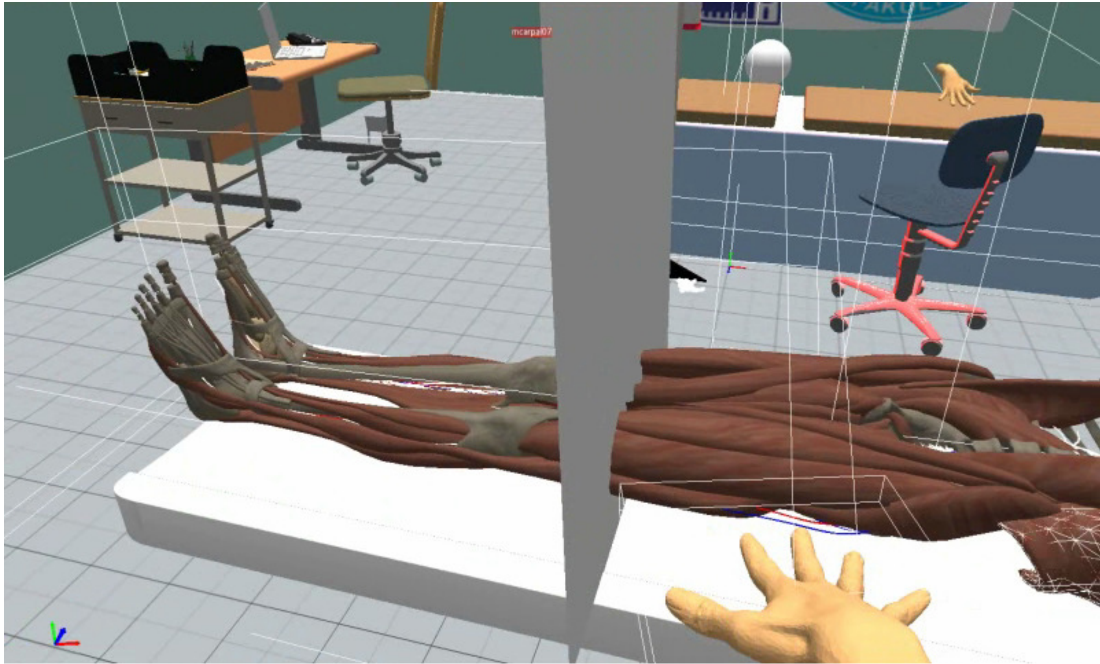


(a)

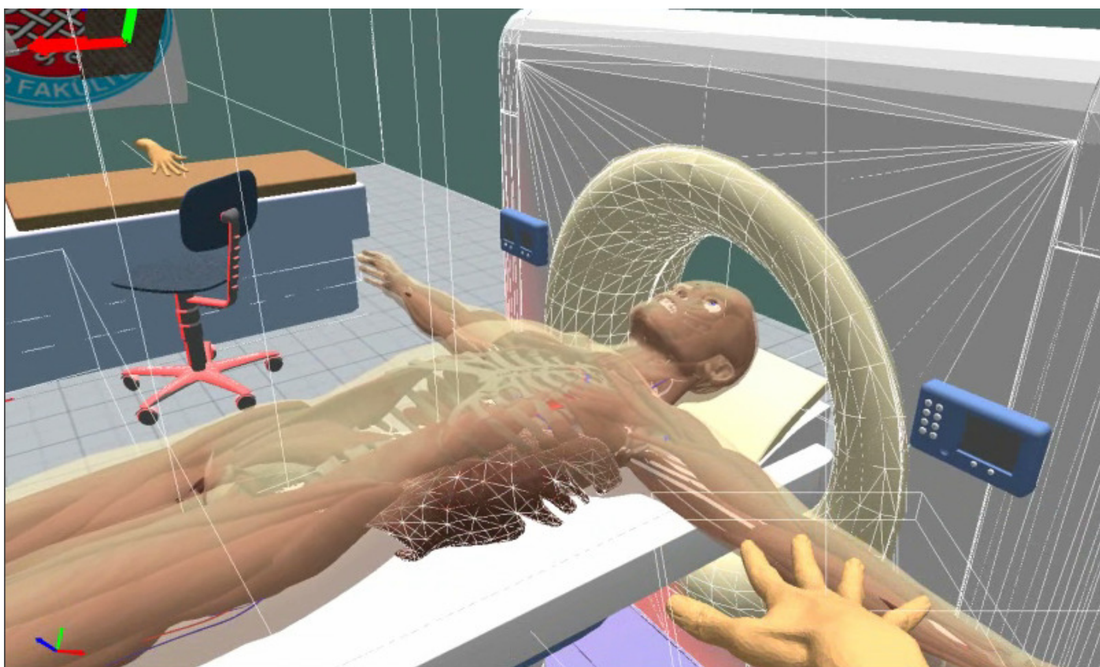


(b)

Figure 9.13 (a) 2-D mesh-spring system for modeling 2-D dynamics and collision. (b) 3-D tetrahedral mesh-spring system for modeling 3-D volumetric dynamics and collision (See chapter 6).



(a)



(b)

Figure 9.14 (a) and (b) presents initial views from the environment prior to programming vertex processors and fragment processors with Cg. White wireframes represent the collision models used for related render models.

Figure 9.15 A person using the system for testing.

9.4 Implementations Completed during Mathematical Elements of Computer Graphics and Real Time Graphics Rendering Research

The time period in which no tracker and data glove exist was also used to get theoretical and practical background on curves and surfaces in 3-D spaces such as Bezier surface, Spline curves and surfaces and their variants, Coons Bicubic surface, etc… and real time graphics rendering. The interested researcher should refer to (Rogers & Adams, 1990) for the theory of mathematical elements of computer graphics. Figure 9.16 presents custom software developed using Visual C++ and without using any graphics API such as Microsoft Direct3D or OpenGL during this period for visualizing and affine transforming several different types of surfaces and curves mentioned above. For theoretical and implementation studies in real time graphics rendering, (Möller & et al., 2008) and (Wright, & et al., 2007) are preferred. Figure 9.17 presents several implementations completed.

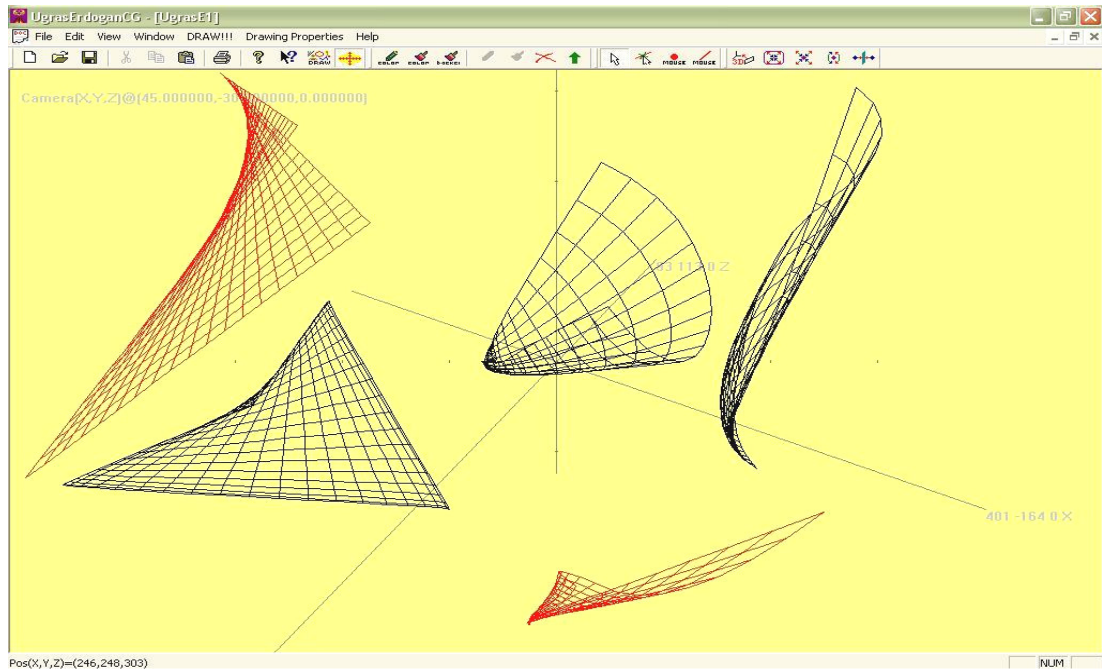
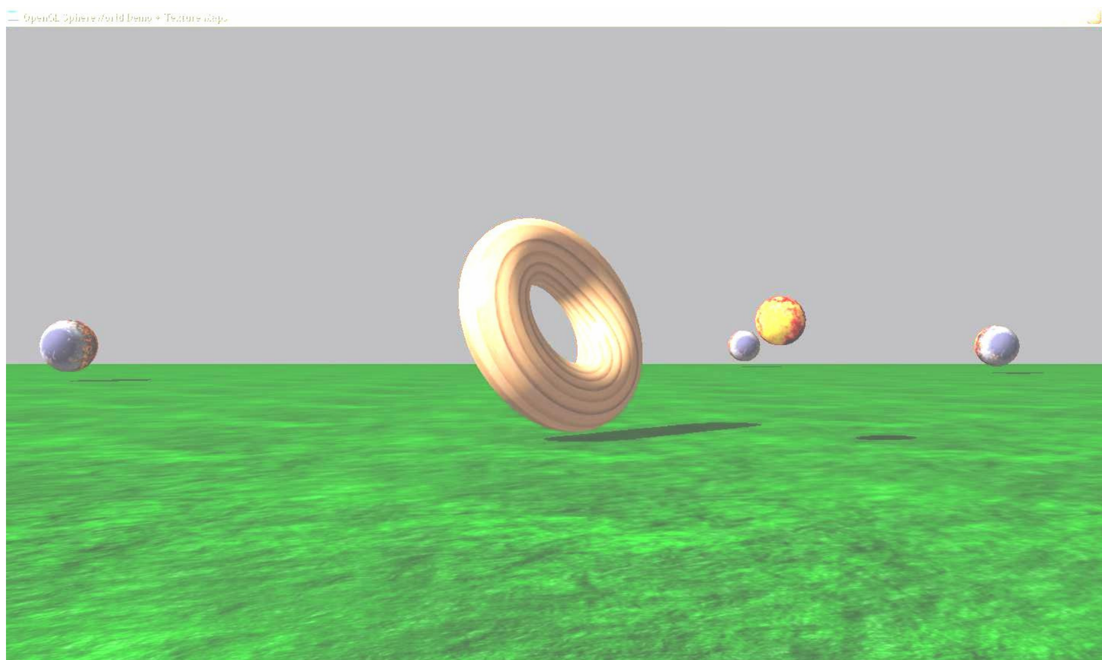
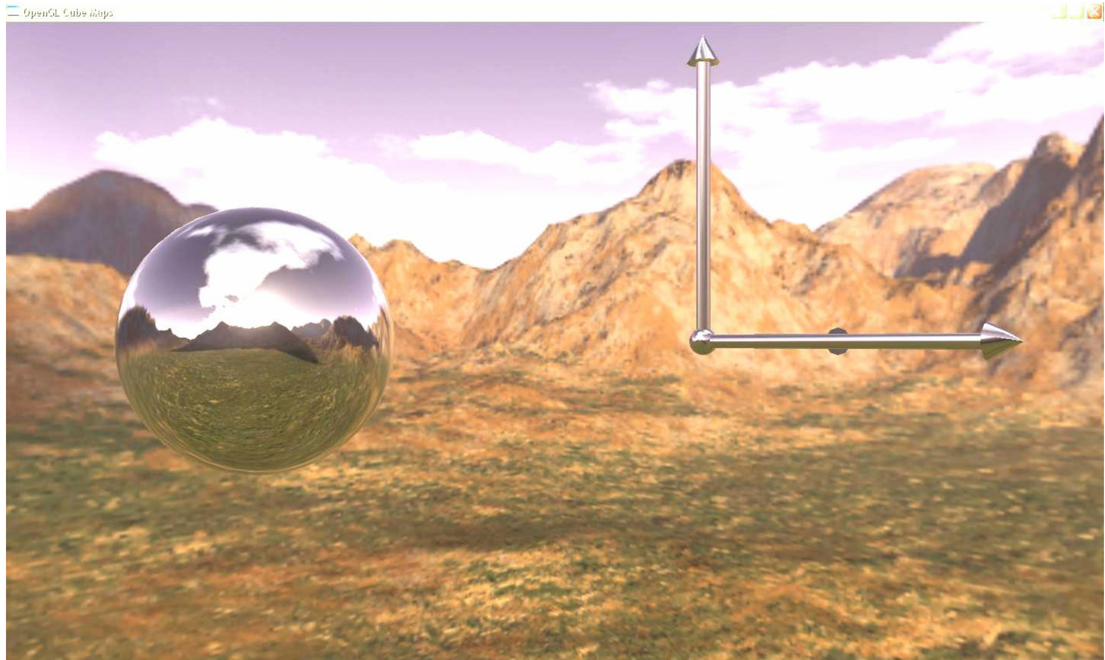


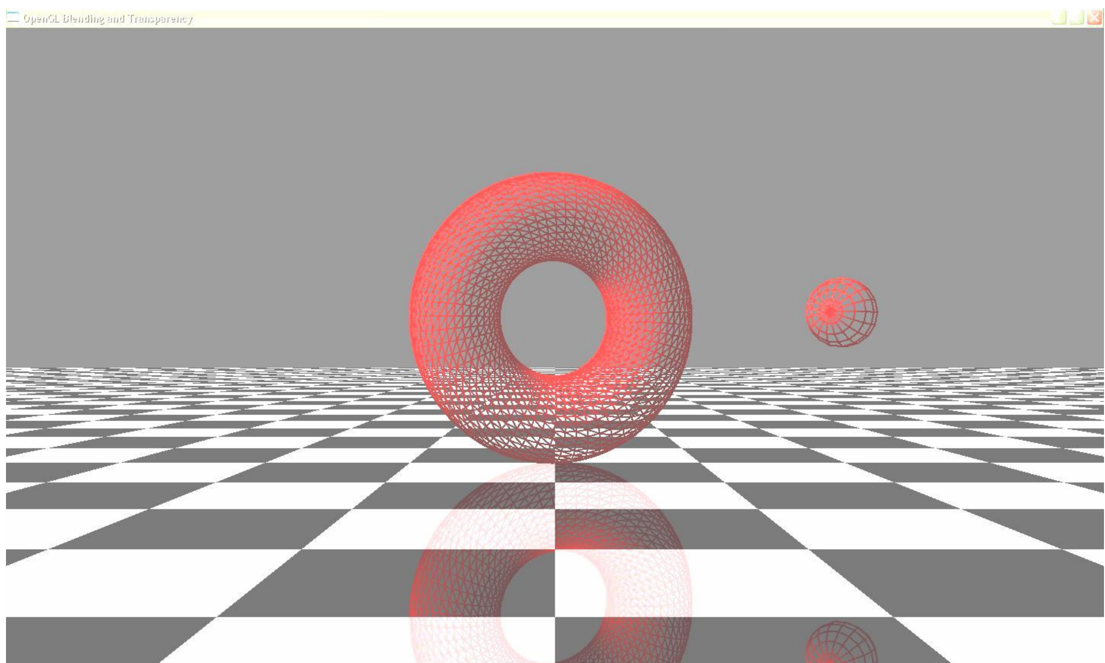
Figure 9.16 A custom software developed for visualizing and transforming several primitives and functions such as Bspline surface, conics, etc … for modeling topology changes in later stages. Using splines instead of linear models for interpolations i.e. in FEM produces more physically consistent results (See section 5.1).



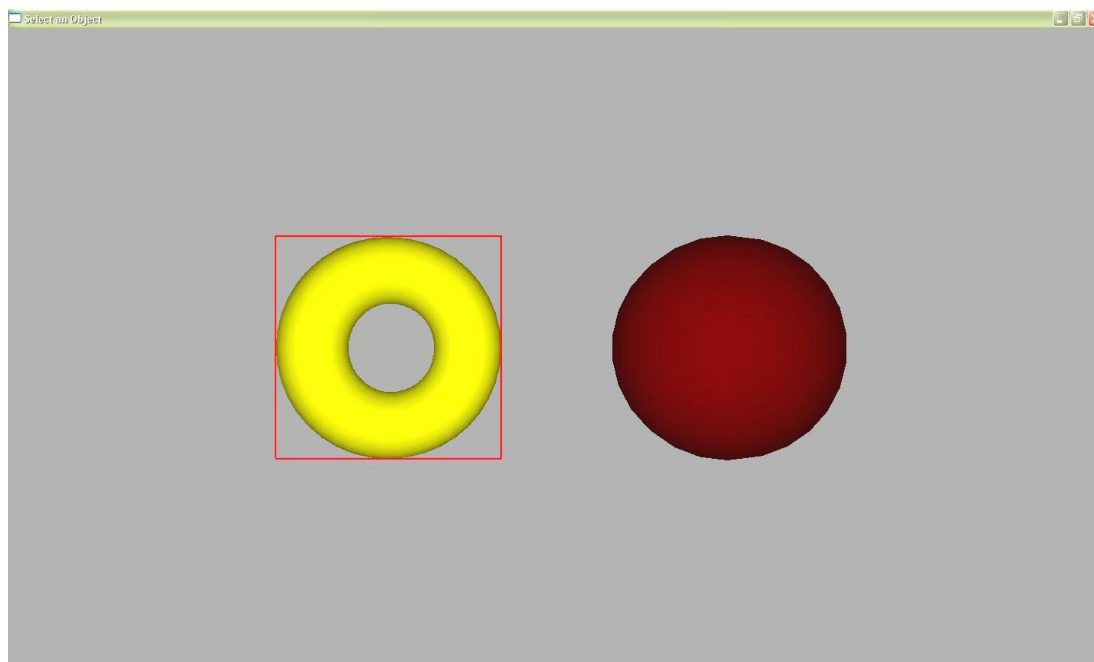
(a)



(b)



(c)



(d)

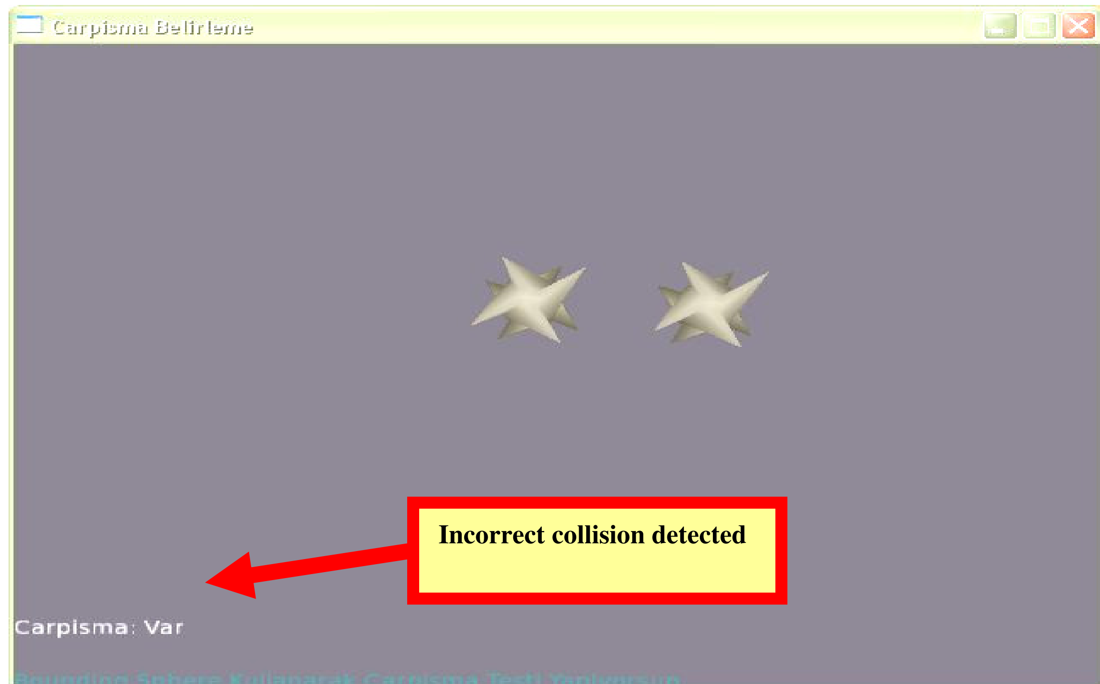
Figure 9.17 Several implementations completed for practicing real time graphics rendering referring to (Wright, & et al., 2007). (a) Texture mapping with lights and shadows. (b) Environment mapping. (c) Reflection. (d) 3-D object selection. Some of these techniques were used in the actual virtual environment (See section 5.1).

9.5 Implementations Completed during Collision Detection Research

As seen from the previous chapters, collision detection is an important subject not only in computer graphics but also in path finding problems and robotics. As seen from the papers referenced in the previous chapters, it would be suitable to use the appropriate collision detection method for appropriate situations. For example, an interactive application simulating deformable objects might require a fine triangle-triangle collision detection that needs a high computational power; but on the other hand an oriented bounding box that needs a low computational power might be sufficient for a game. Therefore prior to implementation, the needs should be analyzed well and the tradeoff between the performance of the application and the error between the collision models that will approximate the actual 3-D render model and the actual render topology should be considered. During this phase of the research period, to get insight into collision detection techniques, well known simple methods such bounding sphere and bounding box are implemented in addition to

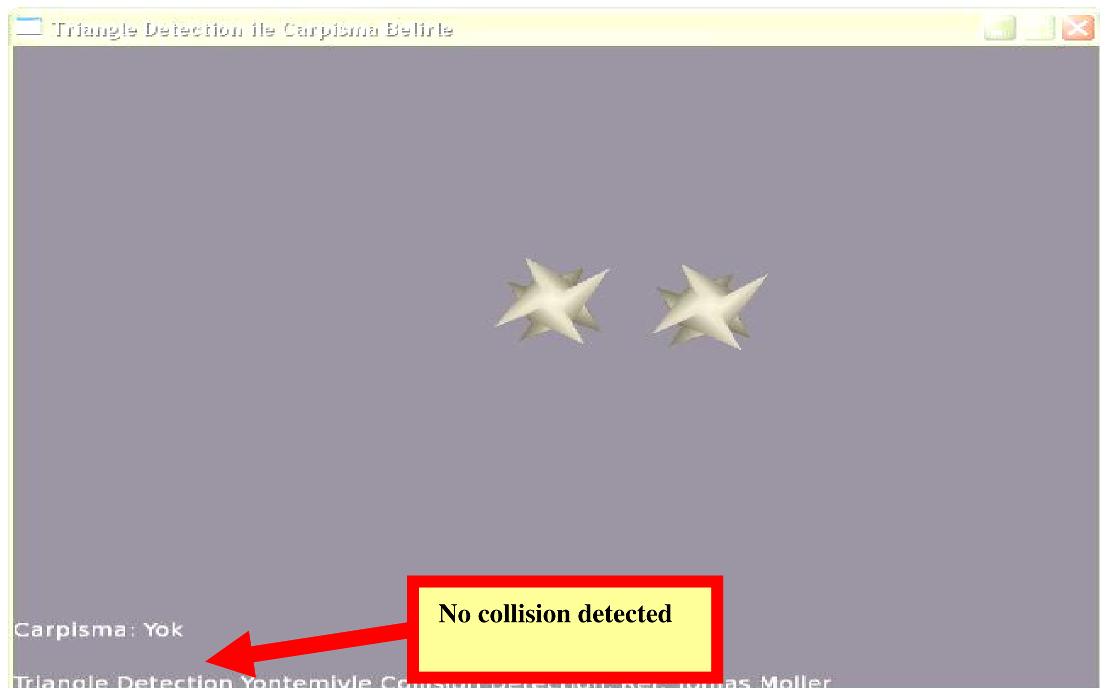
more accurate triangle-triangle collision detection. The mathematical theory and implementation details can be found at (Morefield & Malloy, 2007), (Möller, 1997). For more in depth study in collision detection, the researcher should refer to the papers mentioned in chapter 1. The results are shown in figures 9.18 and 9.19.

(a)

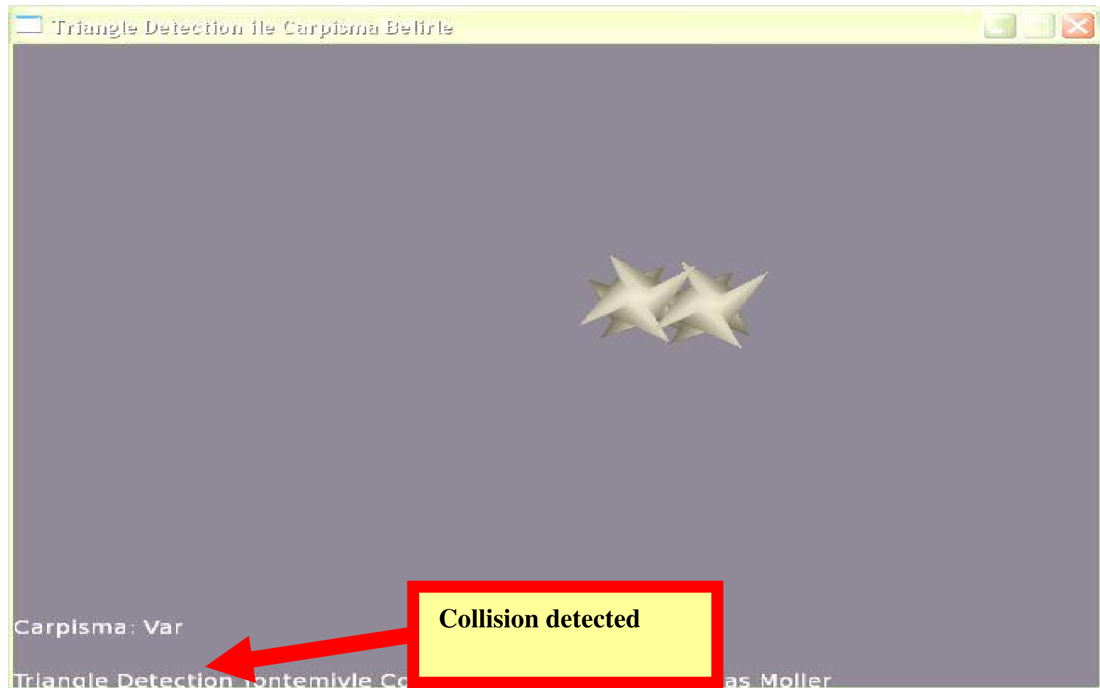


(b)

Figure 9.18 Implementing collision detection in 3-D space using bounding sphere referencing (Morefield & Malloy, 2007). (a) No collision detection. (b) Incorrect collision detection due to the high error rate between the 3-D render topology and the collision model chosen to approximate that topology. Due to its simplicity and several topological properties given in chapter 6, this technique is used where coarse collision detection is adequate (See section 6.5).



(a)



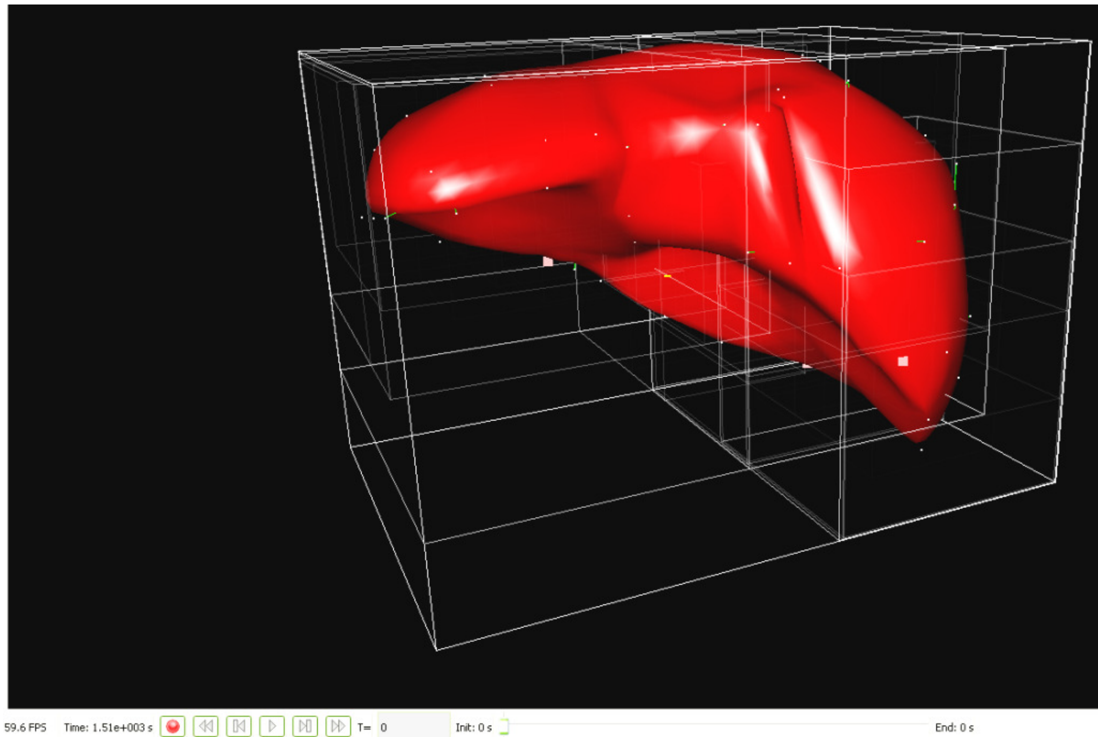
(b)

Figure 9.19 Implementing collision detection in 3-D space using triangle-triangle collision test referencing (Morefield & Malloy, 2007), (Möller, 1997). (a) No collision detected. (b) Correctly detected collision because of the minimization of the error between the collision model chosen and the 3-D render topology. This technique is used only for anatomical models where fine collision detection is needed i.e. for cutting due its high computation power demand (See section 6.6.3).

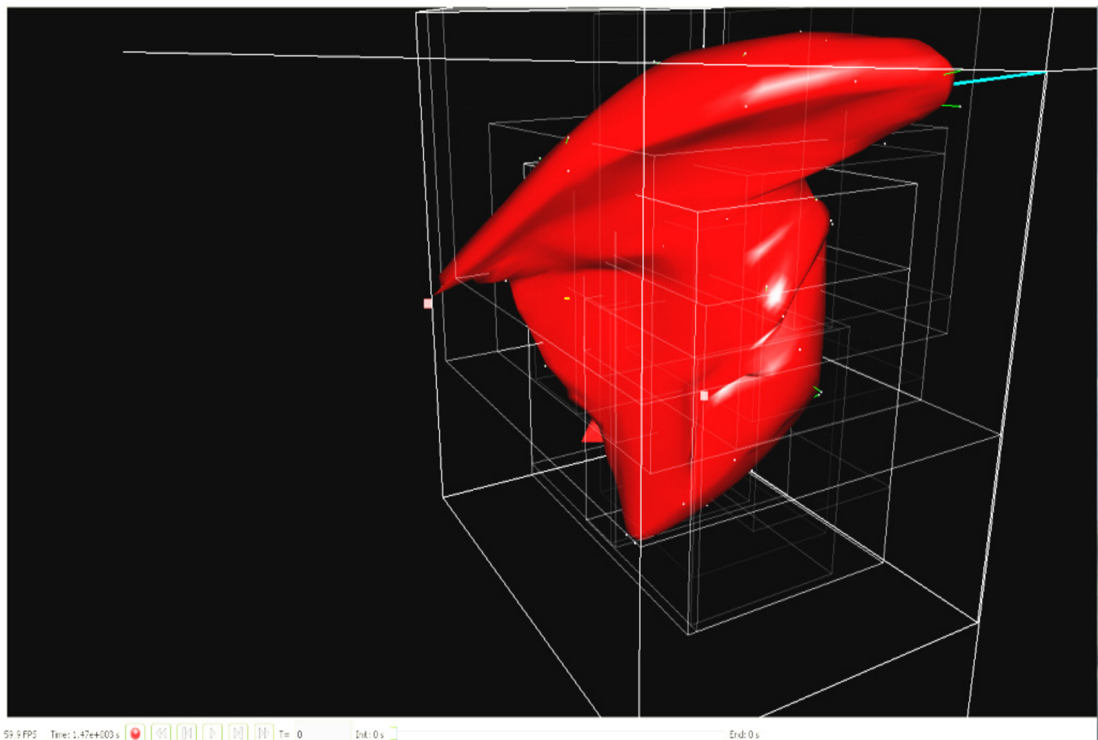
9.6 Experiences with SOFA – Simulation Open Framework Architecture

As mentioned before, SOFA is an important simulation framework developed by INRIA. It supports NVIDIA CUDA. Although not tested yet, it is claimed that it also supports several haptic devices. It consists of a rich numerical algorithm package including finite element model solvers, conjugate gradient solver, mass-spring system solver and etc…, collision detection package, collision model and render model mapping package including barycentric mapping, etc… and several space partitioning methods. During the thesis work, source code of SOFA was inspected for integration with the thesis development. But, due to the complexity of the relations between the software modules of the source code and our limited knowledge on some of the numerical methods used in SOFA at that time, this aim could not be accomplished. But this study on the source code of SOFA provided a

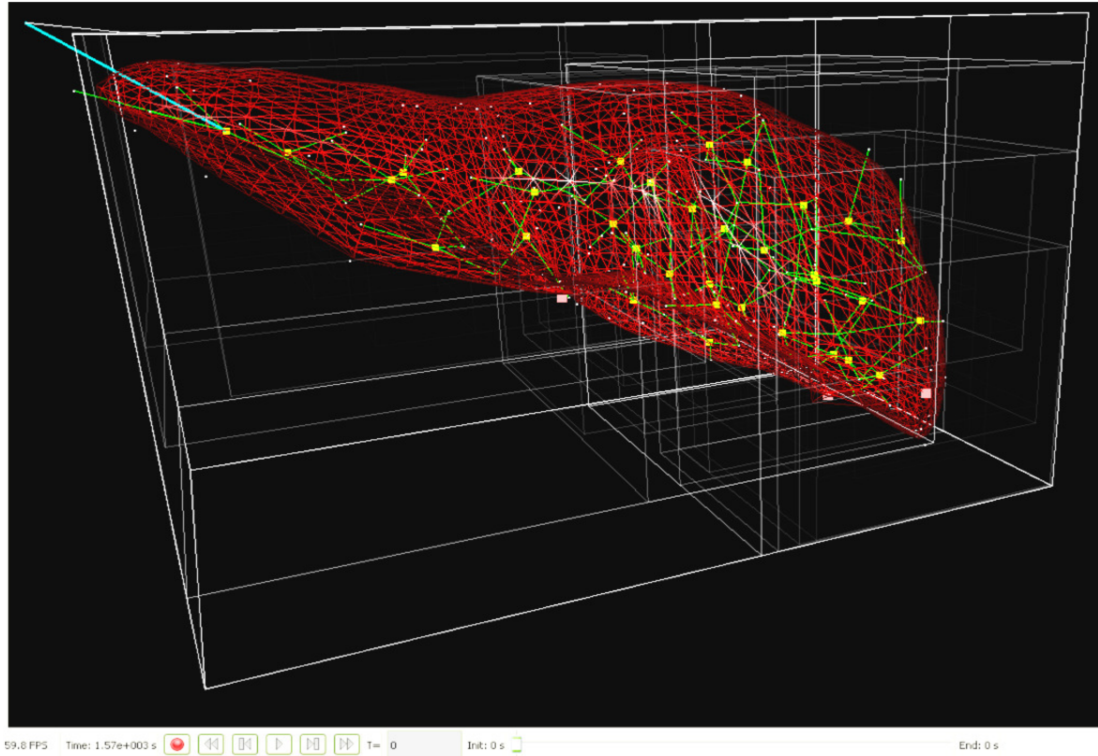
well established knowledge about the design of such simulation framework for future. Figures 9.20 and 9.21 present some results from this period.



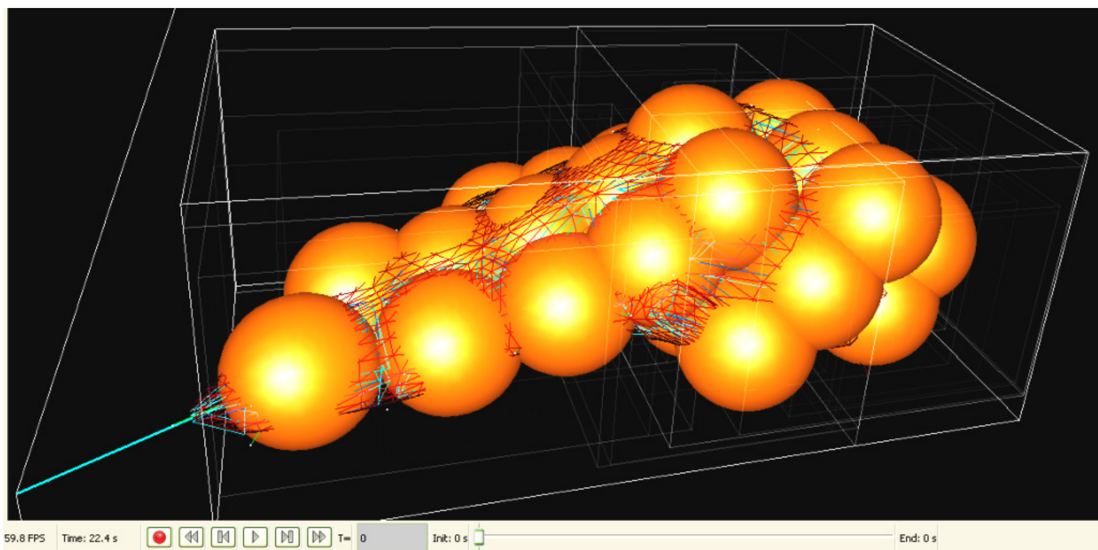
(a)



(b)

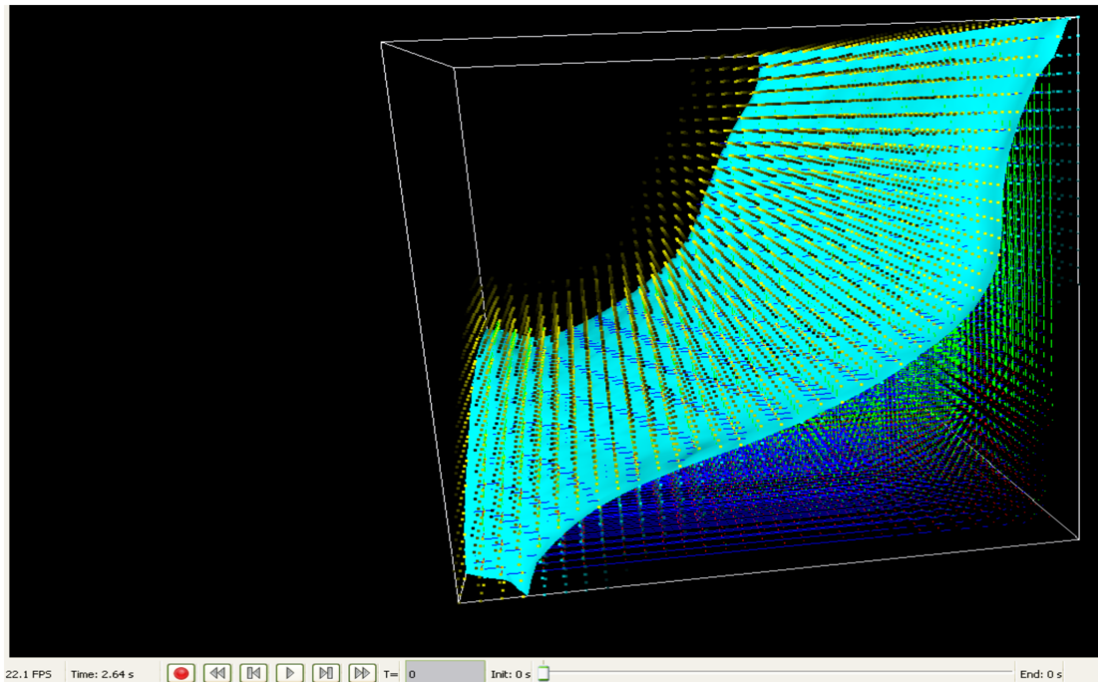


(c)

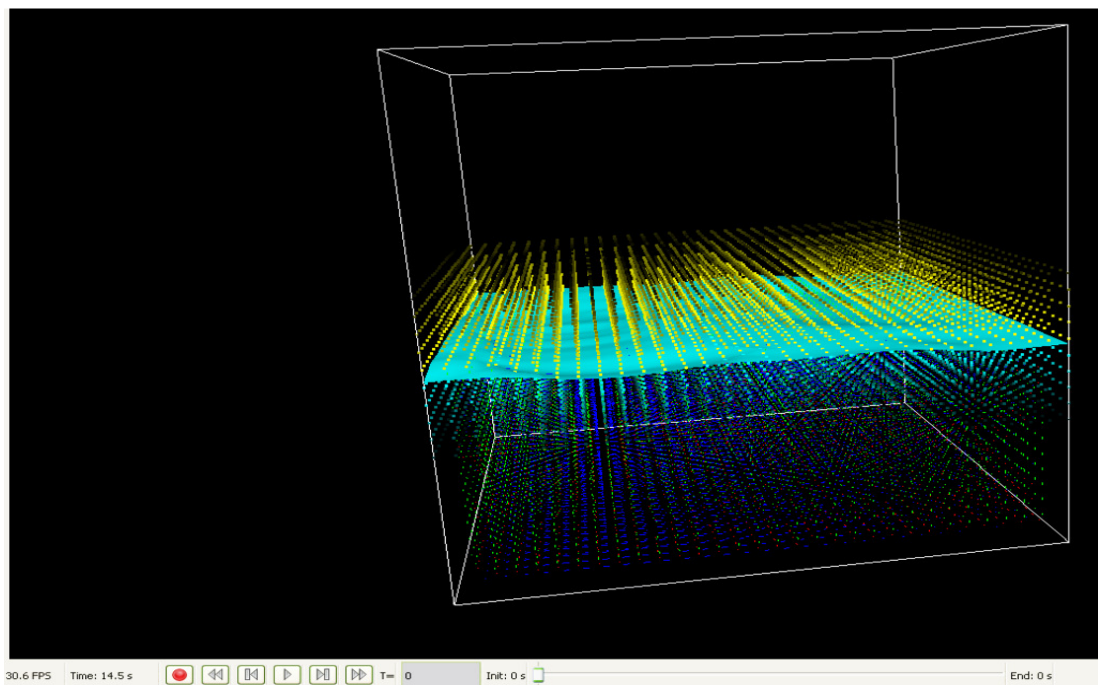


(d)

Figure 9.20 A simulation example with SOFA using NVIDIA CUDA (See section 2.3). (a) and (b) represent two different deformation states of a liver where FEM is used for numerical calculations. (c) Barycentric mapping is used to control the deformation of the render models meaning that a mechanical model of the liver is used in the FEM and each node of this model is the center of collision spheres indicated in orange color. The position of the nodes hence the center of collision spheres are defined as linear combination of suitable triangular render elements indicated in red color (d) Sphere collision models are used for collision detection with the liver to save computation sources.



(a)

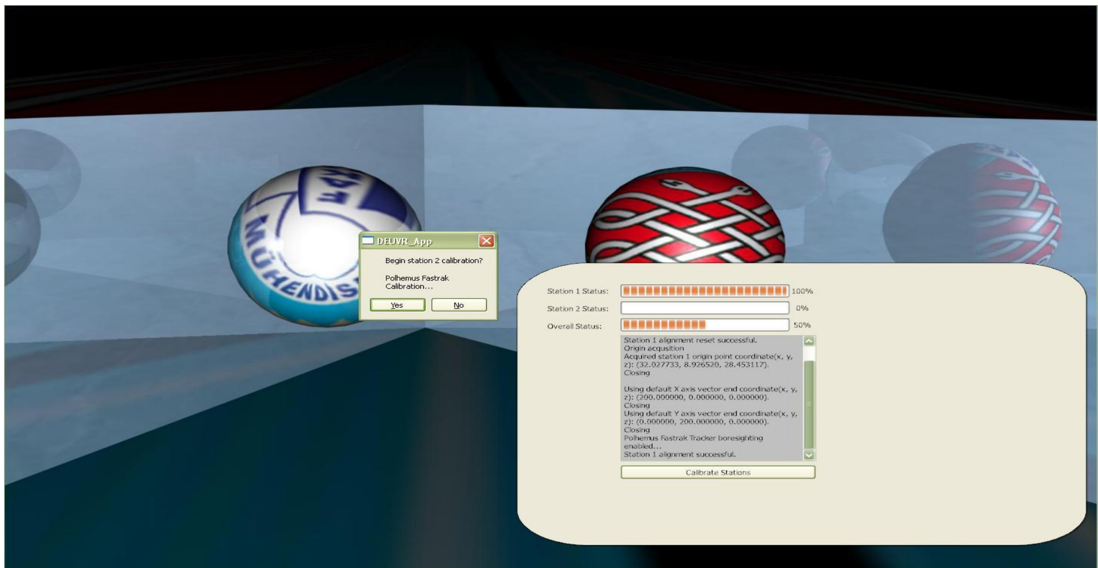


(b)

Figure 9.21 Practicing fluid dynamics in SOFA (See section 2.3). (a) and (b) represent two different states of a stable fluid. This simulation technique can be used to model some of the body fluids and their interaction with i.e. vessels in further studies.

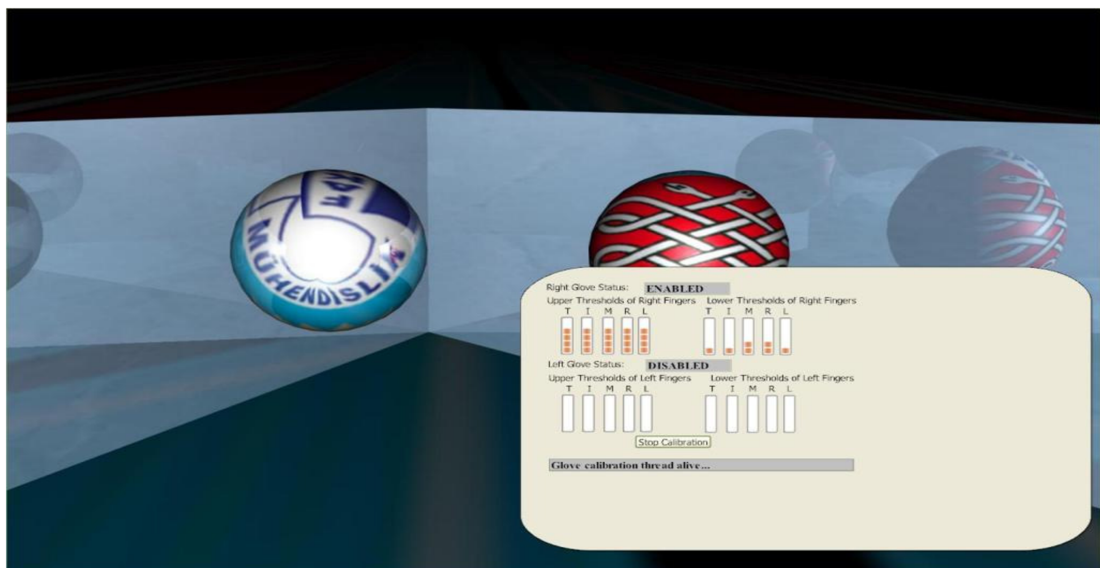
9.7 Development Stages of the Graphics User Interface using Qt Development Kit

(a)



(b)

(c)



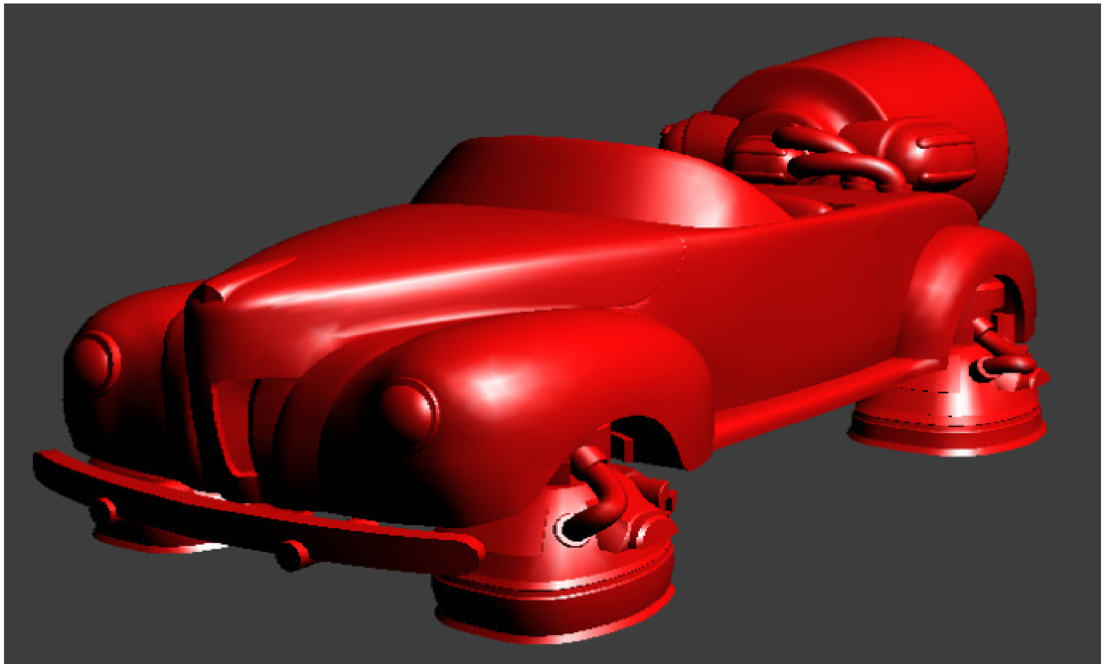
(d)

Figure 9.22 The setup and calibration GUI developed for the application with Qt Kit. The GUI is used for (a) entering to the virtual environment, (b) the tracker calibration, (c) video settings and (d) data glove calibration.

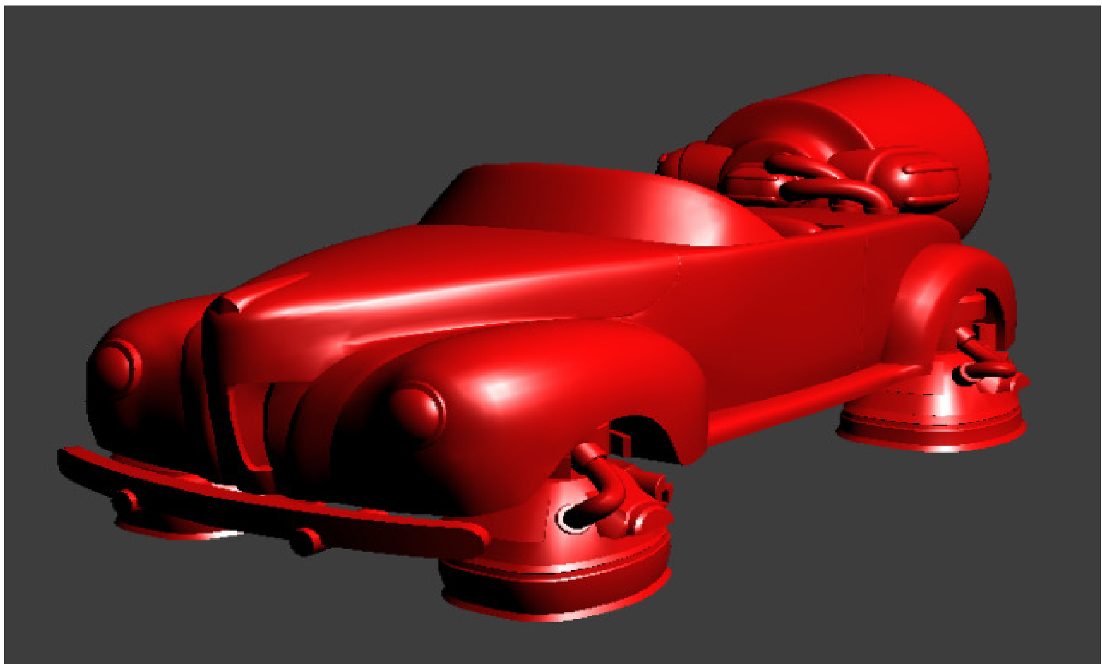
9.8 Experiences with Cg and GPU Programming for Graphics

Prior to the implementation of the GPU programming for graphics for the actual software, several stand-alone practices had been completed. This was necessary to learn implementation of rendering codes for vertex processors and fragment

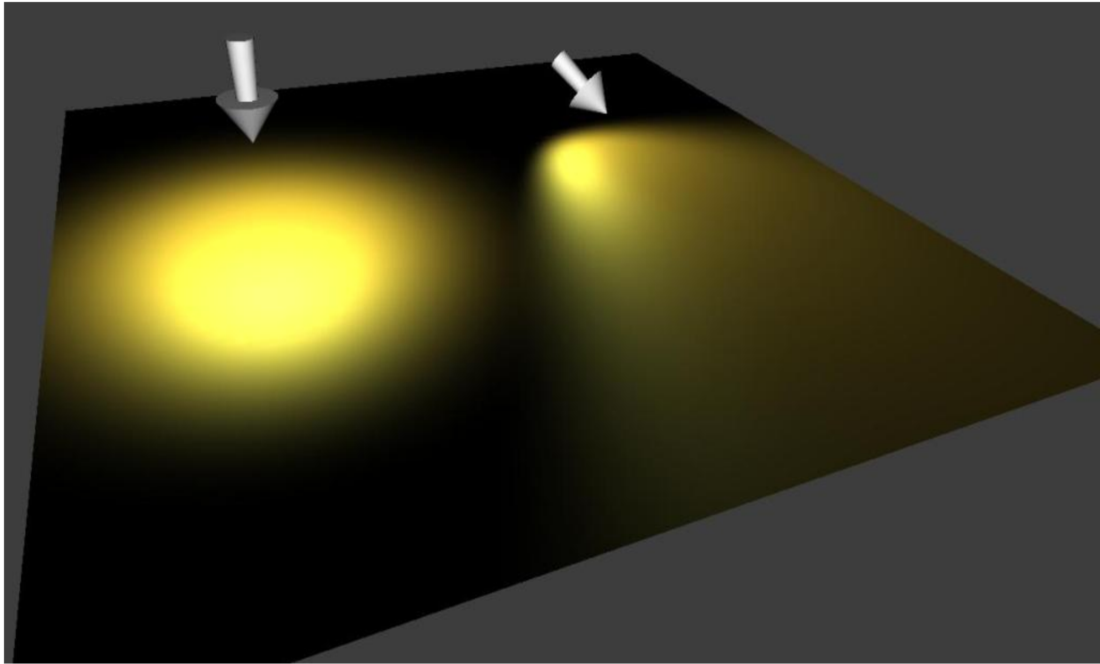
processors, how they were executed on the processor and their differences. The following figures are some results from this period.



(a)



(b)



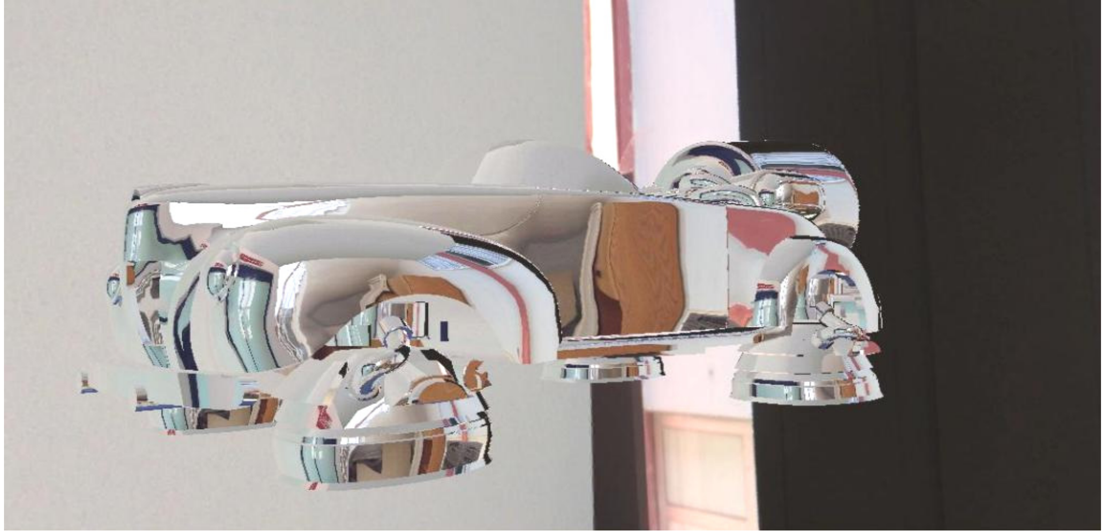
(c)

Figure 9.23 Practical implementations on lighting using vertex processors and fragment processors in the programmable graphics pipeline of the GPU. (a) and (b) present vertex lighting and fragment lighting respectively. Notice the smoothness in the specular lights in (b) because lighting code is implemented in the fragment processor. In this case this code is executed for every pixel in the scene. On the other hand, in (a) the lighting code is implemented in vertex processor. In this case this code is executed for every vertex in the scene. The lighting for remaining pixels where no vertex exists, the lighting is interpolated as a linear combination of corresponding vertices, which is in fact Gouraud lighting. In (c) two spotlights are implemented using vertex processors (See section 5.1 and chapter 4 for GPU programming).

(a)

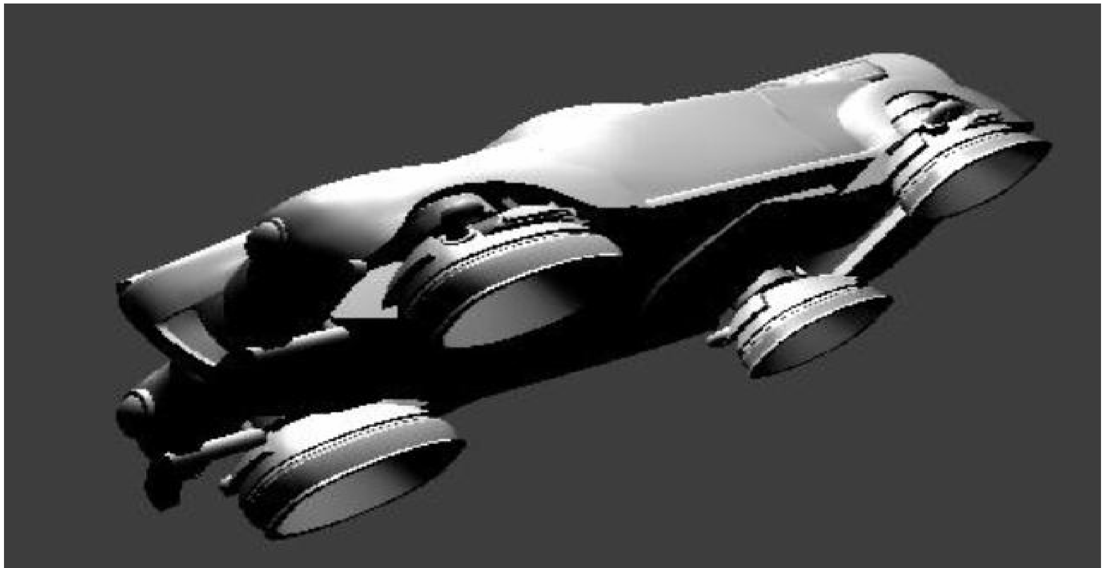
(b)

(c)

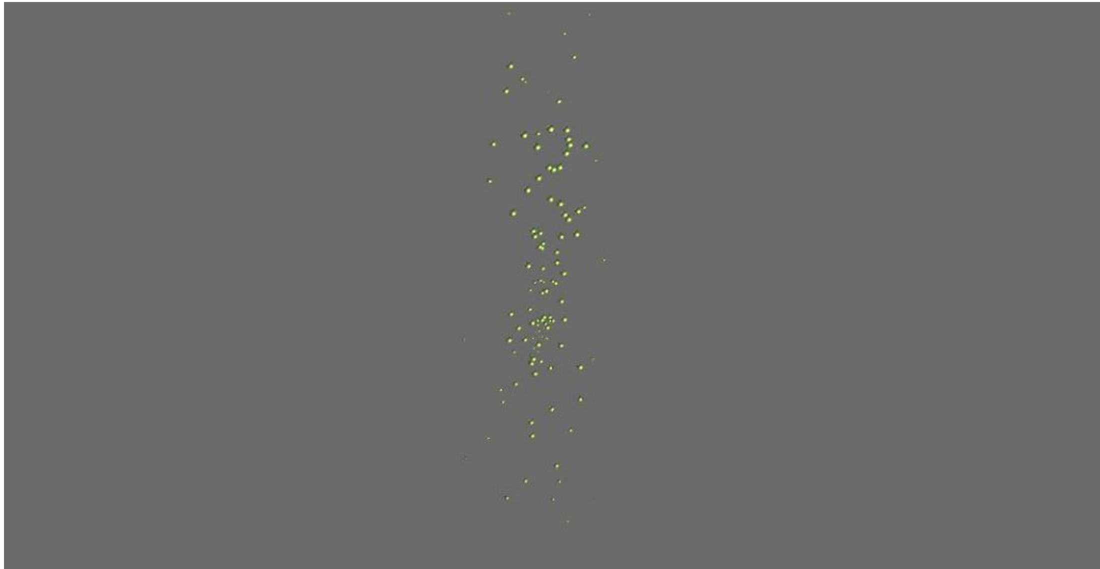


(d)

Figure 9.24 The pictures above represent snapshots from the real time rendering of dispersion, reflection, refraction calculated on GPU using Cg. The environmental mapping method is used to perceive a car in an real environment (See section 5.1 and chapter 4).



(a)



(b)

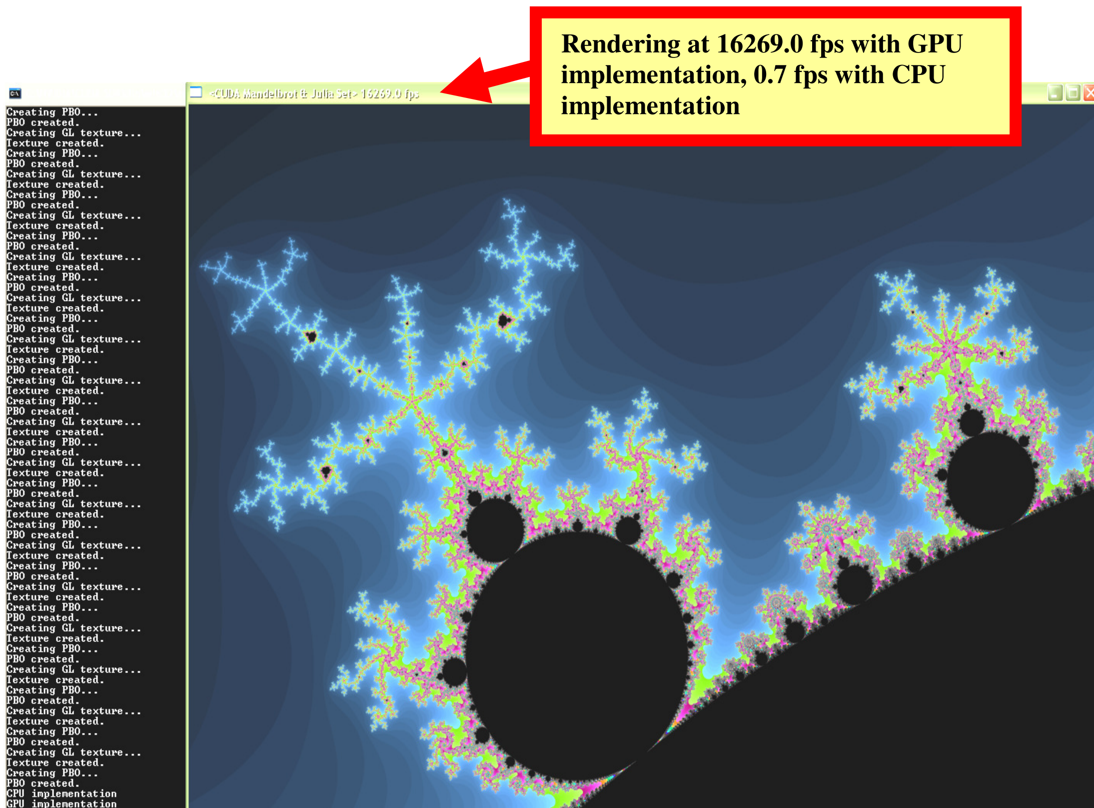
Figure 9.25 In (a) practicing affine transformation using vertex and fragment processors; and in (b) practicing particle simulation using vertex and fragment processors in the programmable graphics pipeline using Cg (See section 4.5).

9.9 Experiences with NVIDIA CUDA and Performance Comparisons for Further Projects and Possible Implementations

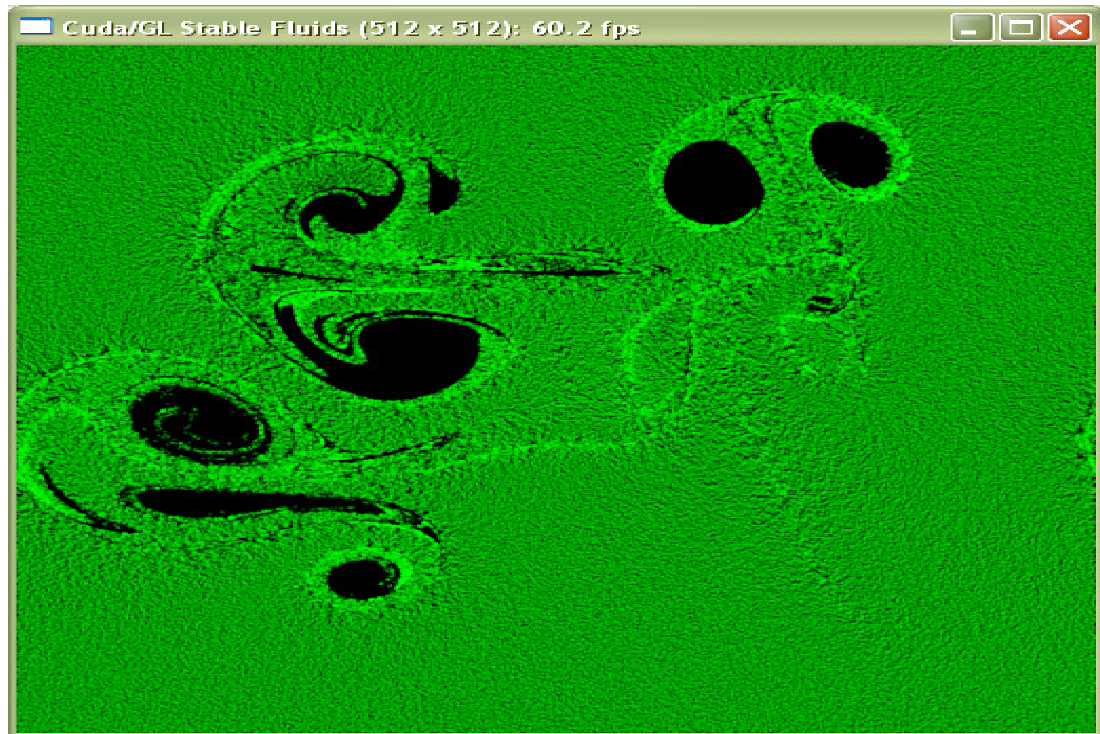
Integration of CUDA to the current developed software will enable using more computationally demanding but on the other more physically consistent numerical methods such as finite element models (FEM) via GPU implementation. Although, the aim was to use FEM to model elastic objects, time was not adequate. But several code implementations were investigated for getting insight to using CUDA. The following pictures present some results.

```
C:\NVIDIA\CUDA SDK\Bin\win32\release\deviceQuery.exe
CUDA Device Query (Runtime API) version (CUDART static linking)
There are 2 devices supporting CUDA

Device 0: "GeForce GTX 295"
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 939261952 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.24 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads
can use this device simultaneously)

Device 1: "GeForce GTX 295"
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 939196416 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.24 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads
can use this device simultaneously)

Test PASSED

Press ENTER to exit...
```

(a)



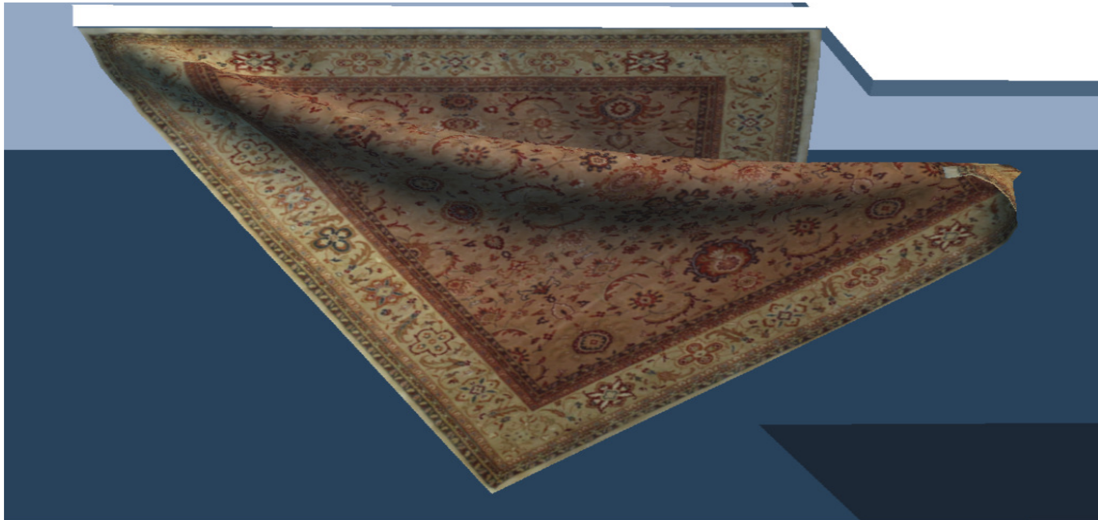
(b)



(c)

Figure 9.26 Practicing NVIDIA CUDA for several simulations. (a) presents the hardware configuration on which the implementations are done. (b) presents an simulation of Mandelbrot fractal for different depth levels. The pixels with black color are in the Mandelbrot set and the other colors represent the rate of divergence of the recursive generating sequence to infinity. The Mandelbrot set is generated in real-time on GPU. (c) presents a fluid dynamics simulation for a stable fluid with defined boundary conditions. Navier-Stokes Equations are numerically solved on GPU. These are all simulated in real time. The necessary solvers and example source codes are found in NVIDIA CUDA SDK. They should be compiled a priori (See chapter 4 for programming GPUs for general purpose computations).

9.10 Experiences with NVIDIA PhysX and Performance Comparisons for Further Projects and Possible Implementations

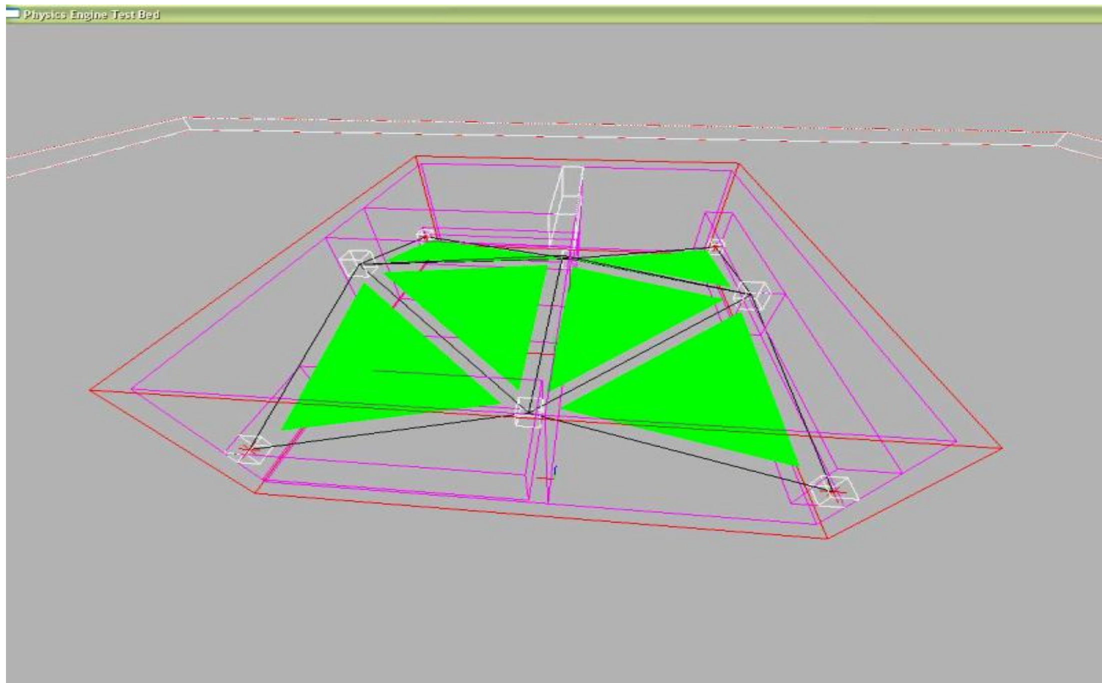


(a)

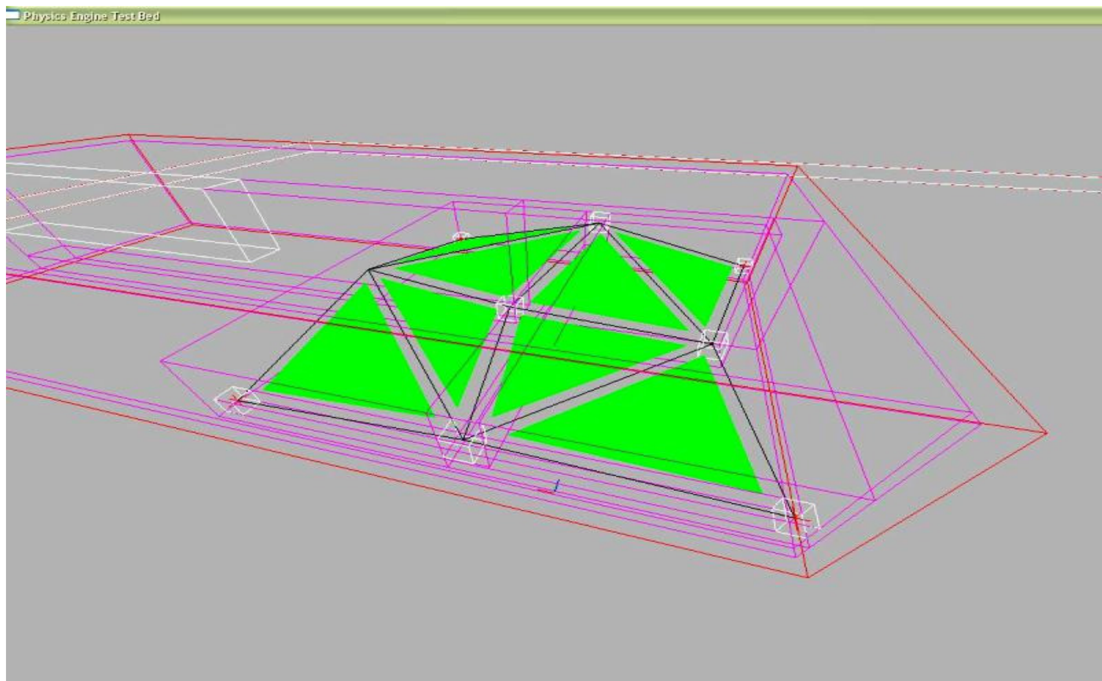
(b)

Figure 9.27 Practicing NVIDIA PhysX for several simulations for taking advantage of physics rendering with GPU. (a) presents a texturize cloth made up of 2-D mass-spring system. Vertices at the top part of the cloth are constrained to a rigid body at the ceiling, hence they are not moveable. The state of the remaining vertices is controlled by the governing differential equation $d^2x/dt^2=F_{net}$ driven by the gravitational force. If desired, a force can be applied to a vertex by selecting and pulling or pushing it to appropriate direction. From there on, that applied force will also be used for calculating the net force to drive the governing differential equation of the system (b) presents a cutting operation which can be executed by canceling links of the selected vertices to adjacent vertices. The necessary solvers and example source codes are found in NVIDIA PhysX SDK. They should be compiled a priori.

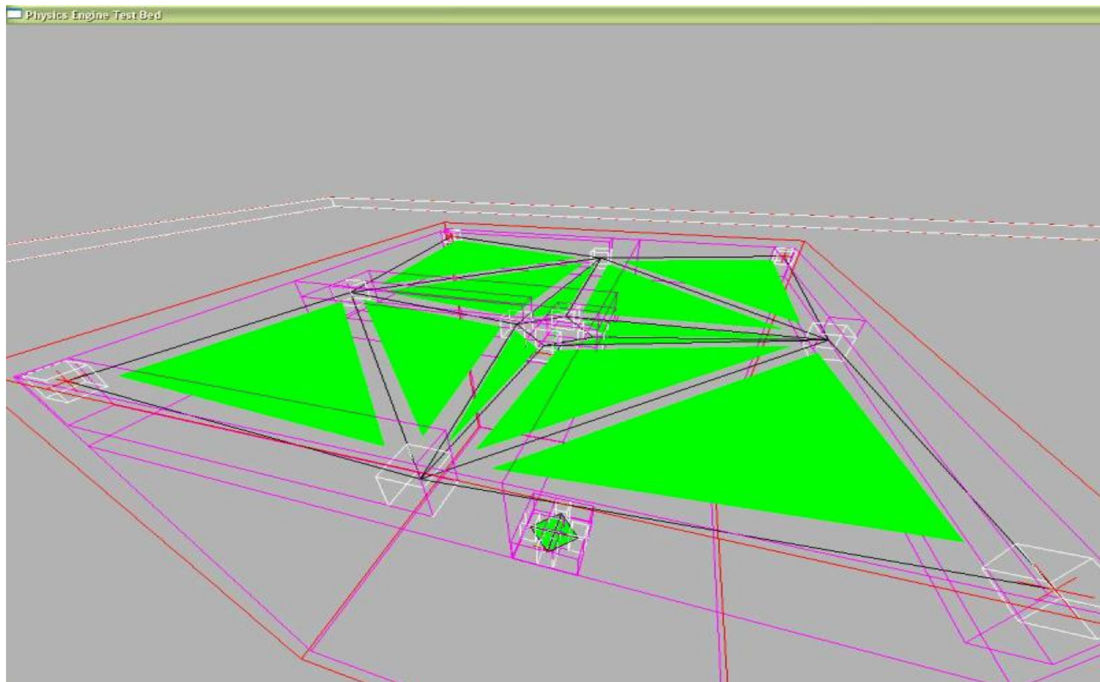
9.11 Construction of Mesh Spring Structures and Implementation of Topology Processing and Refinement for Mesh Cutting Operation Using Bullet Engine



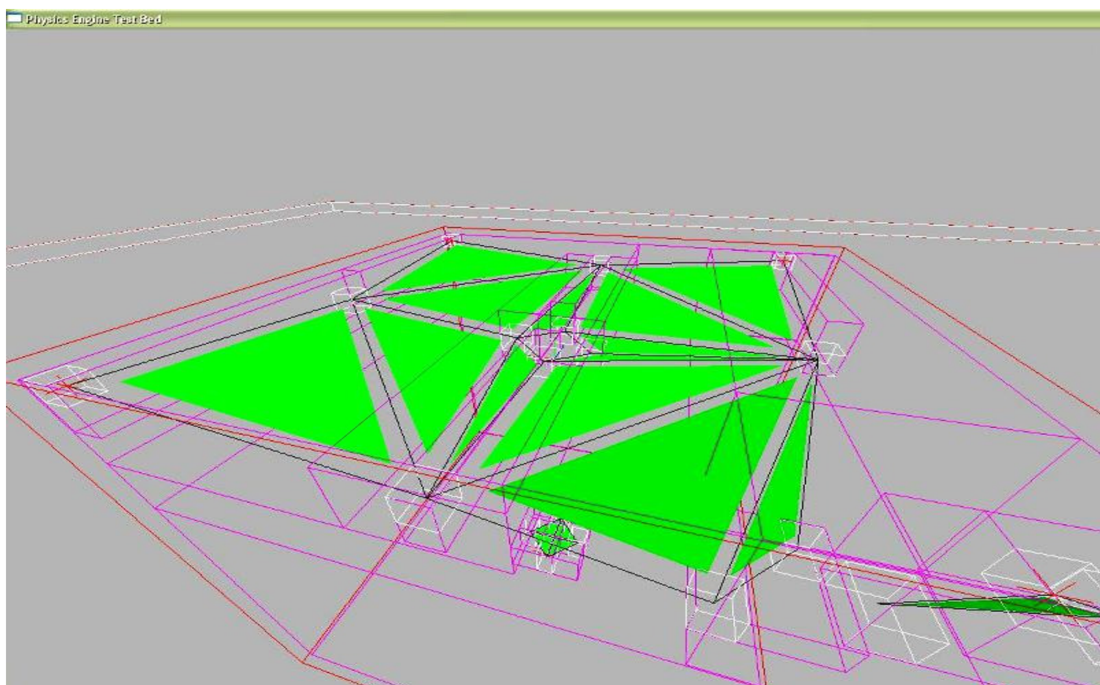
(a)



(b)



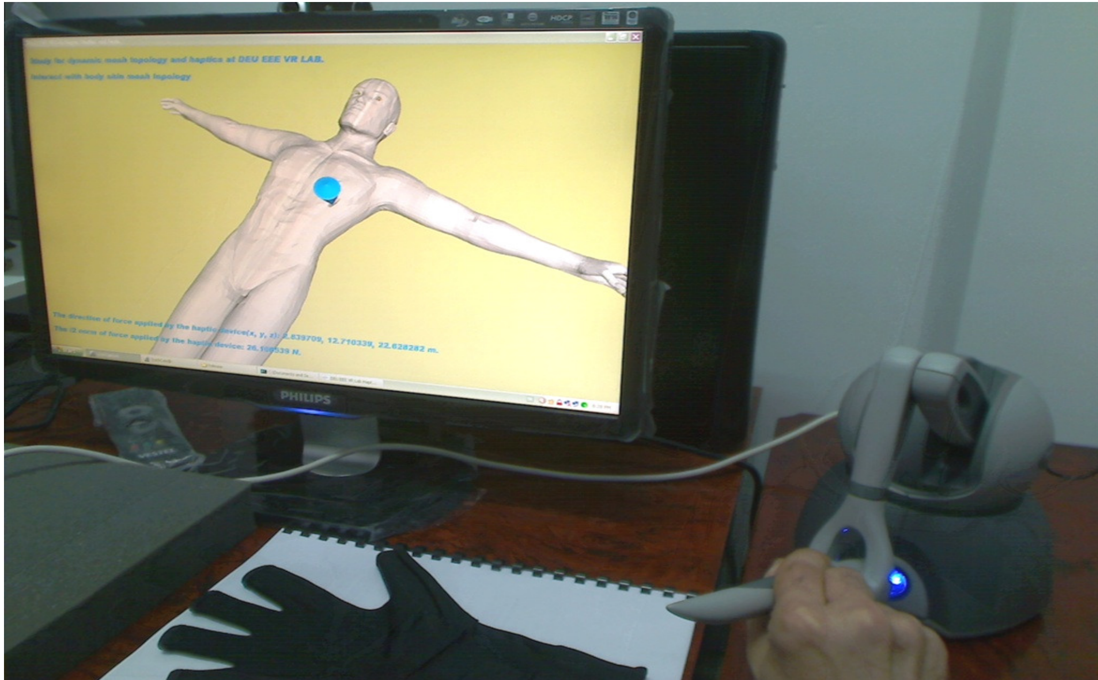
(c)



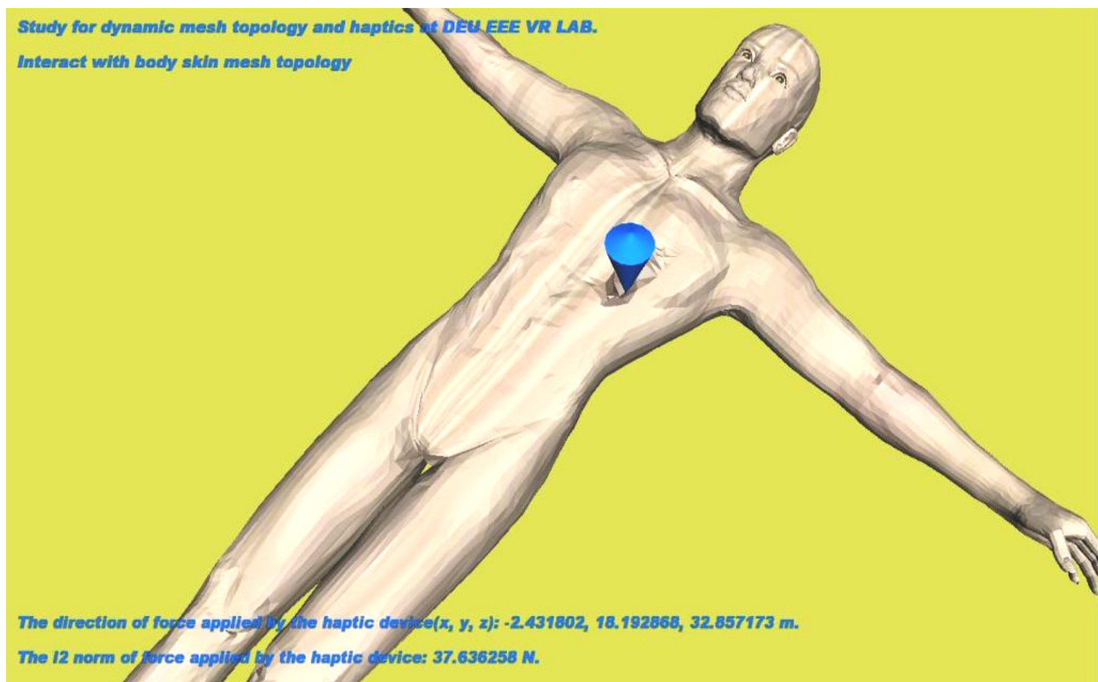
(d)

Figure 9.28 (a) and (b) present a construction of 2-D mass-spring system with AABBs at vertices indicated in white color and model partitioning indicated with violet color for collision detection. (c) and (d) present topology processing for the actual 2-D mass-spring system for simulating a cutting operation. For each cut surface, necessary extra vertices, masses, velocities, related AABBs and faces are generated both on the remaining mass-spring system and on the cut piece (See chapter 6).

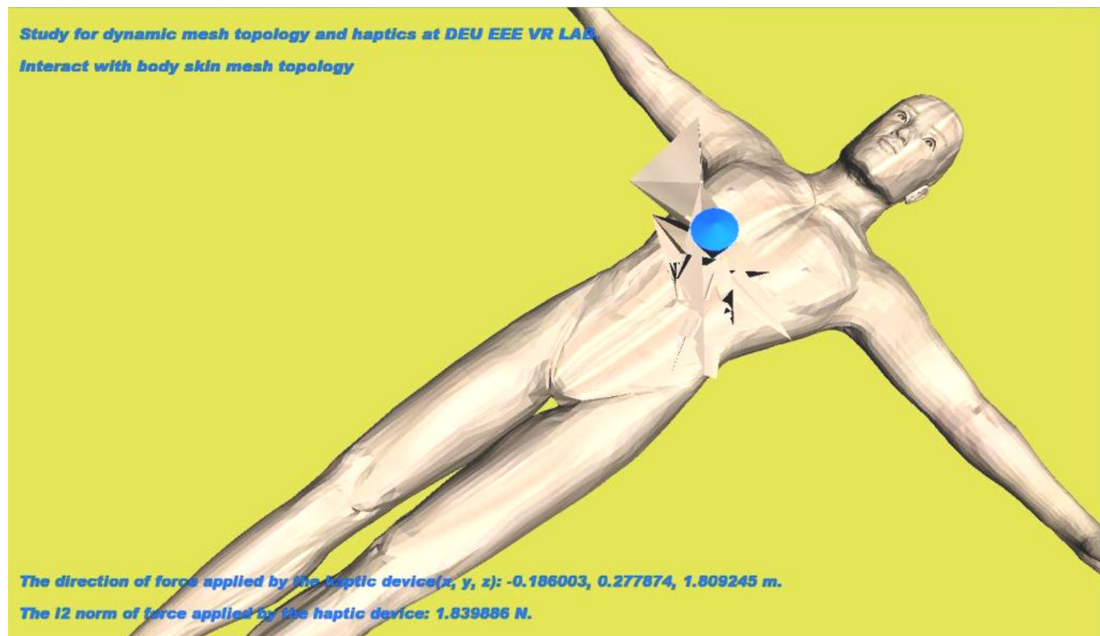
9.12 Haptic Rendering Implementation Results



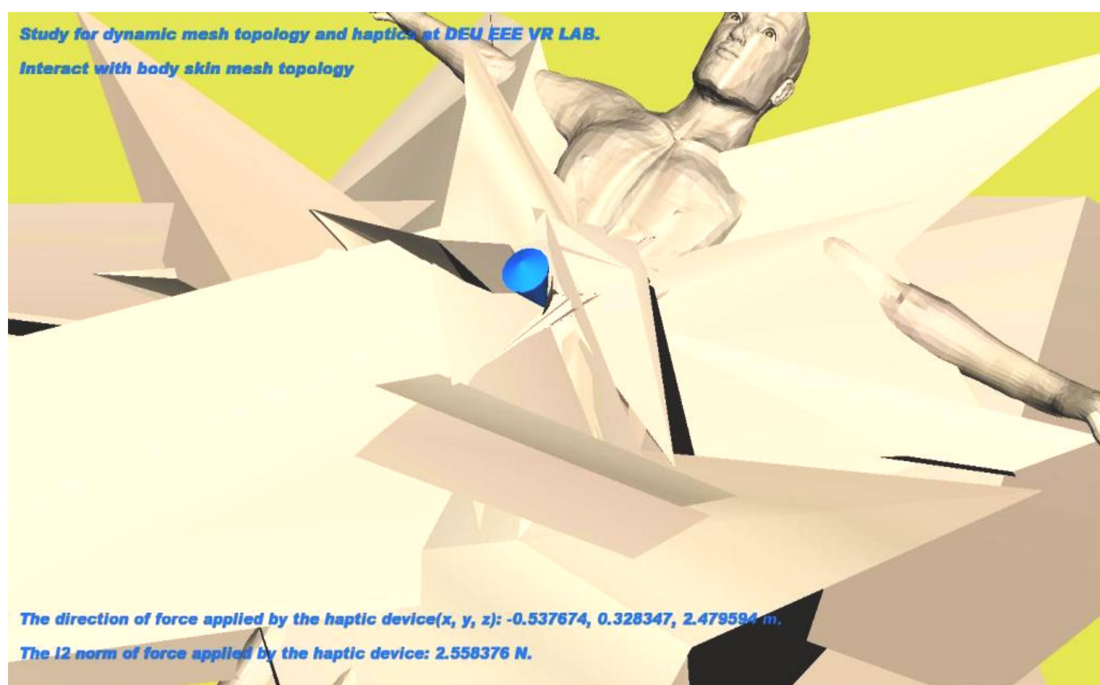
(a)



(b)



(c)



(d)

Figure 9.29 Initial haptic rendering module developed during the thesis period. The integration method used in the module is an Explicit Euler Method. (a), (b) show that for small time steps and low force loading condition, the mesh topology of the body can be deformed via the haptic device. The norm of the applied force, the application direction is overlaid on the screen. (c), (d) show that in high force loading conditions, the mesh system diverges from its equilibrium point (See section 6.9).

CHAPTER TEN

CONCLUSIONS

Prior to going on with the conclusions of the thesis work, the author's viewpoint of a scientific research should be indicated. Computer technology has led to the development of many applications targeted to scientific researches; but the more important concept than an application is the mathematical theory of computation and complexity that forms a basis for all of the today's programming languages, algorithms and computing machines. Therefore the application should not be the only target of the research in engineering, but a tool for understanding the origins of the established theories, thinking styles of the pioneers in the computation field, the "why" and "how" questions these pioneers asked and their solutions. The researches of important pioneers such as Charles Babbage, John Von Neumann and Alan Turing should be well analyzed to maintain a complete and connected background for synthesizing new theories and designs in the field. While doing these, gaining a working knowledge in applied sciences such as physics, chemistry and biology will definitely change the way a researcher handles a problem and understands the nature.

At the end of thesis work and at the end of the software development stage, the following goals were achieved:

- A well equipped and operational computer graphics and virtual reality laboratory was established with the setup seen in chapter eight. The laboratory is the first one in Dokuz Eylül University that specifically target the researches in mathematical theory and applications of computer graphics, scientific simulation and visualization, the architecture and programming of graphics processing units for not only graphics processing but also for general purpose computing and scientific computations.
- A functional interactive virtual environment in which the user can interact with the surrounding rigid and deformable objects and with the other collaborators was constructed.

- Necessary software modules were developed for interfacing the motion tracker device and the data gloves. The software module for interfacing the haptic device was also developed but it was in the early stages.
- Necessary software modules were developed for physics and graphics rendering. Then the communication connections between these modules were established.
- A user interface was developed with Qt to enter the virtual environment, perform tracker calibration, data glove calibration and video settings for the user. Hence, the user's position in real world is transformed properly to coordinates in the virtual world. The user could see a virtual hand deforming according to bending amounts of his/her fingers. This was necessary to generate hand gestures in the virtual environment or to trigger certain events.
- A 3-D hand mesh was rigged and skinned so that when the user in the real world moved his/her fingers, the same movements were also done by the rigged hand mesh using the bending values acquired from the data glove sensors. This gave the user a more immersive and natural feeling in the virtual environment.
- The lighting and texture rendering were accomplished by programming the programmable pipeline of the graphics processing unit. NVIDIA Cg and Microsoft HLSL shader languages were used for this purpose. Accomplishing the rendering tasks on the GPU side, released the CPU for other tasks such as computing the simulation parameters and acquiring data from external hardware such as the motion tracker device and the data gloves.
- Inside the virtual environment, the user could manipulate the virtual objects with an additional 3-D graphical user interface that is shown when the virtual object is being touched.
- Collisions between rigid and deformable objects were detected and parameters such as contact points, normals in local and world space and penetration depths were calculated.
- The user could also cut or deform the soft models.

- NVIDIA CUDA API was evaluated for the research accomplished on programming GPUs for scientific and general purpose computing during the thesis period.
- Several other engines other than Ogre3D and Bullet were tested for their capabilities and usability for future development projects. Those engines were SOFA, ODE – Open Dynamics Engine, NVIDIA PhysX, OpenTissue, SPRING Simulator Framework, OpenInventor, OpenProducer and Havoc.
- A simple augmented reality application was developed for registering the 3-D virtual objects with the video and for tracking the user hand to control a 3-D virtual user interface in real time. The segmented features for registration were artificial and hence imposed by humans.
- Haptics rendering with soft anatomical and tissue models was accomplished by using the methods in chapter 6 via OpenHaptics API.
- Preliminary studies and researches for finite element modeling were accomplished for more precise mathematical representation of the system dynamics being simulated.

The following criterions should be considered to enhance the current application:

- Further improvements for topology processing should be done in order to capture the dynamics of the soft models. The numerical stability of the differential equation solver was not appropriate for scientific usages.
- The opinions of several users should be obtained for user interface development.
- Physics rendering and dynamic system modeling were done using CPU. Moving these calculations onto GPU will free the CPU for other tasks and increasing the frame rate.
- Instead of using a mass-spring system, better numerical methods such as finite element modeling will produce more physically consistent results.
- Instead of rigging and skinning a hand, a stationary calibrated camera system can be used to segment hand features and then inverse kinematics methods can used for estimating the rotation and translation matrices of a hand and

fingers for user interaction and capturing the state of the hand to the virtual environment.

- The augmented reality application may use natural environment features. Hence, it will be more usable and practical to be used in outside environments for information visualization.
- The 3-D anatomical models used in the thesis work should be replaced with 3-D models reconstructed from MRI, CT data acquired from Dokuz Eylül University Faculty of Medicine. For initial researches, the medical data at (United States National Library of Medicine National Institutes of Health [NLM], 2010) can be used.

According to our observations during the thesis period, it can be concluded that VR has the ability to change the interaction styles not only between the human and the computers but also between the humans. The interactive and immersive nature of VR can shorten time needed to understand the fundamental of the dynamics of a scientific processes, because a well-designed VR system can not only simulate the dynamics of a scientific process in real time but also takes the user to the place of occurrence of that process. The interaction ability provided to the user for affecting the current dynamics of the simulated process is a key for providing a learning opportunity of the different behaviors of the simulated system. Additionally, by increasing the intelligence of the agents in the VR systems will increase the interaction capabilities of the system with humans and alter its behavior accordingly.

AR is rather a new way of visualization of the data at hand and also a new way of human computer interaction. As in VR, its development mostly depends on expensive and powerful computation units, but on the other hand, its utilization as a way of human-computer interaction and visualization is expected to increase in the future. At the time this thesis was written, many entertainment companies such as Microsoft and Sony were developing user interfaces for many games and applications using AR technology for XBox360 and PlayStation 3 respectively. Apart from the games, AR changes the real world a person lives in into a mixture of reality and unreality which is limited by the user imagination and the intelligence of

the computing system. The AR software developed during the thesis work is in very preliminary stages and needs improvements. It needs markers in order to track the user and control the virtual user interface. Markerless tracking of the user, more intelligent software that can understand the emotions of the user and respond according to user's movements will definitely make the interaction more natural and will improve its usability.

On the other hand, as seen from the experimentations on graphics processor programming accomplished during the thesis period, programming graphics processing units for scientific calculations and for general purpose computing is an important and promising research area for both hardware and software perspectives. The performance gains of x10 and x100 over the central processing units (CPUs) had been observed during the thesis work. The benchmarking was done using the programmable graphics pipeline for implementing several graphics processing techniques by Cg and NVIDIA CUDA for numerical computing research purposes on graphics processing unit (GPU). The CPU used was Intel Q9550 and graphics processor was NVIDIA GTX 295. The survey completed in the thesis work showed an increasing usage of GPUs in scientific computing. This is not only because of the increasing amount of data to be analyzed by the researchers, but also because of the existence of algorithms and numerical analysis methods that are parallel in their nature and the ability of GPUs in performing matrix vector operations very quickly as these operations are why GPUs were designed and optimized for. The key concept is the parallelism that these GPUs present with their many core hardware architecture and their high bandwidths for data transmission compared to the general purpose CPUs. Additionally, availability of appropriate compilers, high level programming environments for use in heterogeneous computing systems where both CPU and GPU exist simultaneously and the increasing number of GPU general purpose programming APIs such as NVIDIA CUDA, ATI Stream and OpenCL enable the researchers in diverse fields benefit from the computational power that GPUs have. More in depth research on GPUs should be carried on in the future projects. Current stages completed in the thesis work may be ported to GPU code where appropriate.

The collision detection and numerical solutions of differential equations accomplished on GPU will definitely increase the performance of the software.

The planned research areas of the laboratory for undergraduate and graduate levels are as follows:

- Mathematical theory and applications of collision detection in 3-D virtual environments.
- Haptic rendering and applications in human computer interaction, collaborative virtual reality and augmented reality applications in engineering, medicine and applied sciences.
- Numerical computing, development of graphics processing unit based numerical solution packages.
- Graphics processing unit programming for real time computer vision.
- Parallel processing software design patterns.
- Computational geometry and mathematical topology.
- Mathematical theory and applications of scientific simulation and visualization of rigid and elastic bodies, fluid mechanics for gas, liquid and blood flows that will be useful in engineering, applied sciences and medicine.
- Methods for machine intelligence and for more interactive and intelligent communication with computers.

REFERENCES

Advanced Micro Devices, Inc. [AMD]. (2010). *ATI RadeonTM X1550 specifications*. Retrieved September 01, 2010, from http://www.amd.com/us/products/desktop/graphics/other/Pages/x1550-specifications.aspx .

Amorim, R., Haase, G., Liebmann, M., & Santos, R. W. D. (2009). Comparing CUDA and OpenGL implementations for a Jacobi iteration. *HPCS'09, International Conference on High Performance Computing & Simulation, 2009, 22-32*. Retrieved September 20, 2009, from IEEE Xplore Digital Library Database.

Avi. (2007). *RealityPrime >> scenegraphs: Past, present, and future*. Retrieved September 01, 2010, from http://www.realityprime.com/articles.com/articles/scenegraphs-past-present-and-future#tomorrow .

Azuma, R. T. (1997). *A Survey of augmented reality*. Retrieved February 17, 2008, from CiteSeerX Database.

Baraff, D. (1989). Analytical methods for dynamic simulation of nonpenetrating rigid bodies. *Computer Graphics, 23,* (3) 223-232. Retrieved November 28, 2008 , from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.163.5683 .

Baraff, D., & Witkin, A. (1998). *Large steps in cloth simulation*. Retrieved September 11, 2009, from http://www.cs.cmu.edu/~baraff/papers/sig98.pdf .

Baraff, D. (2001). *Collision and contact – physically based modelling, SIGGRAPH 2001 course notes*. Retrieved March 21, 2010, from http://www.pixar.com/companyinfo/research/pbm2001/ .

Barakonyi, I, Psik, T., & Schmalstieg, D. (2004). Agents that talk and hit back: Animated agents in augmented reality. *ISMAR 2004 IEEE and ACM*

International Symposium on Mixed and Augmented Reality, 2004, 141-150. Retrieved March 01, 2007, from CiteSeerX Database.

Barber, C. B., Dobkin, D. P., & Huhdanha, H. (1996). The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software, 22*, (4) 469-483. Retrieved November 27, 2008 from CiteSeerX Database.

Barequet, G., Chazelle, B., Guibas, L. J., Mitchell, J. S. B., & Tal, A. (1996). BOXTREE: A hierachical representation for surfaces in 3D. *EUROGRAPHICS '96, 15* (3) 387-396. Retrieved November 24, 2008, from CiteSeerX Database.

Bathe, K. J. (1996). *Finite element procedures*. New Jersey: Prentice-Hall, Inc.

Bergen, G. V. (n.d.). *Proximity queries and penetration depth computation on 3D game objects*. Retrieved March 16, 2010, from http://www.google.com.tr/url?sa=t&source=web&cd=1&ved=0CBkQFjAA&url=http%3A%2F%2Fciteseerx.ist.psu.edu%2Fviewdoc%2Fdownload%3Fdoi%3D10.1.1.113.6708%26rep%3Drep1%26type%3Dpdf&rct=j&q=Proximity%20Queries%20and%20Penetration%20Depth%20Computation%20on%203D%20Game%20Objects&ei=tmeOTPzuBMKU4gan6eGECg&usg=AFQjCNHwI3SeLYmNt1zrMc87EL5X_au8jw .

Bergen, G. V. D. (1998). Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools, 2*. Retrieved May 17, 2010, from CiteSeerX Database.

Bergen, G. V. D. (1999). *A fast and robust GJK implementation for collision detection of convex objects*. Retrieved March 13, 2010, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.7659 .

Bergen, G. V. D. (2004). *Collision detection in interactive 3D environments*. CA: Morgan Kaufmann Publishers.

Bielser, D., Maiwald, V. A., & Gross M. H. (1999). Interactive cuts through 3 dimensional soft tissue. *EUROGRAPHICS'99, 18,* (3) 31-38. Retrieved March 31, 2010, from CiteSeerX Database.

Bielser, D., & Gross, M. H. (2002). Interactive simulation of surgical cuts. *The Eight Pacific Conference on Computer Graphics and Applications 2000 Proceedings*. Retrieved February 03, 2009, from http://ieeexplore.ieee.org/st/amp/stamp.jsp?arnumber=00883933 .

Bielser, D., Glardon, P., Teschner, M., & Gross, M. (2003). A state machine for real-time cutting of tetrahedral meshes. *11th Pacific Conference on Computer Graphics and Applications 2003 Proceedings*. Retrieved February 02, 2009, from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1238279&tag=1 .

Blanchette, J., & Summerfield, M. (2008). *C++ gui programming with Qt 4* (2nd ed.). Massachusetts: Prentice Hall.

Bolz, J., Farmer, I., Grinspun, E., & Schröder, P. (2003). Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Transactions on Graphics, 22*, 917-924. Retrieved September 20, 2009, from CiteSeerX Database.

Bradsky, G., & Kaehler, A. (2008). *Learning OpenCV computer vision with the OpenCV Library* . CA: O'Reilly Media, Inc.

Bridson, R. E. (2003). *Computational aspects of dynamics surfaces – PhD thesis*. Retrieved March 29, 2010, from http://www.cs.ubc.ca/~rbridson/ .

Brown, D., Julier, S., Baillot, Y., & Livingston, M. A. (2003). An event based data distribution mechanism for collaborative mobile augmented reality and virtual environments. *Proceedings of the IEEE Virtual Reality 2003 VR'03,* 43-52. Retrieved March, 01, 2007, from CiteSeerX Database.

Cerveri, P., Momi, E. D., Lopomo, N., Baud-Bovy, G., Barros, R. M. L., & Ferrigno G. (2007). Finger kinematic modeling and real-time hand motion estimation. *Annals of Biomedical Engineering, 35,* (11) 1989-2002. Retrieved September 10, 2010, from SpringerLink Database.

CGAL. (2009). *CGAL user and reference manual: All Parts*. Retrieved February 01, 2009, from http://www.cgal.org/Manual/ .

CGAL. (2010). *CGAL - computational geometry algorithms library*. Retrieved February 01, 2009, from http://www.cgal.org/download.html .

Comas, O, Taylor, Z. A., Allard, J., Ourselin, S, Cotin, S., & Passenger, J. (2008). Efficient nonlinear FEM for soft tissue modelling and its GPU implementation within the open source framework SOFA. *International Symposium on Computational Models for Biomedical Simulation, 2008,* 28-39. Retrieved September 10, 2009, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.156.8345 .

Cook, D. R., Malkus, D. S., & Plesha M. E. (1989). *Concepts and applications of finite element analysis* (3rd ed.). NY: John Wiley & Sons, Inc.

Cotin, S., Delingette, H., & Ayache, N. (1999). Real-time elastic deformations of soft tissues for surgery simulation. *IEEE Transactions on Visualization and Computer Graphics, 5,* (1) 62-73. Retrieved January 6, 2009, from IEEE Xplore Digital Library Database.

Coumans, E. (2009). Bullet 2.74 physics SDK manual. Retrieved October 16, 2009, from http://code.google.com/p/bullet/downloads/list .

Coumans, E. (2010). Bullet physics library. Retrieved October 16, 2009, from http://code.google.com/p/bullet/downloads/list .

Davis, E. J., Ozsoy A., Patel S., & Taufer M. (2009). *Towards large – scale molecular dynamics simulation on graphics processors*. Retrieved September 10, 2010, from http://www.nvidia.com/object/cuda_apps_flash_new.html#state=detailsOpen;aid=55b2f736-0a6b-4893-aaed-272cb5dd676d .

Deitel, H. M., Deitel, P. J., & Choffnes, D. R. (2004). *Operating systems* (3rd ed.). New Jersey: Prentice Hall.

Desbrun, M., Schröder, P., & Barr, A. (1999). *Interactive animation of structured deformable objects – technical report 034*. Retrieved July 07, 2009, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.4150&rep=rep1&type=pdf .

Devillers, O., & Guigue, P. (2002). *Faster triangle – triangle intersection tests*. Retrieved November 24, 2008, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.6.1725&rep=rep1&type=pdf .

Diestel, R. (2005). *Graph theory* (3rd ed.). Heidelberg: Springer-Verlag.

Dijkstra, E. W. (1959). A note on two problems in connection with graphs. *Numerische Mathematik, 1*, 269–271.

Dunn, F., & Parberry I. (2002). *3D math primer for graphics and game development*. Texas: Wordware Publishing, Inc.

Eberly, D. H. (2004). *Game physics*. CA: Morgan Kaufmann Publishers.

Eberly, D. (2004). *Primitive tests for collision detection*. Retrieved March 15, 2010, from http://www.cse.ttu.edu.tw/~jmchen/compg/slides/collision/taxonomy.pdf .

Eberly, D. (2008). *Intersection of convex objects: The method of seperating axes*. Retrieved November 28, 2008, from http://geometrictools.com/Documentation/M

ethodOfSeparatingAxes.pdf .

Eden, A. H., Gil, J., Hirshfeld, Y., & Yehudai, A. (1998). *Towards a mathematical foundation for design patterns*. Retrieved August 17, 2010, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.6332 .

Edmunds, M. (2010). Contact*, the Antikythera Mechanism research project*. Retrieved September 01, 2010, from http://www.antikythera-mechanism.gr/contact .

Elliot J. (2010). *Professional Graphics Controller notes*. Retrieved September 06, 2010, from http://www.seasip.info/VintagePC/pgc.html .

Engel, W. F. (2004a). *ShaderX2: Introductions & tutorials with DirectX 9*. Texas: Wordware Publishing, Inc.

Engel, W. F. (2004b). Shader *X2: Shader programming tips & tricks with DirectX 9*. Texas: Wordware Publishing, Inc.

Ericson, C. (2005). *Real time collision detection*. CA: Morgan Kaufmann Publishers.

Farias, T., Almeida, M., Teixeira, J. M., Teichrieb, V., & Kelner, J. (2008). *A high performance massively parallel approach for real time deformable body physics simulation*. Retrieved July 21, 2009, from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4685727&tag=1 .

Fernando, R. (2004). *GPU gems programmaing techniques, tips, and tricks for real-time graphics*. MA: Addison-Wesley, Pearson Education, Inc.

Fernando, R., & Kilgard, M. J. (2003). *The Cg tutorial the definitive guide to programmable real-time graphics*. MA: Addison-Wesley, Pearson Education, Inc.

Ferreira, A. J. M. (2009). *MATLAB codes for finite element analysis solids and structures*. Springer.

Fifth Dimension Technologies [5DT]. (2004a). *5DT data glove ultra series user's manual*. 5DT.

Fifth Dimension Technologies [5DT]. (2004b). *5DT HMD 800-26 series user's manual*. 5DT.

Forsyth, D. A., & Ponce, J. (2003). *Computer vision a modern approach*. NJ: Prentice Hall.

Foster, G. (2010). *GameDev.net – understanding and implementing scene graphs*. Retrieved September 01, 2010, from http://www.gamedev.net/reference/programming/features/scenegraph/default.asp .

Freeth, T., Jones, A., Steele, J. M., & Bitsakis, Y. (2008). Calendars with olympiad display and eclipse prediction on the Antikythera Mechanism. *Nature International Weekly Journal of Science, 454*, 614-617.

Fujimoto, N., (2008). Faster matrix-vector multiplication on GeForce 8800GTX. *IPDPS 2008, IEEE International Symposium on Parallel & Distributed Processing, 2009,* 1-8. Retrieved September 20, 2009, from IEEE Xplore Digital Library Database.

Fürnstahl, P., Reitinger, B., & Schmalstieg, D. (2006). Global mesh partitioning for surgical planning. *Central European Multimedia and Virtual Reality Conference, 2006*. Retrieved February 17, 2008, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.164.9922 .

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design patterns elements of reusable object-oriented software*. IN: Addison-Wesley.

Georgii, J., & Westermann, R. (2005). Mass-spring systems on the GPU. *Simulation Modelling Practice and Theory, 13*, (8) 693-702. Retrieved September 21, 2009, from ScienceDirect Database.

Gilbert, E. G., Johnson, D. W., & Keerthi S. S. (1988). A fast procedure for computing the distance etween complex objects in three dimensional space. *IEEE Journal of Robotics and Automation, 4* (2) 193-203. Retrieved March 13, 2010, from IEEE Xplore Digital Library Database.

Glencross, M., Otaduy, M., & Chalmers, A. (2005). Interaction in distributed virtual environments. *EUROGRAPHICS 2005*. Retrieved September 01, 2009, from http://isg.cs.tcd.ie/eg2005/T8.html .

Goose, S., Sudarsky, S., Zhang, X., & Navab, N. (2002). SEAR: Towards a mobile and context-sensitive speech-enabled augmented reality. *IEEE International Conference on Multimedia and Expo ICME'02 Proceedings, 2002, 1*, 849-852. Retrieved March 01 2007, from IEEE Xplore Digital Library Database.

Gottschalk S., Lin M. C., & Manocha D. (1996). OBBTree: A hierachical structure for rapid interference detection. *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, 1,* 171-180. Retrieved December 02, 2008, from http://portal.acm.org/citation.cfm?id=237244 .

Göddeke, D., Buijssen, S. H. M., Wobker, H., & Turek, S. (2009). GPU cluster computing for finite element applications. *SIAM Conference on Computational Science and Engineering Emerging Manycore Architectures Minisymposium.* Retrieved September 14, 2010, from http://people.maths.ox.ac.uk /~gilesm/SIAM_CSE/goeddeke.pdf .

GPGPU.org. (2010). *GPGPU.org:: General-purpose computation on graphics processing units*. Retrieved September 01, 2010, from www.gpgpu.org .

Grady, S. M. (2003). *Virtual Reality: Simulating and enhancing the world with computers*. NY: Facts On File, Inc.

Groen D., Harfst S., & Zwart S. P. (2009). *The living application: A self-organizing system for complex grid tasks*. Retrieved September 10, 2010, from http://www.nvidia.com/object/cuda_apps_flash_new.html#state=detailsOpen;aid=a0d09099-5643-406d-9d4a-9e7053425028 .

Guest. (2010). Offset mapping or parallax effect [2004, Feb 09]. Message posted to http://www.ogre3d.org/forums/viewtopic.php?f=3&t=343 2&start=0 .

Hamam, A., Nourian, S., El-Far, N. R., Malric, F., Shen, X., & Georganas, N. D., (2006). *A distributed, collaborative and haptic-enabled eye cataract surgery application with a user interface on desktop, stereo desktop and immersive displays*. Retrieved July 28, 2009, from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4062520&tag=1 .

Harada, Y., Nazir, N., Shiote, Y., & Ito, T. (2006). Human-machine collaboration system for fine assembly process. *International Joint Conference SICE-ICASE, 2006,* 5355-5360. Retrieved March 01, 2007, from IEEE Xplore Digital Library Database.

Heidelberger, H., Teschner, M., & Gross, M. (2003). *Volumetric collision detection for deformable objects*. Retrieved October 4, 2009, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.2297&rep=rep1&type=pdf .

Heidelberger, B., Teschner, M., Keiser, R., Müller, M., & Gross, M. (2004). *Consistent penetration depth estimation for deformable collision response*. Retrieved April 01, 2010, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.130.5656 .

Heidelberger, B., Teschner, M., Keiser, R., Müller, M., & Gross, M. (2004). *Consistent penetration depth estimation for deformable collision response*. Retrieved April 01, 2010, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi =10.1.1.130.5656 .

Heim, M. (1998). *Virtual realism*. New York: Oxford University Press.

Held, M., Klosowski, J. T., & Mitchell, J. S. B. (1995). Evaluation of collision detection methods for virtual reality fly-throughs. *In Canadian Conference on Computational Geometry,* 205-210. Retrieved November 25, 2008, CiteSeerX Database.

Held, M. (1998). ERIT – A collection of efficient and reliable intersection tests. *Journal of Graphics Tools, 2*, 25-44. Retrieved November 28, 2008, from CiteSeerX Database.

Hermann, E., Faure, F., & Raffin, B. (2008). *Ray-traced collision detection for deformable bodies*. Retrieved October 5, 2009, from http://citeseerx.ist.psu.edu/v iewdoc/summary?doi=10.1.1.141.7858 .

Hoff III, K. E., Zaferakis, A., Lin, M, & Manocha, D. (n.d.). *Fast 3-D geometric proximity queries between rigid and deformable models using graphics hardware acceleration*. Retrieved November 24, 2008, from http://citeseerx.ist.ps u.edu/viewdoc/download?doi=10.1.1.111.9155&rep=rep1&type=pdf .

Huang, J., Ponce, S. P., Park, S. I, Cao, Y., & Quek, F. (2008). GPU accelerated computation for robust motion tracking using the CUDA framework. *VIE2008, 5th International Conference on Visual Information Engineering, 2008,* 437-442. Retrieved September 20, 2009, from IEEE Xplore Digital Library Database.

Hubbard, P. M. (1995). *Collision detection for interactive graphics applications –*

PhD thesis. Retrieved November 24, 2008, from ftp://ftp.cs.brown.edu/pub/techreports/95/cs95-08.pdf .

Hutton, D. V. (2004). *Fundementals of finite element analysis*. NY: McGraw-Hill Companies, Inc.

INTEL. (2010). *INTEL® Core™ i7 Processor Extreme Edition*. Retrieved September 06, 2010, from http:www.intel.com/products/processor/corei7ee/index.htm .

Jacob, M. (2010). *OGRE – Open source 3D graphics engine*. Retrieved October 07, 2009, from http://www.ogre3d.org/download/source .

James, D. L. (2008). *Multi-sensory physics and user interaction SIGGRAPH2008 course*. Retrieved April 05, 2010, from http://www.matthiasmueller.info/realtimephysics/index.html .

Jang, H., Park, A., & Jung, K. (2008). Neural network implementation using CUDA and OpenMP. *DICTA '08, Digital Image Computing Techniques and Application, 2008,* 155-161. Retrieved September 20, 2009, from IEEE Xplore Digital Library Database.

Januszewski, M., & Kostur M. (2009). *Accelerating numerical solutions of stochastic differential qquations with CUDA*. Retrieved September 10, 2010, from http://www.nvidia.com/object/cuda_apps_flash_new.html#state=detailsOpen;aid=2234c230-375e-11de-8a39-0800200c9a66 .

Jiménez, P., Thomas, F., & Torras C. (2001). 3D collision detection : A survey. *Computers & Graphics, 25,* (2001) 269-285. Retrieved November 28, 2008, from http://www.stanford.edu/class/cs277/schedule/assets/Jimenez2001.pdf .

Junker, G. (2006). *Pro OGRE 3D programming*. CA: Apress.

Karabassi, E. A., Papaioannou, G., & Theoharis, T. (1999). Intersection test for collision detection in particle systems. *Journal of Graphics Tools, 4*. Retrieved December 01, 2009, from CiteSeerX Database.

Kataria, M. (n.d.). *Force feedback and collision detection of 3D primitives in virtual environments*. Retrieved March 15, 2010, from http://www.ee.iitb.ac.in/student/~kataria/data/Academics/HapticsInVE-report.pdf .

Kato, H. & Billinghurst, M. (2006). *ARToolkit*. Retrieved October 01, 2007, from http://www.hitl.washington.edu/artoolkit/documentation/ .

Kaufmann, P., Martin, S., Botsch, M., & Gross, M. (2008). Flexible simulation of deformable models using discontinuous Galerkin FEM. *Journal of Graphical Models – Special Issue of ACM SIGGRAPH / Eurographics Symposium on Computer Animation 2008, 71*, (4) 153-167.

Kaufmann, P., Martin, S., Botsch, M., Grinspun, E., & Gross, M. (2009). Enrichment textures for detailed cutting of shells. *ACM Transactions on Graphics, 28,* (3) 1-10.

Khalil, H. K. (2002). *Nonlinear systems* (3rd ed.). NJ: Prentice-Hall, Inc.

Kilgard, M. J. (1999). *Improving shadows and reflections via the stencil buffer*. Retrieved September 08, 2010, from http://developer.nvidia.com/object/Stencil_Buffer_Tutorial.html .

Kirk, (n.d.). *The future of massively parallel and GPU computing*. Retrieved September 01, 2010, from http://www.greatlakesconsortium.org/events/GPUMulticore/kirk.pdf .

Kirk, D. B., & Hwu, W. W. (2010). *Programming massively parallel processors a hands on approach*. MA: Elsevier Inc., Morgan Kaufmann.

Knuth, D. E. (1973). *Sorting and searching the art of computer programming*. Boston: Addison – Wesley.

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society, 7,* (1) 48-50.

Krüger, J., & Westermann, R. (2003). *Linear algebra operators for GPU implementation of numerical algorithms*. Retrieved September 20, 2009, from http://wwwcg.in.tum.de/Research/Publications/LinAlg .

Lacoursiére, C. (n.d.). *Splitting methods for dry frictional contact problems in rigid multibody systems: Preliminary performance results*. Retrieved March 28, 2010, from http://www.ep.liu.se/ecp/010/004/ecp01004.pdf .

Lahabar, S., & Narayanan, P. J. (2009). Singular value decomposition on GPU using CUDA. *IPDPS 2009, IEEE International Symposium on Parallel & Distributed Processing, 2009,* 1- 10. Retrieved September 20, 2009, from IEEE Xplore Digital Library Database.

Lander, J. (1999a). Devil in the blue-faceted dress: Real-time cloth animation. *Game Developer, May 1999*, 17-21.

Lander, J.(1999b). Apply the force to get the right amount of friction. *Game Developer, August 1999*, 19-24.

Lander, J.(1999c). Collision response: Bouncy, trouncy, fun. *Game Developer, March 1999*, 15-19.

Larsson, T., & Möller, T. A. (2001). Collision detection for continuously deforming bodies. *The Visual Computer, 19,* (2-3) 164-174. Retrieved November 28, 2008, from SpringerLink Database.

Leiterman, J. C. (2004). *Learn vertex and pixel shader programming with DirectX® 9*. Texas: Wordware Publishing, Inc.

Li, B., Wang, C., Li, Z., & Chen, Y. (2009). A practical method for real time ocean simulation. *4th International Conference on Computer Science & Education, 2009,* 742-747. Retrieved September 20, 2009, from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05228129 .

Lin, M. C., & Gottschalk, S. (1998). Collision detection between geometric models: A survey. In *Proc. of IMA Conference on Mathematics of Surfaces,* 37-56. Retrieved November 24, 2008, from CiteSeerX Database.

Liu, Y., & De, S. (2008). *CUDA-based real time surgery simulation*. Retrieved July 21, 2009, from http://www.acor.rpi.edu/research/CUDA.pdf .

Liu, Y., Jiao, S., Wu, W., & De, S. (2008). GPU accelerated fast FEM deformation simulation. *APCCAS 2008 IEEE Asia Pacific Conference on Circuits and Systems, 2008,* 606-609. Retrieved September 21, 2009, from IEEE Xplore Digital Library Database.

Luebke, D., & Humphreys, G. (2007). How GPUs work. *Computer, February 2007*, 126-130.

Marathe, A. R., Carey, H. L., & Taylor, D. M. (2007). Virtual reality hardware and graphic display options for brain-machine interfaces. *Journal of Neuroscience Methods, 167*, (1) 2-14. Retrieved January 09, 2008, from ScienceDirect Database.

Marsaglia, G. (1996). *Diehard random number testing*. Retrieved October 2002, from http://stat.fsu.edu/~geo/diehard.html .

Martz, P. (2007). *OpenSceneGraph quick start guide*. California: Computer Systems Development Coorporation.

McShaffry, M. & et al. (2009). *Game coding complete* (3rd ed.). MA: Course Technology PTR.

Mitchell, J. (2004). *Light shafts rendering shadows in participating media*. Retrieved January 10, 2010, from http://developer.amd.com/media/gpu_assets/ Mitchell_LightShafts.pdf .

Mizuno, Kato, & Nishida (2004). Outdoor augmented reality for direct display of hazard information. *SICE 2004 Annual Conference 2004, 1,* 831-836. Retrieved March 01, 2007, from IEEE Xplore Digital Library Database.

Möller T. A., Haines E., & Hoffman N. (2008). *Real time rendering*. (3rd ed.). MA: A K Peters, Ltd.

Morefield, R., & Malloy, B. (2007). *3D game development tutorials using SDL and OSG*. Retrieved October 01, 2008, from http://www.cs.clemson.edu/~malloy/ courses/3dgames-2007/tutor/ .

Mosegaard, J., Herborg, P., & Sørensen, S. (2005). *A GPU accelerated spring mass system for surgical simulation*. Retrieved September 21, 2009 from, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.2423&rep=rep1&type=pdf .

Möller, T. (1997). A fast triangle-triangle intersection test. *Journal of Graphics Tools, 2,* 25-30. Retrieved December 11, 2008, from CiteSeerX Database.

Möller, T. A. (2001). *Fast 3-D triangle-box overlap testing*. RetrievedNovember 24, 2008, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.4803 &rep=rep1&type=pdf .

Müller, M., Heidelberger, B., Hennix, M., & Ratcliff, J. (2006). *Position based dynamics*. Retrieved April 06, 2010, from http://www.matthiasmueller.info/publi cations/posBasedDyn.pdf .

Müller, M., James, D., Stam, J., & Thuerey, N. (2008b). *Real time physics SIGGRAPH2008*. Retrieved April 05, 2010, from http://www.matthiasmueller.inf o/realtimephysics/index.html .

Müller, M., McMillan, L., Dorsey, J., & Jagnow, R., (2001). Real-time simulation of deformation and fracture of stiff materials. *EUROGRAPHICS 2001 Computer Animation and Simulation Workshop,* 99-111. Retrieved April 06, 2010, from CiteSeerX Database.

Müller, M., Stam, J., & James, D. (2008a). Real time physics class notes. Retrieved August 15, 2009, from http://www.matthiasmueller.info/realtimephysics/index.ht ml .

Nealen, A., Müller, M., Keiser, R., Boxermann, E., & Carlson, M. (2005). Physically based deformable models in computer graphics. *Computer Graphics Forum, 25,* (4) 809-836. Retrieved April 06, 2010, from http://www.matthiasmueller.info/rea ltimephysics/index.html .

Neumann J. V. (1945). *First draft of a report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania.

Nguyen, H. (2007). GPU Gems 3. MA: Addison-Wesley, Pearson Education, Inc.

Nielsen, M. B., & Cotin, S. (1996). Real-time volumetric deformable models for surgery simulation using finite elements and condensation. *Computer Graphics Forum,* 57-66. Retrieved January 6, 2009, from CiteSeerX Database.

NVIDIA (2008). *NVIDIA PhysX 2.8 documentation*. Retrieved August 01, 2009, from http://developer.nvidia.com/object/physx_downloads.html .

NVIDIA (2009b). *NVIDIA PhysX physics simulation for developers*. Retrieved July 17, 2009, http://developer.nvidia.com/object/physx_downloads.ht ml .

NVIDIA (2009a). *NVIDIA CUDA™ programming guide (Version 2.2.1, 26.05.2009)*. Retrieved August 01, 2009, from http://developer.nvidia.com/object/ cuda_2_2_downloads.html .

NVIDIA. (2010). *GeForce GTX 295*. retrieved September 06, 2010, from http://www.nvidia.com/object/product_geforce_gtx_295_us.html .

OpenSceneGraph. (2010). *OpenSceneGraph web site*. Retrieved February 10, 2009, from http://www.openscenegraph.org/projects/osg/wiki/Downloads .

Otaduy, M., Tamstorf, R., Steinemann, D., & Gross, M. (2009). Implicit contact handling for deformable objects. *EUROGRAPHICS 2009 Journal Compilation, 28,* (2). Retrieved August 31, 2009, from http://geom.mi.fu-berlin.de/res/teaching/ss09/sem_geometrieverarbeitung/material/paper/otaduy_implicit_contact_handling.pdf .

Pathomaree, N., & Charoenseang, S. (2005). Augmented reality for skill transfer in assembly task. *IEEE International Workshop on Robots and Human Interactive Communication, 2005,* 500-504. Retrieved March 01, 2007, from IEEE Xplore Digital Library Database.

Perepelkin, E., Smirnov, V., & Vorozhtsov S. (2009). *Beam dynamic calculation by NVIDIA® CUDA technology*. Retrieved September 10, 2010, from http://www.nvidia.com/object/cuda_apps_flash_new.html#state=detailsOpen;aid=d508073b-38bf-4fc7-b99b-1ad6ff71b868 .

Pharr, M., & Fernando R. (Eds). (2005). *GPU gems 2 programming techniques for high-performance graphics and general-purpose computation*. NJ:Addison Wesley, Pearson Education, Inc.

Piekarski, W., & Thomas, B. H. (2003). Tinmith – mobile outdoor augmented reality modelling demostration. *ISMAR'03 Proceedings of the Second IEEE and ACM International Symposium on Mixed and Augmented Reality, 2003,* 317-318. Retrieved March 01, 2007, from http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/p/Piekarski:Wayne.html .

Polhemus. (2009). *Fastrak user manual (OPM00PI002 REV. F. June 2009).* Colchester, Vermont U.S.A: Polhemus.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery B. P. (2007). *Numerical recipes the art of scientific computing* (3rd ed.). New York: Cambridge University Press.

Prim, C. (1957). Shortest connections networks and some generalizations. *Bell System Technical Journal 36,* (6) 1389–1401.

Provot, X. (1996). Deformation constraints in a mass-spring model to describe rigid cloth behavior. *In Graphics Interface,* 147 - 154. Retrieved July 07, 2009, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.4040 .

Qin, J., Pang, W. M., Chui, Y. P., Wong, T. T., & Heng, P. A. (2008). A novel modelling framework for multilayered soft tissue deformation in virtual

orthopedic surgery. *Journal of Medical Systems, 34*, (3) 261-271. Retrieved September 10, 2009, from SpringerLink Database.

Ranzuglia, G., Cignoni, P., Ganovelli, F., & Scopigno R. (2006). Implementing mesh-Based approaches for deformable objects on GPU. *Fourth Eurographics Italian Chapter, 2006,* 213-218. Retrieved September 21, 2009, from http://vcg.isti.cnr.it/Publications/2006/RCGS06/ .

Rasmusson, A., Mosegaard, J., & Sørensen, S. (2008). Exploring parallel algorithms for volumetric mass-spring-damper models in CUDA. *Lecture Notes in Computer Science 2008,* 5104/2008, 49-58. Retrieved July 30, 2009, from SpringerLink Database.

Reitinger, B., Bornik, A., Beichel, R., & Schmalstieg, D. (2006). Liver surgery planning using virtual reality. *IEEE Computer Graphics and Applications, 26,* (6) 36-47. Retrieved February 17, 2008, from IEEE Xplore Digital Library Database.

Reitinger, B., Zach, C., & Schmalstieg, D. (2007). Augmented reality scouting for interactive 3D reconstruction. *Proceedings of IEEE Virtual Reality 2007,* 219-222. Retrieved February 17, 2008, from http://www.vrvis.at/publications/PB-VRVis-2007-006 .

Reitmayr, G., & Schmalstieg, D. (2001). Mobile collaborative augmented reality. *In Proceedings ISAR 2001*. Retrieved March 01, 2007, from http://studierstube.icg .tu-graz.ac.at/projects/mobile/ .

Reitmayr, G., & Schmalstieg, D. (2004). Collaborative augmented reality for outdoor navigation and information browsing. *In Prooceedings of the Symposium on Location Based Services and TeleCartography 2004,* 31-41. Retrieved October 06, 2007, from CiteSeerX Database.

Reitmayr, G., & Schmalstieg, D. (2007). *Scalable techniques for collaborative outdoor augmented reality*. Retrieved October 06, 2007, from http://www.ims.tu wien.ac.at/media/documents/publications/reitmayrIsmar04.pdf .

Rhee, T., Neumann, U., & Lewis, J. P. (2006). Human hand modelling from surface anatomy. *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2006*. Retrieved September 05, 2010, from http://www.google.com.tr/url?sa=t&source=web&cd=1&ved=0CB0QFjAA&url=http%3A%2F%2Fciteseerx.ist.psu.edu%2Fviewdoc%2Fdownload%3Fdoi%3D10.1.1.119.6648%26rep%3Drep1%26type%3Dpdf&rct=j&q=Human%20Hand%20Modelling%20from%20Surface%20Anatomy&ei=Qc_sTI6MK4PwsgbjotyGDw&usg=AFQjCNEHKE0rAbUlsqXV2Ap2qB0GgXSgFQ&cad=rja .

Roberts, M., Packer, J., Sousa M. C., & Mitchell J. R. (2010). *A work- efficient GPU algorithms for level set segmentation*. Retrieved September 10, 2010, from http://www.nvidia.com/object/cuda_apps_flash_new.html#state=detailsOpen;aid=f695686e-a314-4d4c-a222-7a1e88c753f3 .

Rogers, D. F., & Adams, J. A. (1990). *Mathematical elements for computer graphics* (2nd ed.). Singapore: McGraw-Hill.

Rost R. J., & Kane B. L. (2010). *OpenGL shading language* (3rd ed.). MA: Addison-Wesley, Pearson Education, Inc.

Rugh, W. J. (1996). *Linear system theory* (2nd ed.). NJ: Prentice-Hall, Inc.

Seddon, C. (2005). *OpenGL game development*. Texas: Wordware Publishing, Inc.

SensAble Technologies, Inc. (2008). Specifications for the PHANTOM Omni® haptic device. MA: SensAble Technologies, Inc.

Sewell, G. (2005). *The numerical solution of ordinary and partial differential equations* (2nd ed.). New Jersey: John Wiley & Sons, Inc.

Smith, D. (2004). *Light shafts photo – Duncan Smith photos at pbase.com.* Retrieved November 20, 2010, from http://www.pbase.com/duncansmith/image/37466541 .

Spampinato, D. G., Elster, A. C. (2009). Linear optimization on modern GPUs. *IPDPS 2009, IEEE International Symposium on Parallel & Distributed Processing, 2009,* 1-8. Retrieved September 20, 2009, from IEEE Xplore Digital Library Database.

Srinivasan, M. A. (n.d.) *What is haptics?* Retrieved September 01, 2010, from http://touchlab.mit.edu .

Stam, J. (2009). Nucleus: Towards a unified dynamics solver for computer graphics. *11th IEEE International Conference on Computer-Aided Design and Computer Graphics, 2009,* 1-11. Retrieved April 05, 2010, from IEEE Xplore Digital Library Database.

Steinemann, D., Harders, M., Gross, M., & Szekely, G. (2006). Hybrid cutting of deformable solids. *Virtual Reality Conference 2006*. Retrieved August 31, 2009, from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1667 624 .

Steinemann, D., Otaduy, M. A., & Gross, M. (2006). Fast arbitrary splitting of deforming objects. *EUROGRAPHICS / ACM SIGGRAPH Symposium on Computer Animation 2006*. Retrieved August 31, 2009, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.1089 .

Strang, G. (1986). *Introduction to applied mathematics*. MA: Wellesley- Cambridge Press.

Stroustrup, B. (2000). *The C++ programming language* (3rd ed.). AT&T Labs. Murray Hill, New Jersey: MA: Addison – Wesley.

Stroustrup, B. (2008). *Programming – principles and practice using C++*. Addison – Wesley.

Tan, T. S., Chong, K. F., & Low, K. L. (1999). Computing bounding volume hierarchies using model simplification. *1999 ACM Symposium on Interactive 3D Graphics,* 63-70. Retrieved November 25, 2008, from CiteSeerX Database.

Tatarchuk, N. (2006). *Artist-directable real-time rain rendering in city environments, SIGGRAPH 2006 advanced real-time rendering in 3D graphics and games course notes.* Retrieved September 11, 2010, from http://developer.amd.com/media/gpu_assets/Tatarchuk-Rain.pdf .

Tatarchuk, N., & Shopf, J. (2007). *Real -time medical visualization with FireGL. SIGGRAPH 2007, AMD Technical Talk.* Retrieved September 11, 2010, from http://developer.amd.com/media/gpu_assets/MedicalVisualization.pdf .

Tekalp, A. M. (1995). *Digital video processing*. NJ: Prentice Hall.

Taylor, Z. A., Cheng, M., & Ourselin, S. (2008). High-speed nonlinear finite element analysis for surgical simulation using graphics processing units. *IEEE Transactions on Medical Imaging, 27*, (5) 650-663.

Teschner, M., Heidelberger, B., Müller, M., Pomeranets, D., & Gross, M. (2003). *Optimized spatial hashing for collision detection of deformable objects*. Retrieved March 30, 2010, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.5881&rep=rep1&type=pdf .

Teschner, M., Kimmerle, S., Heidelberger, B., Zachmann, G., Raghupathi, L., Fuhrmann, A., Cani M. P., Faure, F., Thalmann, N. M., Strasser, W., & Volino, P.

(2004). Collision detection for deformable objects. *EUROGRAPHICS Association, 23,* (3) 1-22.

The SOFA Team at INRIA Grenoble. (2009). *Simulation open framework architecture – The SOFA project.* Retrieved August 05, 2009, from http://www.sofa-framework.org/ .

The Sofa Team. (2008). *Sofa documentation*. Retrieved August 05, 2009, from http://www.sofa-framework.org/manual .

Thuerey, N. (2008). *Real-time physics part III: Fluids*. Retrieved April 05, 2010, from http://www.matthiasmueller.info/realtimephysics/index.html .

Tropp, O., Tal, A., & Shimshoni, I. (2005). *A fast triangle to triangle intersection test for collision detection.* Retrieved November 25, 2008, from http://mis.hevra.haifa.ac.il/~ishimshoni/papers/TroppTalShimshoni.pdf .

Turing A. M. (1936). *On computable numbers, with an application to the Entscheidungsproblem*. The Graduate College, Princeton University, New Jersey, U.S.A.

United States National Library of Medicine National Institutes of Health, [NLM]. (2010). *The National Library of Medicine's Visible Human Project*. Retrieved September 01, 2010, from http://www.nlm.nih.gov/research/visible/ .

University of North Carolina at Chapel Hill Department of Computer Science, (2004). *Fast penetration depth computation*. Retrieved March 16, 2010, from http://www.cs.unc.edu/Research/ProjectSummaries/penetration.pdf .

Vallino, J., & Brown, C. (1999). Haptics in augmented reality. *IEEE International Conference on Multimedia Computing and Systems 1999, 1,* 195-200. Retrieved March 01, 2007, from IEEE Xplore Digital Library Database.

Velamparambil, S., Cormier, S. M., Perry, J., Lemos, R., Okoniewski, M., & Leon J. (2008). GPU Accelerated Krylov Subspace Methods for Computational Electromagnetics. *EuMC 2008, 38th European Microwave Conference, 2008,* 1312-1314. Retrieved September 20, 2009, from IEEE Xplore Digital Library Database.

Viola, P., & Jones, M. J. (2004). Robust real-time face detection. *International Journal of Computer Vision, 57*, (2004) 137-154.

Vlack, K., & Tachi, S. (2001). *Fast and accurate spacio – temporal intersection detection with GJK algorithm.* Retrieved March 13, 2010, from http://www.vrsj.org/ic-at//papers/01079.pdf .

Wang, P., Becker, A. A., Jones, I.A., Glover A. T., Benford, S. D., Greenhalg, C. M., & Vloeberghs, M. (2007). Virtual reality simulation of surgery with haptic feedback based on the boundary element method. *ELSEVIER Computer and Structures, 85*, (2007) 331-339.

White, S., & Feiner, S., & Kopylec, J. (2006). Virtual vouchers: Prototyping a mobile augmented reality user interface for botanical species identification. *3DUI 2006 IEEE Symposium on 3D User Interfaces, 2006,* 119-126. Retrieved March 01, 2007, from IEEE Xplore Digital Library Database.

Wikipedia. (2010c). *Singleton (mathematics): Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Singleton_%28mathematics%29 .

Wikipedia. (2010s). *Radeon R520: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, http://en.wikipedia.org/wiki/Radeon_R520 .

Wikipedia. (2010a). *Scene graph: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Scene_graph .

Wikipedia. (2010p). *Shader:- Wikipedia, the free encyclopedia*. Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Programmable_shader .

Wikipedia. (2010m). *Colossus Compute:r – Wikipedia, the free encyclopedia*. Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Colossus_computer .

Wikipedia. (2010o). *Professional Graphics Controller: Wikipedia, the free encyclopedia*. Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Professional_Graphics_Controller

Wikipedia. (2010g). *Antikythera Mechanism: Wikipedia, the free encyclopedia*. Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Antikythera_mechanism .

Wikipedia. (2010i). *The Difference Engine: Wikipedia, the free encylopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Difference_engine .

Wikipedia. (2010e). *Facade: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Facade .

Wikipedia. (2010d). *C++ classes: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/C%2B%2B_classes .

Wikipedia. (2010l). *Z3 (computer): Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Zuse_Z3 .

Wikipedia. (2010r). *Dither: Wikipedia, the free encyclopedia.* Retrieved September 08, 2010, from http://en.wikipedia.org/wiki/Dither .

Wikipedia. (2010k). *Turing Completeness: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Turing-complete .

Wikipedia. (2010t). *GeForce 200 series: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/GeForce_200_Series .

Wikipedia. (2010b). *Singleton pattern: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Singleton_pattern .

Wikipedia. (2010f). *Computer: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Computer .

Wikipedia. (2010j). *The Analytical Engine: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/Analytical_engine .

Wikipedia. (2010n). *ENIAC: Wikipedia, the free encyclopedia.* Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/ENIAC .

Wikipedia. (2010h). *Charles Babbage: Wikipedia, the free encyclopedia*. Retrieved September 01, 2010, from http://en.wikipedia.org/wiki/C harles_Babbage .

Wikipedia. (2010s). *Graphics processing unit: Wikipedia, the free encyclopedia*. Retrieved November 02, 2010, from http://en.wikipedia.org/wiki/Graphics_processing_unit .

Wikipedia. (2010t). *Gimbal lock: Wikipedia, the free encyclopedia*. Retrieved

November 20, 2010, from http://en.wikipedia.org/wiki/Gimbal_lock .

Witkin, A., & Baraff, D. (2001). *Physically based modeling, differential equation basics*. Retrieved September 10, 2009, from http://www.cs.cmu.edu/~baraff/sigcourse/notesb.pdf .

Wojtan, C., Thuerey, N., Gross, M., & Turk, G. (2009). Deforming meshes that split and merge. *ACM Transactions on Graphics, 28*, (3) 1-10.

Wright, R. S. J., Lipchak, B., & Haemel, N. (2007). *OpenGL superbible comprehensive tutorial and reference* (4th ed.). MA: Addison-Wesley.

Yan, Z., Gu, L., Huang, P., Lv, S., Yu, X., & Kong, X. (2007). So*ft tissue deformation simulation in virtual surgery using nonlinear finite element method*. Retrieved July 04, 2009 from http://ieeexplore.ieee.org/xpls/abs_all.jsp?tp=&arnumber=4353120&tag=1 .

Yu, R., Chiang, P., Chen, W., Zheng, J., Cai, Y., Ye, X., Zhang, S., Zhang, Y. & Mak K. H. (2009). A framework for GPU-accelerated virtual cardiac intervention. *The International Journal of Virtual Reality, 8,* (1) 37-41.

APPENDICES

The followings are submitted in the DVD in the pocket attached to the backcover of this thesis. The detailed information regarding the directory structure can be found in *"README_Directory_Structure.txt"* in the DVD.

1. All the C, VC++, Cg, HLSL codes developed using Microsoft Visual Studio 2005 in the scope of this M.Sc thesis. These are 3-D interactive and immersive virtual reality application for medical simulations; the haptic rendering application and the augmented reality application respectively.
2. Bullet physics library with the necessary code modifications for the M.Sc thesis.
3. Ogre3D real time graphics engine with the necessary code modifications for the M.Sc thesis.
4. Qt Toolkit used for the user interface development in the M.Sc thesis period.
5. Microsoft DirectX 9.0c was used during the software development. It should be downloaded from its web site. NVIDIA CUDA Library and NVIDIA PhysX Library can be downloaded from NVIDIA Web Site.
6. 3-D model meshes used in the developed software during this M.Sc thesis.
7. NVIDIA Cg used for GPU programming for graphics during this M.Sc thesis.
8. NVIDIA Texture Tools and Photoshop plug-ins.
9. SOFA Library used for research and practice oriented purposes.
10. OpenCV library used in the augmented reality application developed during the M.Sc thesis period.
11. ARToolkit library used in the augmented reality application developed during the M.Sc thesis period
12. OpenSceneGraph library used in the augmented reality application developed during the M.Sc thesis period.