A GPU-BASED INTEGRATED APPROACH TO SIMULATION FOR DEFORMABLE SURFACE MESHES

by

Vidya Kotamraju

B.E., Goa University, 2001

THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE
in the School
of
Engineering Science

SIMON FRASER UNIVERSITY
Fall 2008

APPROVAL

Name:	Vidya Kotamraju
Degree:	Master of Applied Science
Title of thesis:	A GPU-based Integrated Approach To Simulation for Deformable Surface Meshes

Examining Committee: Dr. Andrew Rawicz
Professor
Chair

Dr. John Dill
Senior Supervisor
Professor

Dr. Shahram Payandeh
Supervisor
Professor

Dr. Tom Calvert
Internal Examiner
Professor, Graduate Chair

Date Defended/Approved: 02 - Dec - 2008

Abstract

Surgical techniques have evolved from direct hands-on maneuvers to indirect minimally-invasive procedures, learning which involves using virtual simulation environments with standardized exercises typically comprising scene representation, collision-detection, force feedback, and rendering. Key challenges are the need to improve speed and realism of the simulation while being run on consumer-grade computing platforms.

This thesis aims at overcoming these challenges by developing an algorithm to utilize the Graphics Processing Unit (GPU) as a parallel processor. The approach presented consists of three phases: off-line surface wrapping, implicit integration for deformation, and tactile feedback.

A prototype implementation using the NVIDIA GeForce 7600GS is presented. Interactive surface wrapping models a cloth-like surface into a closed mesh while the deformation phase is a GPU-based parallelized Implicit Euler method. Point-based haptic interaction with virtual coupling provides tactile feedback. The results show signicant speedups, upto a factor of 6.5 times, for the GPU-based simulation over the CPU.

To the Supreme Spirit - the heroic in man.

"Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

- *Lewis Carroll,* THROUGH THE LOOKING-GLASS AND WHAT ALICE FOUND THERE, *1871*

Acknowledgments

I am grateful to Prof.John Dill and Prof. Shahram Payandeh for introducing me to the exciting area of Virtual Reality. I thank both of them for their invaluable guidance, technical insight and support. I also appreciate all their patience during many unproductive periods of my thesis due to courses, teaching assistantship, a long internship and a full-time job. A special thanks to Prof.Dill for allowing me to use the SIAT (Surrey Interactive Arts and Technology) graduate lab facility and to Prof. Payandeh for providing me the haptic device for my work. Reviewing an unpolished manuscript is demanding work and doing it under time constraints only makes it harder. I thank Prof. Dill for reviewing my work over several revisions.

I feel fortunate to have had an opportunity to work in an exciting areas in Computer Graphics, that of General Purpose GPU (GPGPU). I thank Prof.Payandeh for initiating the idea and Prof.Dill for leading me through it. I thank Prof.Richard Zhang for his comprehensive course in Computer Graphics which set the foundations of my understanding of it. I also thank Prof.Tom Calvert for critically reviewing this thesis, and Prof. Andrew Rawicz for chairing the defense. Thanks also to Raj Pabla and Judi Fraser, our graduate secretaries, for making the paperwork a cakewalk.

I thoroughly enjoyed my years at Simon Fraser University, made possible by PRECARN Scholars Program and the steadfast support of my supervisors and family. I thank all my colleagues at the Experimental Robotics Lab and SFU SIAT Graduate Lab for keeping me company during the long days and nights - Paul, Richard, Mavis, Fuhan, Haibo, Tai, Yuan and Ai. Special thanks to Nasim Vafai and Shilpi Rao for being such great friends - I cherish our wonderful times together.

I am indebted to my grand-mom, Sita, and grand-uncle, Narasimha Rao, whose love and values paved the way for a privileged upbringing and to my parents, Manju and Rao, for their unconditional love and unflinching support to pursue my goals in life. I cannot thank my brother Vinay enough for showing me what it is to be a free spirit. He is also instrumental in inspiring me to bring this thesis to its logical conclusion.

Loads of love to my bestest friend and indulgent husband, Bhushan, for being my biggest critic and doing all the chores!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter places the work described in this thesis in perspective with previous research in the area. Motivational arguments and a summary of the forthcoming chapters are presented.

1.1 Motivation

The philosophy behind this thesis is succinctly summarized in the following quote by William Osler in the Principles and Practice of Medicine [86],
"To learn medicine without books is to sail an uncharted sea,
While to learn medicine only from books, is to not go to sea at all"

For over a century, medical education has been based on the Halstedian system involving standard residency programs with learning through real-life patient encounters. This system bases technical and cognitive performance assessment on the subjective impression of the assessor. Apart from such informal validation, the efficacy, economics, ethics, and extent of responsibility in the system have been subject to scrutiny [59][56][26][11]. In order to overcome these challenges, medical educators proposed criteria for a competency-based curriculum. It was determined that succesive mastery of skills would be needed to develop technical expertise while standardized assessment tools would enable objective evaluation of performance. Repetitive practice sessions in a risk-free environment would help the learner develop necessary cognitive skills [75]. In order to satisfy these criteria, alternate approaches to the residency-based system were sought.

In addition, since the 1980s, surgical techniques for certain procedures evolved from direct hands-on maneuvers to indirect minimally invasive procedures, involving remote-control manipulation of minimally invasive instruments like laparoscopes and endoscopes with indirect observation of the operating site through video cameras attached to the instruments (Figure 1.1). Such techniques are known to be considerably less costly with improved patient recovery rates. The introduction of minimally-invasive techniques required surgeons to develop dexterity in motor control, hand-eye coordination and visual perception. However, with medical errors shown to be the third leading cause of death in North America, and surgery-related errors amounting to 50% of the cases [18], training and performance assessment of such techniques constituted major challenges in the field of medicine. There is a necessity to train health professionals to a greater level of skill, experience and knowledge, one way of which is through training environments.

Simulation environments satisfy many of the criteria for an alternative to the Halstedian approach and therefore are increasingly being used as a valuable addition to traditional teaching methods in medical education. The narrow visual area, limited tactile feedback, specialized tools, and video display characteristic of minimally invasive surgeries make them ideally suited to simulation. Simulators are known to improve the productivity of the trainee by providing unlimited practice and objective assessment of surgical skills. In addition, patient-specific anatomies can be included to enable surgical planning, thus reducing risk and operation time. Although, simulators train for specific surgical skills, they also help develop cognitive skills, like accurate judgement, in the context of the specific simulation [20].

Figure 1.1: Minimally-invasive surgery in progress.

A goal of current simulators is to help surgeons-to-be acquire skills needed to perform minimally-invasive surgery [53]. To do this, such simulators typically provide visual and haptic interaction through virtual training environments with standardized exercises. Simulators must support scene representation, detection of organ-tool interaction, tactile response, and scene rendering. Simulators need to maintain speed and realism while being run on consumer-grade computing platforms. For stable interactions, the update rates for graphics (i.e on-screen rendering of virtual models) need to be 30-60 Hz while the generally accepted goal for haptics (i.e providing sense of touch to user) is 500Hz-1KHz [64]. Therefore, a key challenge in many simulation environments is to achieve the required computational power to realistically deform and render the virtual models whilst maintaining stability to calculate forces for haptic feedback.

Accordingly, this thesis describes a study of the use of Graphics Processing Units (GPUs), a chip in many mid-range graphics cards, to perform the bulk of the simulation computations in parallel with the CPU. Recent advances in the speed and programmability of the GPU have enabled high-performance general-purpose and non-graphics applications [49]. The key for our framework is the suitable formulation of the training system model, design and development of scalable parallel algorithms, and efficient usage of the GPU in doing so.

1.2 Contributions

This thesis focuses on the viability of realistic surgical simulations in real-time. The approach is to exploit the inherent parallelism in the simulation equations using the GPU, and hence the thesis is
Development of a Surgical Simulation Framework using a Graphics Processing Unit.
Leading from the thesis statement, the hypothesis to test is
A GPU-based parallel framework offers the potential of providing a realistic and dynamic tactile simulation experience in real-time.

The main contribution of the thesis is the reformulation of the Implicit Euler integration method to compute the deformations of the mesh[85]. This method exploits the inherent parallelism in the resulting reformulated integration equations using the GPU. A mass-spring based surface mesh models the organ and a sparse, large linear system solved at each time step of the simulation models the deformations. This linear system computation involves large matrices in the order of $10000 - 15000$ nodes. The inverse of this large sparse matrix is an approximation, and is computed using the Neumann-polynomial method[15]. The matrix manipulations to do this calculation are implemented on the GPU.

As a pre-processing step, an interactive surface wrapping algorithm is used to obtain closed surface meshes from planar square meshes. This pre-processing is needed in order to extend prior work done by the author on square meshes to surface meshes over 3D objects while maintaining the structure of the corresponding Hessian matrices used in the integration in [85]. Although accurate methods exist to obtain such surface meshes [30], we chose a crude approximation using Discreet Autodesk 3DS Max as working towards an accurate approach is beyond the scope of this thesis. The crux of the method is to set the 2D mesh as a deformable object and simulate it such that it falls onto the 3D rigid object in a natural fashion. The method is implemented in a semi-automated fashion and the results support satisfactorily the above described GPU-based simulation. Simulation constraints and relaxation techniques are used to achieve required visual complexity with minimum wrinkles and foldovers.

A three-Degree-of-Freedom haptic device, the PHANTOM® Omni™, is used to provide force feedback, for example when a virtual tool contacts a virtual organ. Haptics rendering includes collision detection, force computation, and bidirectional communication with the haptic device. Using OpenHaptics' HDAPI, a spring-damper model is implemented with a point-based collision detection algorithm. Multi-threaded synchronization of the haptics and graphics simulation update cycles enables query of the device state and performs haptic rendering in conjunction with the GPU-based graphics thread.

Results of the techniques developed by us as a parallel framework (simulation system using the CPU and GPU as co-processors) for rendering, deformation, and tactile feedback are presented. The prime contributions of the thesis are

- Interactive Surface Wrapping
- GPU-based Implicit Euler Integration
- GPU-friendly Haptic Interaction

1.3 Outline

This thesis is organized as follows. The next chapter provides a brief introduction to the surgical simulation environment and challenges encountered with CPU-based simulators. Chapter 3 discusses the wrapping technique. Chapter 4 describes the Graphics Processing Unit (GPU) and related work in the area of general-purpose GPU and GPU-based simulations. Chapter 5 presents techniques developed for implementing deformation equations on the GPU. It also describes in detail the solution of Implicit Euler equation for simulations. Chapter 6 extends this simulation to one with haptic interaction, including force-feedback. Chapter 7 presents an in-depth analysis of performance bottlenecks in the graphics hardware as well as the software implementation. Finally, Chapter 8 concludes and describes directions for future work.

Chapter 2

Preliminaries and Related Work

This chapter presents the components of a surgical simulation system. Challenges in traditional CPU-based simulation systems will also be discussed.

2.1 Introduction

Surgical Simulation combines virtual deformable body motion with visual and haptic interaction leading to two fundamental challenges: realism and real-time interactions. Modern simulation systems, though based on geometric representation of anatomical structures, need to incorporate the corresponding physical phenomena for realism. This involves combining modeling accuracy with computer efficiency giving rise to a number of issues including, but not limited to, deformation computation and modeling, accuracy of solutions to differential equation systems, real-time collision detection among rigid and deformable bodies, and real-time force feedback [34].

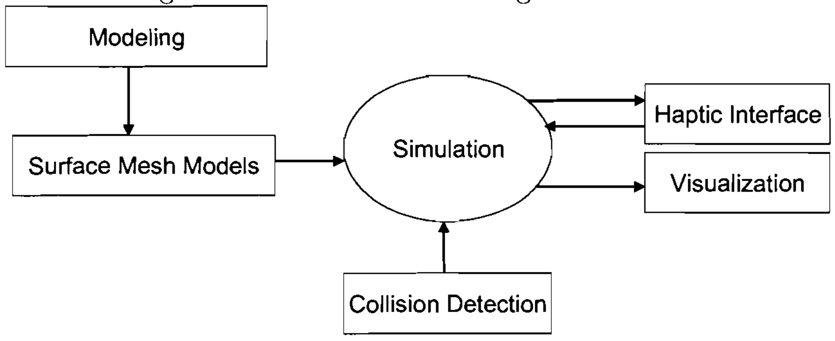
Despite extensive research focussing on these issues, overcoming them is difficult due to the need to run the simulation software on consumer-grade computing platforms like a mid-range desktop PC. Such a cost-efficient yet powerful system is needed as it will be used by surgeons and we assume that they would not have access to high-end systems like supercomputers. This chapter presents components of traditional CPU-based simulation systems along with discussion of work related to the research of this thesis.

2.2 Surgical Simulation

Surgical simulators (Figure 2.1) are components of computer-based training environments designed to allow physicians-in-training to practice current, and experiment with new surgical procedures. The environment simulates interactions between surgical instruments, like graspers or cauteries, and organs or bodies, which are deformable in nature. Apart from training, such simulators also provide scope for improving the productivity of physicians through feedback based on validated techniques.

Therefore, a complete simulation system uses many areas of virtual reality. In this section, we elaborate on the essential software components of a basic framework to simulate organ deformation.

Figure 2.1: Schematic of a Surgical Simulator



2.2.1 Surface Meshes

In computer graphics, three-dimensional models are created using surface or volumetric mesh representations. Volumetric or solid meshes simulate model interiors but can get complex with topological modifications - a significant process in surgical simulation. Surface meshes like polygonal meshes, NURBS patches, or subdivision surfaces are easier to implement and manipulate and are therefore more commonly used. This thesis makes use of triangular surface meshes to represent both deformable and rigid models.

2.2.2 Modeling of Deformable Objects

Deformable modeling is a well-studied area in computer graphics and has been applied to a variety of applications. One of the earliest works on deformable models is presented in [24], coining the term and formulating it in terms of a time-varying partial differential equation.

Deformable modeling can be classified as geometric or physical. Geometric techniques do not take into account the material of the model being deformed. They are easy to implement and have relatively low computational demand but do not include the mass or viscoelastic properties in the model. Splines and patches use a grid of control points to describe surfaces and are widely used due to their flexibility and efficiency. Free-form deformation enables smooth deformations with local control for complex surfaces [70].

For realistic deformations, it is essential to model physical characteristics. Mass-Spring systems are one such widely used physical technique. These systems are a structure of mass nodes and linear stretch springs that connect each node to its immediate neighbors. Springs are associated with a spring constant that generates restoring forces to bring the model back to equilibrium on application of an external force to it. Mass-spring systems have low computational complexity, simple implementation and support for topological manipulations[54][82].

The finite-element method (FEM) is another physical technique that divides the model into a set of elements and solves a global equilibrium function for each of the elements.

In comparison with mass-spring systems, FEM offers more accuracy but comes with higher computational cost and need for careful pre-processing and meshing [48].

This thesis makes use of mass-spring systems for deformable modeling. Simulation of mass-spring models is usually done using explicit integration methods. These methods, though easy to implement, are limited by needing small time steps for numerical stability. This problem was overcome in [23] by the use of implicit integration. However, this method involves solving large linear systems, making them challenging to implement for real-time interactive applications.

2.2.3 Haptic Interface

The word haptic, originates from the Greek word *Haphe* and means pertaining to the sense of touch. Haptics is used widely as an essential component of virtual reality systems as it can give additional information to its user. Currently, there are several commercial haptic devices that can add a sense and feel of touch to computer applications. These devices measure reactive forces that are applied by the user's body (typically the hand) and provide force feedback to the user.

One such device is the PHANTOM® Omni$^{\text{TM}}$ (Figure 2.2) with a six degree-of-freedom input and three degree-of-freedom output. Omni$^{\text{TM}}$ is one of the most cost-effective haptic devices.

The PHANTOM® Desktop$^{\text{TM}}$ (Figure 2.3) is a high-precision device that delivers high fidelity, and lower friction as compared to the Omni$^{\text{TM}}$. This device has a six degree-of-freedom input [35].

The Laparoscopic Impulse Engine$^{\text{TM}}$ by Immersion is another high fidelity interface providing four degrees of-freedom motion (Figure 2.4).
The PHANTOM® OMNI$^{\text{TM}}$ device has been chosen as the haptic interface for

Figure 2.2: PHANTOM® Omni^{TM} Haptic Device

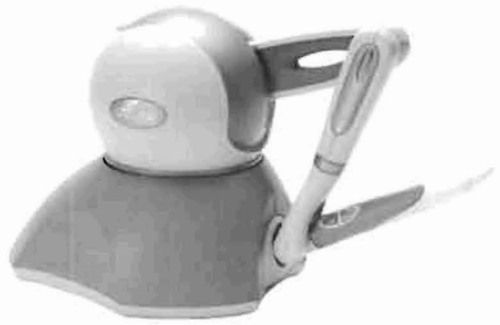


Figure 2.3: PHANTOM® Desktop^{TM} Haptic Device

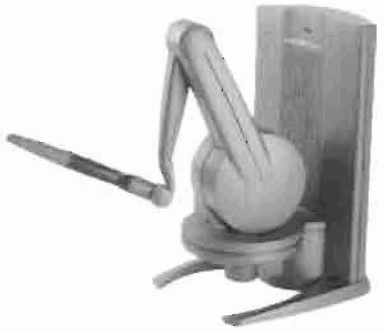
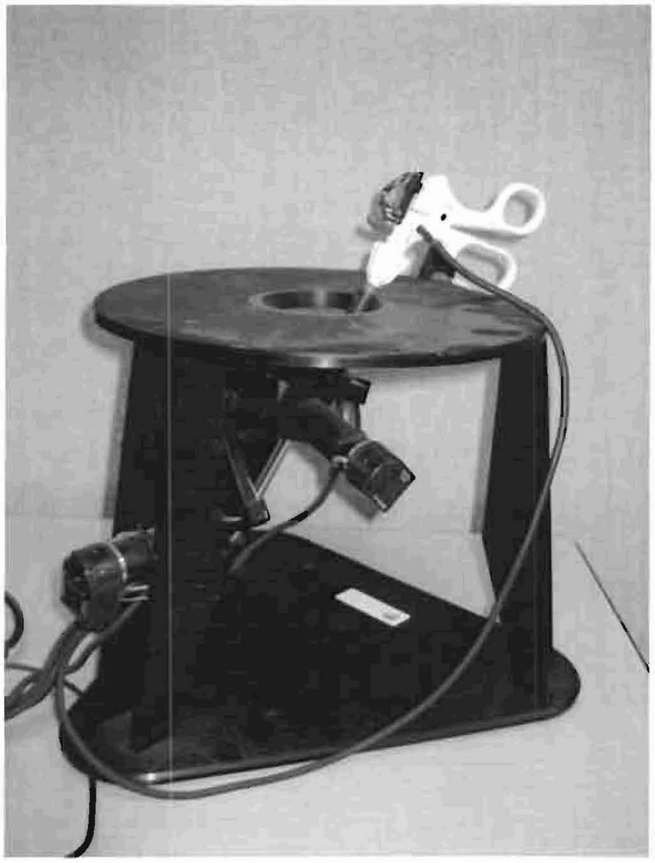


Figure 2.4: Laparoscopic Impulse Engine^{TM} Haptic Device



this thesis due to its cost-effectiveness and availability with the required degrees-of-freedom.

2.2.4 Force feedback

Over the last decade, there has been significant progress in research for modeling haptic interactions in virtual environments. Several algorithms have been developed to compute and display forces for the user to perceive and manipulate virtual objects through touch. A haptic algorithm in virtual environments typically includes collision detection and collision response.

Using the haptic device as a probe, the user manipulates the virtual objects. The change in probe position and orientation causes a collision test to be triggered that detects object-probe interactions. If a collision is detected, interaction forces are calculated and sent to the haptic device. This force-feedback enables the user to acquire a sense of touch with the virtual object and its surface.

Collision response can be implemented using one of the three common techniques: penalty-based techniques, constraint-based techniques, or impulse-based techniques. Penalty-based techniques apply collision forces based on the interpenetration amount [58]. Extensive penetration depths can be avoided by using local or pre-contact penalty methods. However the limitation is that these methods need high stiffness values which can cause instability. They also cause locality issues with multiple object interaction and force issues with small or thin objects. Constraint-based techniques were initiated by [52] [90] that made use of *god objects* (virtual models of haptic interface) constrained on contact with a virtual object. The god object location is computed as a point on the contact-model's surface such that its distance from the haptic interface point is minimized. This method was extended to virtual proxies placed on the object surface directly above the penetrated haptic position. Impulse-based techniques [55] halt the simulation in the event of a collision and resolve contacts based on impulses. However this method cannot handle multiple collisions in a single time step.

2.2.5 Collision Detection

Collision detection in haptics is based on two types of interactions: point-based and ray-based [20]. In point-based interactions, only the tip of the haptic probe is taken into consideration. Collision tests check for the penetration of the tip into the virtual object, calculate the depth and corresponding surface point (called the proxy) position. This method is basic and cannot simulate more complex tool-object interactions involving different parts of the tool being in contact with the object. Ray-based interactions, on the other hand, model the tool as a line segment and check for collisions between the line and the virtual object. This method enables torque interactions and can be used to model complex tools in terms of multiple line segments[20]. This thesis makes use of point-based interactions as our virtual environment makes use of a probe point.

2.3 Virtual Training Environment

The Virtual Training Environment (VTE) [81] is a surgical simulator developed at Simon Fraser University; containing basic and advanced standardized exercises, aimed to develop technique, coordination, and precision for minimally-invasive surgeries. The interactions in the VTE are enabled by one or more haptic devices, which control the position and movement of the virtual surgery tool. The user is provided with both visual and haptic feedback - a three-dimensional view to display real-time interactions of the tool and organ, and force feedback to indicate the precision of the interaction.

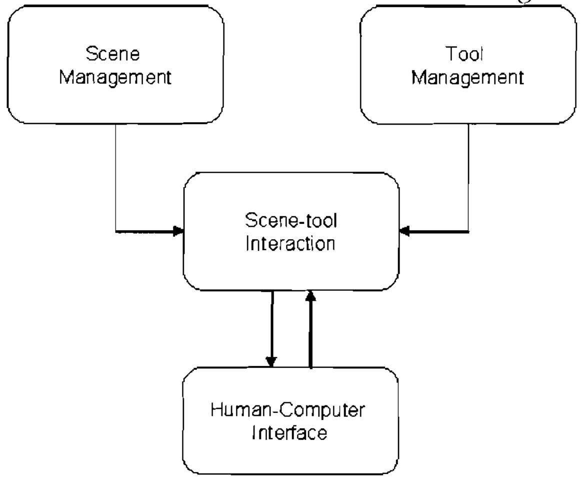
2.3.1 Software Architecture of VTE

The VTE has been developed in the C++ language and currently runs on an Intel 2.8 GHz Intel Xeon processor with Microsoft Windows XP.

Structural View

Structural architecture view identifies the high-level components of the system, and the relationships among them. The VTE framework can be divided into four modules: Human-computer Interface, Scene management, Tool management, and Scene-tool interaction and is depicted in Figure 2.5.

Figure 2.5: Structural view of the Virtual Training Environment



- Human-Computer Interface: The VTE can be integrated with different types of input: keyboard and mouse, the Laparoscopic Impulse Engine, the Virtual Laparoscopic Interface, the PHANTOM® Desktop™ and the Omni™. The Laparoscopic Impulse Engine is the default input and two or more devices can be used in tandem with the software.

- Scene Management: This module involves defining and rendering the virtual scene. A scene contains virtual objects and tools and is defined at initialization in terms of viewing, lighting, and shading parameters. After initialization, the scene, based on user-selection parameters, is rendered. Organ-models are modeled as deformable which enables each organ-model to deform on collision with a tool. A deformable model is represented as a mass-spring system with each vertex linked to its neighbors by mass-less springs of natural length greater than zero.

- Tool Management: A tool in a scene is a rigid body representing the input device. The VTE simulates three surgical instruments: scalpel, grasper, and cautery hook. This module reads the position of the input device, uses this to position the device in the scene and subsequently renders the device.

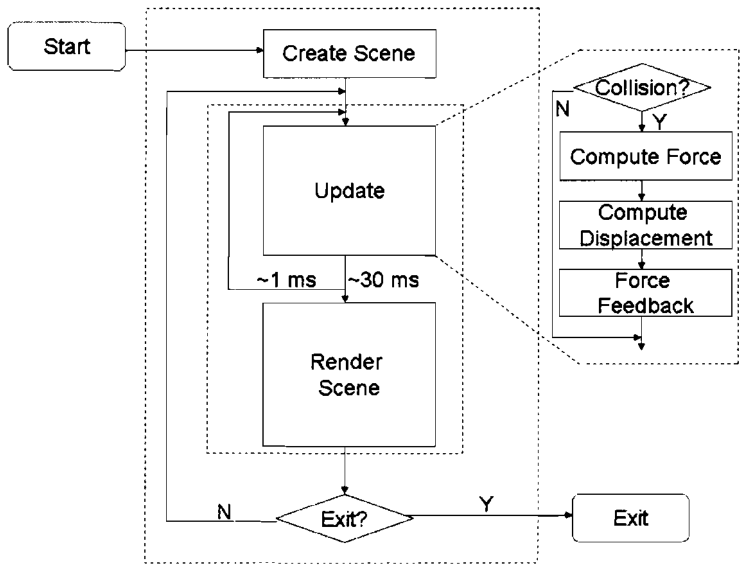
- Tool-Scene Interaction: On contact with a tool, each deformable body goes through a cycle of two processes; collision detection and collision response. Based on the type of tool the contact can result in four different kinds of actions: probing, grasping, cutting, and suturing. Collision detection: A line-triangle intersection method (please refer Appendix B for details of this method) is used to detect the collision. Initially, a global search (through a list containing the model primitives) tests the intersection of the deformable object with the tool. On detection of the first intersection point, the amount of force exerted on the object surface by the tool is determined. The effect of the force is calculated for the neighboring vertices to which the force has propagated. The force stops propagating after a limited number of steps. This limit is controlled in the system by means of an input value and results in local deformations. Collision response: The mass spring model used in the VTE is described by a system of first-order differential equations. Many differential equations cannot be solved analytically, so an approximation to the solution is used. Numerical techniques can be used for such approximations. The VTE uses the Explicit Euler's method for its simplicity and efficiency. Displacement calculation is initiated at the three vertices of the intersected triangle and is calculated for each of the affected vertices. At each time step, new positions are updated on the screen, while reaction forces are sent to the input device.

Functional View

This section describes the sequence of events that occur when the user interacts with the VTE (Figure 2.6). Windows and OpenGL calls are used to define the parameters for scene creation. The user is then provided with a menu option to select the scene to be rendered. The menu options are:

- Basic Tasks: Basic tasks include touching a point, tracing a curve, grasping, and teasing an object.
- Cutting: Cutting a deformable object can be done either along default paths or through user-defined paths.
- Suturing: Suturing involves tying a knot through a cut on the object with demos shown for the same.
- Coach: Audio and visual coaches help provide the necessary skills for laparoscopic surgeries.

Figure 2.6: Functional view of the Virtual Training Environment



On menu selection, the appropriate data structures are created and populated in preparation for the scene setup. At the next stage, the tools selected by the user are initialized and connected to the user interface. At this point, the lighting and texture parameters are set. After the initialization process, a new thread (Task Thread as in Figure 2.6), to render the scene, is created based on the choice selected by the user. The choices provided are different types of virtual models (eg. liver model, stomach model etc.) and different types of virtual tools (probe tool, grasper etc.). The deformable organ in the scene is represented as a mass-spring system [10]. The current position of the haptic device is represented graphically in terms of the virtual tool.

A simulation run consists of evenly-spaced time steps. At each time step, the simulation advances in three stages: Initially, a collision detection test (details in Appendix B) performs contact-check between the tool and the deformable body. Next, on contact, deformation of the organ is computed in terms of a first-order differential equation solved by Explicit Euler integration (details of this method are presented in the next section). Based on the deformed positions, the restoring force is calculated using Hooke's Law, mapped and sent to the haptic device. The mapping essentially uses the computed force vectors to provide feedback to the user through the haptic device. The position of the deformed body is updated. After a few update cycles, the deformed body is rendered on-screen. The rendering thread can be interrupted at the invocation of a scene change by the user. This can be done my means of the menu option.

A menu option in the software provides scene selection that varies based on the specific virtual model displayed as well as the number of surgical tools that can be used.

2.3.2 Challenges

Consider a three-dimensional mass-spring model of N nodes with corresponding position vector $X = (1_x, 2_x, ...N_x, 1_y, 2_y, ...N_y, 1_z, 2_z, ...N_z)^T$, velocity vector $V = \dot{X}$, and spring parameter k. The motion of the mass-spring model, as governed by Newton's second law of motion, is given by

$$F = M * \frac{d^2X}{dt^2} + K * X, \tag{2.1}$$

M,K being the mass and stiffness matrices, and F being the total force vector acting on the mass-spring model. F is a result of internal (spring parameters like stiffness and viscosity) and external (gravity) forces. The above equation implies that an external force acting on a node causes it to deform and move towards an equilibrium constrained by the stiffness of its springs. Numerical evaluation of the equation 2.1 is done in the VTE by using the explicit Euler integration method,

$$\begin{aligned} V^{t+1} &= V^t + F^t * \frac{dt}{m}, \\ X^{t+1} &= X^t + V^{t+1} * dt. \end{aligned} \tag{2.2}$$

As can be seen from the Equation 2.2, the position and velocity of affected nodes in a time-step are determined by forces in the corresponding previous step. This technique can lead to wild changes in positions and velocities. Also, explicit technique needs to satisfy the Courant Condition [72]. Else, large time steps can cause instability due to stiffness of equations with high spring constants. Depending on the complexity of the computations and the speed of the machine being used, these factors could cause the simulation to run 10 to 1000 times slower than real-time.

2.4 Profiling

A profiler is a code analysis tool that allows examination of the run-time behavior of a software application. A profiler helps analyze the performance of the application such as identifying the functions that consume most of the time, isolating potential bottlenecks and analyzing memory usage. This helps in determining the efficiency of the algorithms used. I have used an independent profiler, GlowCode Loader [1], to analyze the VTE source code. A one-object model was simulated, deformed and subjected to profiling. Function profiling was done to identify functions that consume the most amount of time in the simulation. This helped detect inefficient code.

Potential bottlenecks were isolated that include the line-triangle collision detection algorithm (within Update in Figure 2.6) and the use of a single thread for CPU-intensive tasks (Task Thread as in Figure 2.6). The collision detection is a major bottleneck because at each time step, the algorithm performs a global collision test involving all the nodes of the deformable model. Another major constraint is the sequential computation of the simulation algorithm on the entire collection of model nodes. The algorithm runs the same set of instructions sequentially on each model node. This causes an increase in update time of the simulation, thus decreasing the rendering frame rate. This computation time can be reduced by distributing the data, in this case model-nodes, to multiple processors and running the same set of instructions in parallel on them.

2.5 Discussion

On a parallel computer, the computation can be distributed among different processors. Surgical simulation can fit a parallel model as it involves implementing a single instruction on a series of mass nodes of the mass-spring model. As described in the previous section, the update cycle is implemented on each mass-node of the system. Using multiple processors, the update cycle can run in parallel on several groups of mass-nodes, thus reducing the simulation time. Also as the processing power of the computer increases, model complexity can be increased, thus making a more realistic scene.

Therefore, the next logical step is to develop a multiprocessor simulation system to increase the computational power. A master-slave parallel model can be used with the CPU being responsible for organization and dissemination of tasks, and an additional processor used for deformation calculation and rendering. The next chapter details the additional processor chosen to do so.

Chapter 3

Surface Mesh Mapping

Geometry is the science of correct reasoning on incorrect figures.

George Polya

This chapter deals with the issue of wrapping a two-dimensional planar surface mesh onto a three-dimensional closed surface mesh. *Wrapping* is used here in the context of deforming a two-dimensional object mesh around a three-dimensional model such that they are in complete contact with each other. Related parametrization and wrapping techniques are explored and a semi-automated wrapping method is presented and discussed.

3.1 Introduction

Most recent simulations use 3D models for realistic visualization. These models can be represented either as a surface or a volume. Volumetric models can simulate objects with interior structures but result in complex topological modifications if they undergo progressive cutting - an essential element of surgical training software. Since we used surface meshes for modeling, we chose the former for simplicity. Surface models can be approximated by triangular or quadrilateral primitives, and are generated using commercial Computer-aided Design (CAD) packages. This thesis makes use of triangle primitives as they have a single configuration based on their edge lengths, used since VTE simulations involve deformation with restorative spring forces.

Prior GPU-based integration techniques [85] were implemented for 2D planar meshes. The details of these techniques will be discussed in chapter 5. These techniques needed to be extended to 3D closed meshes while maintaining the structure of the corresponding Hessian matrices used in the integration in [85]. Parametrization and wrapping techniques were explored to accomplish this.

3.2 Surface Parameterization

Surface parameterization is a one-to-one mapping process of a 2D input domain to a 3D surface. These mappings are usually piecewise linear and the surfaces are represented by triangular or quadrilateral primitives. Parameterization methods were initially developed for texture mapping, which is a technique for adding visual detail to a three-dimensional surface [79]. Other applications include scattered data fitting (reconstructing smooth surfaces from scattered data points)[31], modification of CAD models and surface approximation.

Various kinds of parameterization of mappings are defined in [78]; Isometric mapping, where the length of any arc on the input surface is preserved in the output surface; conformal mapping where the intersection angles of any two arcs on the input surface is preserved in the output surface and equi-real mapping where every part of the input surface is mapped to the output surface preserving the area. The quality of parametrization is usually characterized by two factors: distortion and validity. Distortion, intuitively, is a measure of the amount by which the edges are stretched after the parametrization is applied.

Orthographic projection modifies both angles and areas while stereographic projection modifies areas but preserves angles. Mercator projection does not preserve areas while Lambert projection modifies angles. A parameterization scheme is considered valid if there are no edge overlaps. Since parameterizations are known to introduce distortions, most research aims at reducing them.

Texture atlas [67] is one of the earliest approaches to parameterization which involves partitioning surface into charts (part of the surface associated to a region of the input image), parameterizing each of them to a plane, and packing them into an atlas. A stored file describes the coordinate mapping from the textures to the atlas. However the two main drawbacks in this approach are the introduction of visible surface seams and complexity in accessing usage information. Barycentric maps were introduced in [88] in which each vertex is mapped to the barycentric center of its neighbors. Given a topological disk, the boundary is mapped to a convex polygon on the plane. Floater [78] extended this method to arbitrary convex combinations using weights to control the distortions. Another common approach is semi-regular remeshing involving partitioning a mesh into charts, and subdividing it before individually parameterizing them. However, this approach is subject to high distortion and needs extra storage requirements.

Geometry images was introduced in [89] in which an input arbitrary surface geometry is remeshed into a regular square image grid. The method involved defining a series of cut-paths to cut the surface and then map the resultant surface to a square grid. Surface information (normals, color, etc.) are stored in 2D arrays. This method achieves a seamless parameterization with implicit correspondence between geometry and grid, thus ruling out storage needs. These images are ideally suited for hardware rendering as they can be sent to the graphics pipeline in a compressed form just like square textures. There are several solutions readily available for the parameterization process, some of which have been explored by the author. An exhaustive survey of these methods can be found in [78]. However these techniques are constrained by *a priori* chartification and heuristic cut paths, which poses optimization problems.

Recently, [71] proposed to take advantage of the above-defined geometry images to flatten deformable models. This was done by means of a multi-layer parameterized representation of unstructured three-dimensional meshes which were resampled into two-dimensions. With a parameterized representation, the high resolution three-dimensional models were interactively deformed with a fast free-form deformation method.

Our simulation system requires genus-zero closed surface meshes, so we needed parametrization techniques that could be applied to such surfaces. The sphere is the most natural parametrization domain for such meshes as it does not involve cutting of the surface. While planar parametrization has been extensively studied, there has been comparatively less work in the area of spherical parametrization. One of the first works in the area was done by [43] in terms of balloon inflation. However, a one-to-one mapping could not be guaranteed. Another technique was the partitioning of a surface into six charts, mapping them to cube faces, and then to a sphere [21]. Spherical parametrization techniques pose two challenges: one, fold-overs are tough to avoid and two, all parts of highly deformed surfaces (like most organs) are difficult to parameterize. We were also restricted by the need to parameterize the mesh to a square in order to conform with the existing Hessian matrix structure which is a sparse symmetric matrix obtained from the implicit integration technique (details of this technique are described in Chapter 5).

3.3 Surface Wrapping

Surface wrapping can be viewed as a reverse-engineered approach to surface parametrization. It was originally developed in automotive modeling for design optimization of vehicles [77]. It has since been used in diverse areas involving complex geometries like artery flow modeling, turbine design, external aerodynamics, textile draping etc. Several processes in the manufacturing industry also use this technique for wrapping a sheet of material over a 3D surface and as a result several CAD/CAM systems like Autodesk 3DS Max include a wrapping feature for surfaces. This feature replicates the movement and deformation of a piece of fabric or clothing on interaction with a collision object and an external force, such as gravity or wind. The main benefit of such a feature is that it provides rapid output of results with reasonably strong control over the wrapping parameters. Complete control of U and V parameters (values that define points on a parameterized surface) as well as surface normals enable achieving the desired 3D shape.

The author also investigated textile wrapping methods as the input mesh can be assumed as a virtual cloth. An in-depth review of the research done in textile wrapping has been provided in [39][40]. Textiles were modeled as particle sets in [27] and performed several simulations using polynomial approximation and linear fitting with results compared with physical textiles.

3.4 Semi-automated Implementation

For ease of usage, we chose to make use of the *Cloth Modifier* tool available in Autodesk 3DS Max [3]. *Cloth* provides a general-purpose physically-based simulation that progressively models the wrapping of a "cloth-like" input mesh over an arbitrary closed surface through a sequence of successive approximations. Gravity and collisions are taken into account to simulate the motion. The simulation is terminated once the desired shape is achieved. Relaxation techniques are used to ensure minimum wrinkles and fold-overs. Input models used varied from 16 vertices to 16384 vertices and the wrapping times varied from 20s to 120s. Relaxation steps are incorporated during the wrapping process to keep the input mesh from excessive stretching or folding. To demonstrate the method, an example surface wrapping simulation is provided in Chapter A.

3.5 Issues

The issues encountered with this method are summarized as follows: The planar mesh penetrates the three-dimensional surface model if it has a low polygon count. We experienced this problem with a 16-vertex model.

The *Cling* parameter of the mesh object, defined as the extent to which the mesh adheres to the collision object, needs to be set very high for it to achieve the desired shape. In our case, we set it to 500.

This process can also cause mapped vertices to get too close together during the wrapping simulation thereby causing distortion or overlapping. An advanced toolset, *Relax modifier*, provides methods and numeric parameters to modify the vertex-spacing and resolve this problem.

3.6 Discussion

The wrapping system presented can compute the complete trajectory of the mesh motion. This is useful as it aids in obtaining an accurate wrap through back-stepping and trail-and-error re-simulations. Various "cloth-like" properties like thickness and cling values can be adjusted to achieve realistic desired shapes. The effects of external forces, like gravity, can be modeled accurately. This aided our simulation as we used gravity force to let the mesh object *drop* and *cling* to the collision object. The method can be extended to meshes of large sizes as well. The clear advantage of this method is its flexibility making it easy to use with multiple-shaped collision objects and multiple-sized mesh models. It also as well as its speed, stability, and accuracy. However this system is part of a professional package and is therefore not a free plugin.

Chapter 4

Graphics Processing Unit

A chain is no stronger than its weakest link.

William James

This chapter introduces the Graphics Processing Unit (GPU) and describes its architecture and functionality. A summary of the applications of GPGPU (General-purpose GPU) in linear algebra and surgical simulation are presented.

4.1 Introduction

A GPU is a graphics rendering device available on commercial off-the-shelf graphics cards. It implements graphics primitive operations and consists of a series of computational units with data flow between them. The GPU pipeline [62] is based on a stream programming model, i.e. a data-parallel model where given a set of data, (*a stream*), instructional operations can be applied to each data element in the stream. It thus acts as a data-parallel processor. Current GPUs are programmable through shader programs - a set of software instructions that run simultaneously on every data element on the graphics computation units.

Over the last decade, graphics processing technology has been accelerating at a very fast pace roughly following Moore's law cubed [33]. The GPU has been traditionally used to perform graphics operations such as shading and rendering. With modern GPUs being programmable and supporting floating-point operations, these chips are increasingly being used to solve general problems. In this thesis, data-parallel processing capabilities in the GPU are exploited to overcome the limitations of current CPU-based simulators for deformations of physics-based surface meshes. We reformulate the integration equation for deformation and compute the simulation equations on the GPU (details of this technique are described in Chapter 5).

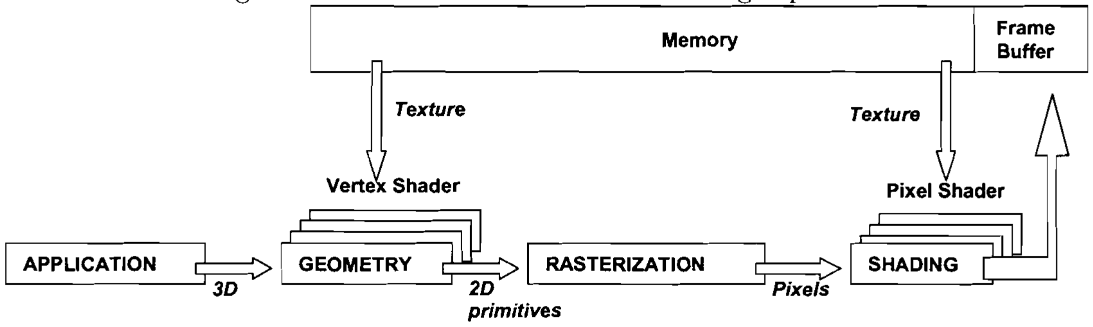
The fundamental reason for the performance increase in GPU as compared to the CPU is due to the differences in architecture. CPUs are designed to deliver high performance for sequential operations with millions of transistors focussed on speeding up the execution of a single thread. GPUs, on the other hand, use many more million transistors on supporting multiple threads concurrently on-chip, facilitating thread communication and sustained high memory bandwidth.

4.2 The Rendering Pipeline

The rendering pipeline constitutes the stages of graphics hardware to render data onto the screen. It can be coarsely divided into four conceptual stages: Application, Geometry, Rasterizer and Shading [14]. Each of these stages is a pipeline itself and consists of several functional sub-stages. A functional sub-stage performs a specific task. The update speed of the images i.e the rendering speed is determined by the slowest stage in the pipeline. This speed is generally expressed in terms of frames per second (fps) or in terms of frequency (Hz). Figure 4.1 depicts the rendering pipeline.

The application stage is software-based and may contain geometrical algorithms - collision detection, acceleration, force feedback etc. Geometric primitives like points, lines, triangles etc. are the output of this stage.

Figure 4.1: Detailed View of the Rendering Pipeline



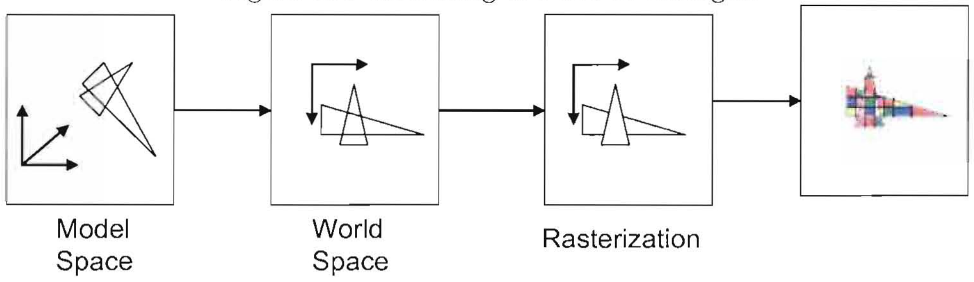
The geometry stage can be either software or hardware-based and deals with geometric transformations, lighting and projections. This stage performs per-primitive operations, the primitives being either vertices or polygons. Each mesh model exists in its own model space which is transformed with its respective model transform to a common world space. A view transform places the camera at the origin and converts the models to the camera space based on the position and orientation of a virtual camera. Based on the lighting sources, a lighting equation is used to compute a color attribute for the model vertices. A projection method (orthographic or perspective) transforms the model data to 2D image space, mapping the 3D scene onto a plane as seen from the virtual camera. The resultant data undergo clipping and are converted into screen or window coordinates. The resultant data is now in the form of 2D primitives.

The rasterizer stage renders the image based on the data obtained from the previous stages, i.e. 2D primitives. This stage performs per-pixel operations. It calculates which pixels are covered by each primitive, and it uses z-culling through the depth buffer to eliminate pixels that are occluded by objects with a nearer depth value. Once the visibility is resolved and primitives are converted into resultant pixels, also called fragments, they are input to the shading stage.

The shading stage is hardware-based and involves applying color and textures to the corresponding primitives. The image can either be rendered on-screen using the frame buffer or fed back using one, two or three-dimensional rectangular arrays called textures into an off-screen render target.

Figure 4.2 details the working of this process on a set of triangles.

Figure 4.2: Rendering of a Set of Triangles



Modern graphics cards enable per-vertex and per-pixel operations to be programmable, terming the functionality as vertex shader and pixel/fragment shader respectively. Shader programs can be written using APIs like OpenGL [5] and DirectX [4] and programming languages like GLSL and Cg [73] that have been customized for GPU programming. These programs are run by the GPU which switches between executing fixed-function and shader program code. A vertex shader can perform simultaneous per-vertex operations on vertex data to move them, modify their attributes or create animations. Vertices are processed independent of each other and therefore several vertex shaders can operate on several vertices in parallel thus reducing the total computation time. Output vertices are built into triangles and undergo a selection process that clips and culls pixels that are not visible on the screen. The pixel shader can perform per-pixel operations on the remainder visible pixels to color or texture map them. This pixel processing is an independent operation and therefore pixel shaders can operate on several pixels in parallel.

The rest of this chapter builds upon the rendering pipeline and details some of the applications that have taken advantage of the parallelism in the pipeline.

4.3 GPU Architecture

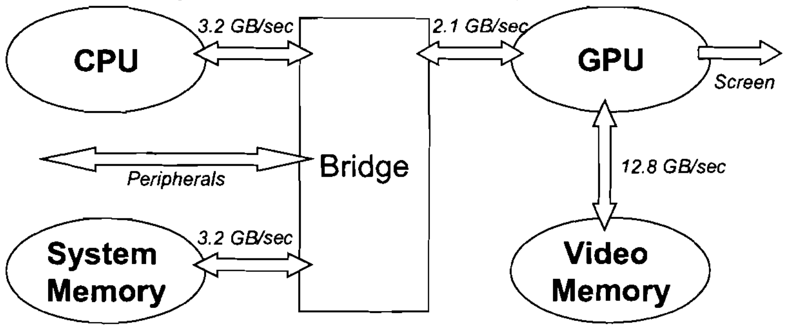
This thesis makes use of the NVIDIA GE Force 7600 GS graphics card for the simulations. This card is a seventh-generation NVIDIA card based on the GeForce 7 Series of graphics chips. This series supports several advanced features including SLI (Scalable Link Interface) that allows linking of two or more graphics cards together to produce a single output, HDR (High Dynamic Range) rendering that creates realistic scenes using lighting done in a larger dynamic range, and Intellisample that improves anti-aliasing quality. Table 4.1 provides specifications of 7600GS [2]. The CPU connects to the GPU by means of a graphics connector - AGP. Vertex and pixel shaders support near-infinite length shader programs as there are minimal hardware-imposed limitations on these programs. Pixel shaders also support multiple render targets (MRTs) providing the ability to render to upto four separate output buffers with a single draw call. For example, photorealistic lighting can be achieved through MRT that enables deferred shading, a technique where the lighting of a scene can be done after rendering all of the geometry, eliminating multiple passes through the scene. This thesis makes use of MRT to achieve high performance, details of which are in Chapter 5. The texture engine contains upto 16 textures per rendering pass with support for 16-bit and 32-bit floating point formats and non-power-of-two texture sizes. The texture and pixel shader operate on squares of four pixels (called quads) at a time. We use MRT and textures to compute positions and velocities simultaneously and solve deformation equations in our simulation system (details in Chapter 5). Figure 4.3 gives an overview of the system interface.

4.4 Languages and Tools

This section gives a brief overview of the languages and tools that can be used to successfully construct GPU programs.

Shading languages are special high-level programming languages that enable writing and compiling shader programs into either vertex or pixel shaders. To map onto the

Figure 4.3: Overview of the system interface



Features	Performance
Graphics Bus Technology	AGP 8x
Memory	256 MB
Memory Interface	128-bit
Memory Bandwidth(GB/sec)	12.8 GB/sec
Core Clock Speed	400 MHz
Vertex Shader Units	5
Pixel Shader Units	12

Table 4.1: Specifications of NVIDIA 7600 GS

graphics pipeline, these languages have special data types, like color, and generate image outputs. Currently there are two APIs used in graphics development - DirectX by Microsoft and OpenGL by ARB (Architecture Review Board). Each of them provides a C-based shading language, HLSL (High Level Shading Language) and GLSL (OpenGL Shading Language), respectively while Cg (C for Graphics) can be used with either API. GLSL has cross-platform compatibility and enables vendor-specific optimized code constructs. Cg (C for Graphics) has been developed by NVIDIA and provides API independence and free tools for asset management. HLSL is tightly integrated with DirectX and limited to the Windows operating system. Cg is very similar to Microsoft's HLSL. All the three languages offer comparable features in terms of syntax and semantics. However, they differ in terms of hardware and platform support. For this thesis, we choose to make use of GLSL as it is a core component of the OpenGL 2.0 specification [44]. It was also chosen due to its language compatibility with the CPU-based surgical simulation software.

There has been very little support for debugging GPU applications. The Microsoft Shader debugger [7] is integrated into Visual Studio and provides runtime variable watches and breakpoints. However, it runs the shader in software emulation rather than the hardware. GLIntercept [25] and gDEBugger [9] help debug OpenGL programs. They too provide breakpoints and variable watch but only allow runtime editing of shaders. This thesis made use of gDEBugger intermittently for software support.

4.5 Simulations and General-purpose Applications

GPUs were traditionally used to perform computer graphics computations. But with programmable shaders, these chips are increasingly being used to solve general problems.

The general-purpose GPU (GPGPU) is a relatively new area in the field of GPU computing that explores the use of graphics hardware for general-purpose computing. For peak performance in a GPU, the arithmetic intensity, i.e. the ratio of arithmetic operations to memory access, should be high else memory access latency will limit computation speeds. Applications with high data-parallelism and involving minimum CPU-GPU interactions are desirable. Such applications work well on the GPU as a single kernel program can evaluate multiple data units in parallel. Also, latency due to downloading data to and reading data from the CPU can be minimized.

In [49], Thompson et al developed a C++ framework for writing general-purpose programs with vector operations. They applied this framework to a variety of problems; arithmetic, exponential, factorial and multiplicative operations and compared the speed of graphics card implementations to CPU implementations with increasing size of input vector data. The results showed a range of 5 to 16 times faster GPU implementation for the largest vector size (10^7 elements) demonstrating the efficiency of using GPU for vector operations. The paper also showed the importance of using the GPU for executing a maximal amount of work on each chunk of submitted data.

This has been reiterated in [51] which investigates the efficiency of dense matrix-matrix multiplications in GPU. The investigation revealed that such computations involve constant cache access (due to reuse of data) and are therefore inefficient on a GPU due to current low GPU-cache bandwidths.
The approach in [46] is of relevance to this thesis. The authors validate the effectiveness of matrix-vector operations by developing a framework for implementing techniques to solve difference equations on GPUs. In their paper, vectors are represented as two-dimensional textures and matrices are represented as a set of diagonal vectors. Arithmetic operations for matrices are performed by specifying multiple adjacent diagonals as multi-textures and combining these textures in a shader program to evaluate the result. Using this technique, Conjugate-Gradient and Gauss-Seidel methods have been implemented on the GPU. Similar approaches have been used in [41] and [65]. A contribution of this thesis is to make use of a similar matrix representation for the sparse matrix used in implementing implicit Euler integration on

the GPU (details in Chapter 5).

GPUs have also been used for solving computationally intensive collision detection for deformable bodies. Using Z-buffer depth comparison, object pairs can be tested for overlap against each other. If overlap occurs, a second pass tests the sub-objects for overlap. The overlapped sub-objects constitute potential colliding sets (PCS) and are then tested for collision on the CPU [66]. A similar approach was also taken in [28] in which the distance information for computing Voronoi diagrams is obtained by rasterizing distance functions for the geometric primitives. In [22], the authors make use of graphics hardware for collision detection in surgical simulation. Each scene is rendered relative to a viewing volume encompassing the tool volume. A collision check is done on a reduced search space containing parts of the scene that can be viewed from the viewing volume. This technique is very simple to implement and runs much faster that traditional bounding box techniques as no pre-computation is required. GPUs have also been used to detect self-collisions in deformable bodies [42]. In this case, a stencil test is used to reduce the search space for collision checking. The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer and a reference value.

A large particle system simulation was implemented by [69] on the GPU. Their method included making use of pixel shader programs for sorting particles for collision-pairing and detecting collisions. Another GPU particle system simulator was implemented by [12] that performed depth comparisons on the GPU to compute accurate collisions. Another particle system was implemented entirely on the GPU to visualize steady flow fields [45]. A dedicated GPU-based surgical simulator was implemented by [47]. In their paper, vector processing capabilities of the GPU were exploited to calculate the positions of a mass spring system and visualize it. The particles of the mass spring system were connected either as a regular 3D grid or as an irregular grid with a separate texture array maintained to store the connectivity for all the particles. Verlet integration was used for the integration scheme. This method calculates the position at the next time step from the positions at the previous and current time steps without making use the velocity.In this thesis we chose an alternative integration scheme, details of which are in Chapter 5

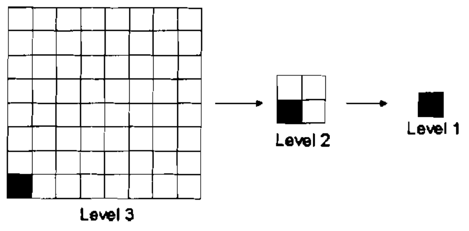
Transferring data from CPU to the GPU is a bottleneck that makes it difficult to fully exploit the potential computational throughput of the graphics processing unit. This transfer also introduces overhead that makes applying the GPU to small data problems inefficient. Communication latency between the CPU and GPU can be reduced by using a hierarchical read-back of colliding pairs [42]. Such a structure is implemented by a set of encoded 2D off-screen buffers containing collision results (as boolean values) and is shown in Figure **??**. The textures are hierarchical i.e. layer of off-screen buffers encode the original off-screen color buffer. The pixels at each buffer encode results of the corresponding $4x4$ pixels in its lower-level buffer. Pixels are read from the GPU only when a collision occurs. Another issue in a GPU implementation is minimizing the communication between GPU and CPU. Packing 4 different computed values into the RGBA channels of each pixel can be done to save time on reading pixels from CPU to GPU [87]. This method would however involve writing onto the same pixel 4 times in the GPU.

Overall, these methods had a speedup in the range of ten to twenty times as compared to a similar CPU-based system.

4.6 Performance Analysis of GPU-based Applications

The performance of a GPU application can be improved by locating bottlenecks in the pipeline. The bottleneck can be eliminated by either trying to reduce the workload of the bottle-necked stage or by attempting to increase the workload of the other stages in the pipeline.

Figure 4.4: Heirarchical encoding as in [42]. Each pixel in the higher level buffer encodes the contents of 16 pixels in the lower level buffer.



There are a few tools available for performance analysis of GPU applications. A benchmarking approach can help improve performance by running standard tests to assess the application. GPUBench [37] is one such collection of software tools to evaluate the performance of graphics hardware.

NVPerfHUD [6] is a real-time performance analysis tool by NVIDIA that has been proven to improve application performance by an average of 35% after being used to tune these applications. It indicates draw-call duration, unit bottlenecks, double-speed z/stencil and pixel count, thus helping identify bottlenecks.However it can only be used for Direct3D applications.

Another performance measure is in terms of memory bandwidth, i.e. the rate at which data is transferred to and from the graphics or system memory and cache efficiency, i.e. the optimal number of cache hits for an application.

For this thesis, we used GPUBench to analyze the application with respect to the graphics hardware. We also analyzed the application in terms of the update rates, computational complexity and bandwidth usage, the results of which are presented in Chapter 7.

Chapter 5

Deformation of Surface Meshes

Truth is much too complicated to allow anything but approximations.

John von Neumann

In this chapter, existing integration methods for computing mesh deformations are explored. Implicit Euler's integration is chosen over other methods and implemented on the GPU with details of the simulation presented and analyzed.

5.1 Introduction

Deformable modeling and simulation have been a constant challenge in the field of computer graphics. There are two kinds of modeling approaches; geometric and physical. One of the most common physical approaches is the mass-spring system that models deformations with low computational complexity, simple implementation and provides support for topological manipulations [54] [82]. Physically-based deformations are formulated as time-varying partial differential equations. These equations are discretized and numerically solved as ordinary differential equations (ODE).

For interactive real-time applications like surgical simulation, explicit integration methods are generally used to solve ODEs as they are fast and easy to implement [38]. However, these methods are known to cause instability and can result in simulations running slower than real-time. We therefore choose to make use of an implicit integration technique to achieve stable simulation of deformable anatomical structures. This integration technique is compute-extensive as it involves solving large linear equations (see Section 5.5). One way of meeting the computational requirements is to evaluate if the current algorithm can be parallelized and then develop a system to take advantage of the inherent parallelism in the Graphics Processing Unit - a chip in many mid-range graphics cards to solve the equation.

This chapter explains the details of such an implementation.

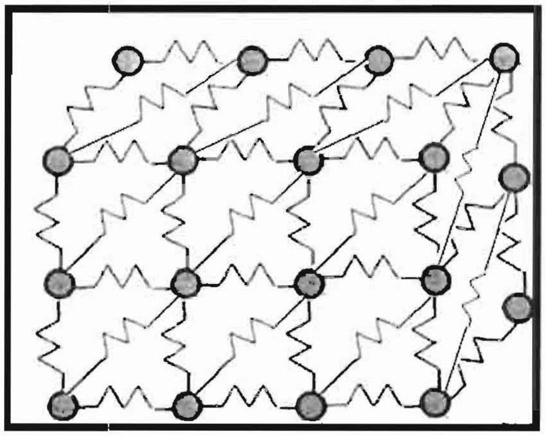
5.2 Deformation of Surface Meshes

This thesis makes use of mass-spring models to represent the anatomy. As described earlier, a mass-spring model is a special particle system with fixed topology connecting neighboring particles that constrain the motion of the model by means of internal forces.

The deformable model in our simulation is represented as a three-dimensional mass-spring system.

Consider a model of an anatomical structure consisting of N nodes, $i \in R^3$. These nodes contain mass m but do not constitute a volume i.e the anatomy is represented as a three-dimensional surface structure in three-dimensional space. A spring constant associated with springs cause a restoring force to bring the model to equilibrium when an external force is applied. The model can be stretched (in-plane deformation) and bent (out-of-plane deformation). The springs' internal forces are affected by the distance between their connecting nodes (Figure 5.3).

Figure 5.1: Mass-spring system



This internal force vector F_i is computed as follows:

$$F_i = k \sum \frac{l_{ij}}{|l_{ij}|}(l_{ij} - r_{ij}) + f_i \tag{5.1}$$

where

$$f = -kx \tag{5.2}$$

k = mesh spring constant,

l_{ij} = edge length between neighboring nodes i and j,

r_{ij} = rest length of edge between neighboring nodes i and j,

f_i = external force applied on the ith node.

Given the initial system configuration (in terms of Equation 5.1, l_{ij} equals r_{ij} and f_i being zero), on application of external force, the simulator computes the motion of the system over time. A simple example is the motion of mass, m, on a single spring as in figure ??. The motion, as governed by Newton's second law, is given by

$$f = m\ddot{x} \tag{5.3}$$

f being the force exerted on the mass and $\ddot{x}$ being the acceleration of the spring. Assuming that the spring in the system follows a simple linear relation, Hooke's Law states that the amount of deformation in the spring is linearly related to the force causing the deformation, or

$$f = -kx \tag{5.4}$$

k being the spring constant and x being the spring displacement. From 5.3 and 5.4 we get

$$m\ddot{x} = -kx \tag{5.5}$$

or

$$m\ddot{x} + kx = 0 \tag{5.6}$$

Extending the above equation to a n-node system we get,
$M\ddot{x} + Kx = f(x, \dot{x})$ or

$$M\frac{d^2x}{dt^2} + Kx = f(x, \dot{x}), \tag{5.7}$$

$\ddot{x}$ being the acceleration vector, f the external force vector, M the mass matrix and K the stiffness matrix of the system. In our simulator, the system is 3D so x and f are vectors of size $3n$ (n being the total number of nodes) while M and K are $3n \times 3n$ matrices defined as $M = diag(m_1, m_1, m_1, m_2, m_2, m_2, ..., m_n, m_n, m_n)$ and $K = diag(k_1, k_1, k_1, k_2, k_2, k_2, ..., k_n, k_n, k_n)$ respectively. Note, $diag(a_1, ..., a_n)$ defines a diagonal matrix whose diagonal entries starting in the upper left corner are $a_1, ..., a_n$. Equation 5.7 implies that an external force acting on the system causes it to deform and move towards an equilibrium constrained by the stiffness of its springs.

Equation 5.7 is a differential equation and can be expressed as two first order differential equations:

$$\frac{dv}{dt} = \frac{1}{M}(f(x, \dot{x}) - kx), \tag{5.8}$$

$$\frac{dx}{dt} = v \tag{5.9}$$

Formally, we need to solve an initial value problem by integrating a set of ordinary differential equations in time [17]. There are several existing methods. One way is to use a first-order Taylor series approximation of the derivatives to propagate the solution forward in time.

This approximation around initial values of velocity and displacement are given by:

$$v(t = \Delta t) = v(t = 0) + \Delta t \frac{dv}{dt}\Big|_{t=0} \tag{5.10}$$

$$x(t = \Delta t) = x(t = 0) + \Delta t \frac{dx}{dt}\Big|_{t=0} \tag{5.11}$$

Iteratively using the above equation, we can express equations for velocity and displacement at any given time-step, $N + 1$, as:

$$v^{N+1} = v^N + \Delta t \frac{dv^N}{dt} \tag{5.12}$$

$$x^{N+1} = x^N + \Delta t \frac{dx^N}{dt} \tag{5.13}$$

Substituting for the derivatives from Equations 5.8 and Equation 5.9, the following explicit Euler expressions for velocity and displacement can be obtained.

$$v^{N+1} = v^N + \frac{\Delta t}{M}(F^N - kx^N) \tag{5.14}$$

$$x^{N+1} = x^N + \Delta t v^N \tag{5.15}$$

The above equations are explicit as they calculate the state of a system at a later time (in this case, $N + 1$) from the state of the system at the current time (in this case, N).

5.3 Existing Techniques and Issues

As explained in Chapter 2, computing deformations using explicit methods, though easy and fast, causes the force at a time step N to contribute to node velocities, and therefore positions, in the subsequent time step, $N+1$. As explained in [23], this can lead to wild changes in positions and velocities. Also, explicit methods need to satisfy the Courant Condition [72]. Large time steps can cause instability due to stiffness of equations with high spring constants. Therefore, unless time-steps are very small, small enough to meet Courant's condition, the simulation will be unstable. However small time-step means longer computation times, which can cause the simulation to run much slower than real-time.

5.4 Implicit Euler Integration

The deformable model in this thesis is a three-dimensional mass-spring surface mesh with linear *stretch* springs. We us the approach in [23] to reformulate the integration equations of the surface mesh and have forces at each time step contribute to velocities in the same time step. This is called the implicit method as it obtains a solution to the numerical equation by involving both the current state of the system and the later one. We simulate the surgical task of probing ie. pulling or pushing part of the model using a surgical probe. This may cause the distance between nodes to change but their neighborhood connectivity remains fixed.

Numerical evaluation of equation 5.3 can be done by using the implicit Euler integration method,

$$\begin{aligned} V^{N+1} &= V^N + F^{N+1} * \frac{\Delta t}{M}, \\ X^{N+1} &= X^N + V^{N+1} * \Delta t. \end{aligned} \tag{5.16}$$

$$\begin{aligned} V^{t+1} &= V^t + F^{t+1} * \frac{dt}{m}, \\ X^{t+1} &= X^t + V^{t+1} * dt. \end{aligned} \tag{5.17}$$

$$F^{t+1} = F^t + \frac{\partial F}{\partial X}\Delta^{t+1}X. \tag{5.18}$$

$$\Delta^{t+1}V = (I - \frac{dt^2}{m}H)^{-1}(F^t + dtHV^t)\frac{dt}{m}. \tag{5.19}$$

$$\Delta^{t+1}X = V^{t+1} * dt. \tag{5.20}$$

The superscript indices indicate the corresponding time-step. This equation stabilizes the simulation as the motion of nodes will always be consistent with the forces corresponding to its current time-step. However, this technique involves determining F^{n+1} without knowing the position values of the nodes at the corresponding time step.

This can be done using Taylor's theorem for linear approximation,

$$F^{n+1} = F^n + \frac{\partial F}{\partial x}\Delta^{n+1}x. \tag{5.21}$$

Using Equation 5.3 and Hooke's law of elasticity, the stiffness matrix K evaluates to the negated Hessian matrix, H, of the system. The Hessian matrix is a large, sparse and narrow-banded matrix, the size, number of diagonals, and non-zero element values of which depend on the structure of the mass-spring model. Substituting Equation 5.21 into Equation 5.16 and using the backward operator, $\Delta^{N+1}X = (V^N + \Delta^{N+1}V)dt$, for the position vector, we get:

$$\Delta^{N+1}V = (I - dt^2H/m)^{-1}(F^N + dtHV^N)dt/m, \tag{5.22}$$
$$\Delta^{N+1}X = X^N + V^{N+1} * dt. \tag{5.23}$$

The above equation is a linear system that is solved on the GPU.

5.5 GPU-based Deformation

To solve Equations 5.22 and 5.23 on the GPU, we need to perform matrix and vector operations, specifically vector additions, matrix inverse, matrix-vector and matrix products. The Hessian matrix used in equation 5.23 is large, sparse, and narrow-banded. As shown in [68], it is more efficient to compute the product of two sparse banded matrices diagonal-wise instead of the traditional column-wise computation method. This is done by vectorizing the matrices in terms of their diagonals.

For example, using the notation in [68], for $C = A \times B$, the main diagonal is designated to be the 0th diagonal. The kth diagonal above the main diagonal of C, c_k, with $k \geq 0$ is computed by deleting bottom k rows of A and top k rows of B^T - the transpose of B. The resulting $(n-k) \times n$ matrices are multiplied diagonal by diagonal (element-by-element) to form the $(n-k) \times n$ matrix D_k. The diagonals of D_k are added to form the resultant diagonal c_k. The kth diagonal below the main diagonal of C ,c_{-k}, for $k \geq 0$ is computed by deleting top k rows of A and bottom k rows of B^T and following the steps as above. Similarly, for a matrix-vector product $b = Ax$, each $-k$th diagonal of A is appended to its kth diagonal to obtain a vector d_k of the size of x. Each of the resulting diagonal vectors is multiplied element-by-element with x to obtain vectors d_k. Summing up these resultant vectors will give b. The algorithms are explained in detail in [68]. On the GPU, the matrices and vectors are represented as two-dimensional texture maps and fragment shader programs implement the above diagonal algorithm on them.

Now, let us consider the inverse component in Equation ??. Let us call this inverse component as W. $W = (I - dt^2 H/m)$. Using the approach in [80], we can use the Neumann polynomial method to approximate W^{-1} as

$$W^{-1} = 2D^{-1} - D^{-1}WD^{-1} \tag{5.24}$$

where D is a diagonal matrix of W. D is invertible as all its diagonal elements are non-zero. The diagonal elements of D^{-1} evaluate to the reciprocal of corresponding elements of D. This means that the inverse component can be obtained in terms of matrix-vector computations.

Therefore, Equations 5.22 and 5.23 are now

$$\Delta^{N+1}V = (2D^{-1} - D^{-1}(I - dt^2H/m)D^{-1})(F^N + dtHV^N)dt/m, \tag{5.25}$$

$$\Delta^{N+1}X = X^N + V^{N+1} * dt. \tag{5.26}$$

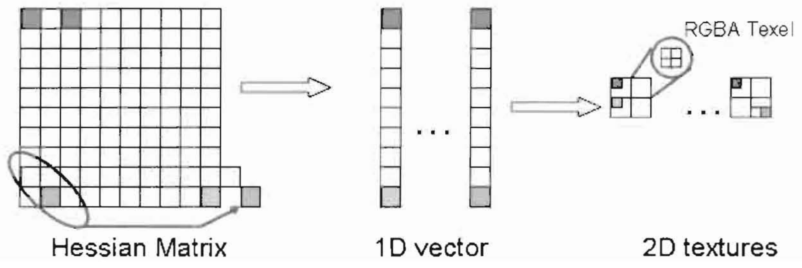
5.6 Implementation

As discussed in [45], rendering 1D textures is much slower than similar-sized 2D textures. Current graphic cards provide double the performance for 2D over 1D textures. Also, there are limitations with respect to the maximum size possible with 1D textures. We therefore make use of 2D texture maps in our implementation.

Therefore input data in the GPU are represented as three floating-point texture maps of size $\sqrt{N/4}$, N being the number of mesh nodes. The input data that these textures contain are the positions, velocities, and forces of the mass-spring nodes at each time step. The NxN hessian matrix is also stored in a 2D texture similar to the approach in [45]. As explained in the previous section, the non-null diagonals of the matrix are extracted and each diagonal from the ith column of the first row in the upper-half of the matrix is appended to its corresponding diagonal from the $N-i$th row of the first column in the lower half of the matrix. This allows for a convenient way of performing matrix vector operations. For ease of understanding, this approach is illustrated in Figure 5.2.

A single rendering pass can be defined as a general SIMD instruction wherein same operations are performed simultaneously for all pixels in an object. Equations 5.25 and 5.26 are implemented on the GPU using three rendering passes and using the diagonal algorithm for matrix-vector products. The matrix H and its inverse are constructed once for a given mesh and then used throughout the on-line update computation. The subsequent steps in the simulation are evaluated for each time step over multiple rendering passes. Once the linear equation is solved, a final rendering pass updates the velocities and positions of the mesh nodes. The result is fed back

Figure 5.2: Diagonal-wise Matrix Computations

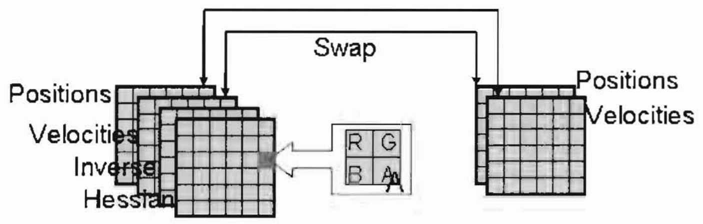


to the position and velocity textures, the input and output textures are swapped and the next iteration begins (Figure 5.3).

In order to reduce the storage size, 2×2 values are packed to a single texel (texel being defined as a pixel on a texture). Textures are also used for storing intermediate results of the simulation. The frame-buffer object and ping-pong rendering are used to feedback values computed in one iteration of the simulation to the next. A one-to-one mapping between geometry, texture, and pixel coordinates ensure proper control over the data accessed and computed.

The pseudocode of the GPU implementation is given in Figure 5.4. As can be seen in 5.4, one rendering pass computes temporary values while the remaining two passes compute the velocities and positions.

Figure 5.3: Multi-pass Rendering



```
**procedure** $ImplicitEuler(X^1, V^1, F^1, m, dt, k, row, col)$
 1: $H \leftarrow hessian(k, row, col)$ {hessian() computes stiffness matrix of a system with
    row*col nodes}
 2: $W \leftarrow (I - \frac{dt^2}{m} * H)$
 3: $D \leftarrow diagonal(W)$
 4: $W^{-1} \leftarrow 2D^{-1} - D^{-1} * W * D^{-1}$
 5: $n \leftarrow 1$
 6: **while** update(graphics) **do**
 7:     $Temp \leftarrow F^n + H * V^n * dt$
 8:     $V^{n+1} \leftarrow W * Temp * \frac{dt}{m}$
 9:     $X^{n+1} \leftarrow X^n + V^{n+1} * dt$
10:     $n \leftarrow n + 1$
11: **end while**{update() performs screen rendering}
```

Figure 5.4: Pseudocode of the GPU implementation

Chapter 6

Tactile Input and Feedback

....

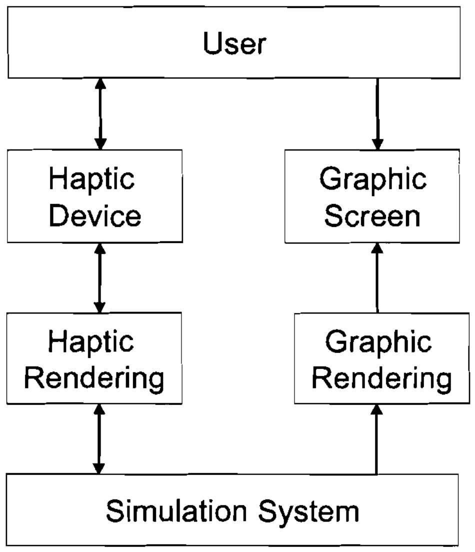
...

This chapter discusses integration of haptic interface with the GPU.

6.1 Introduction

The word Haptic pertains to the sense of touch. Haptic interfaces provide a sense of immersion to virtual reality environments and can enhance the learning process in surgical simulations [16]. Haptic devices like the PHANTOM® Omni™ are used as input devices to provide force-feedback to the user when virtual organs are deformed in the simulation. Figure 6.1 depicts the basic architecture of the simulation system and Figure 6.2 details integration of haptics and graphics loops in the system. As can be seen from the figures, multi-dimensional haptic input causes the mesh model to deform. This in turn leads to computing the forcs which is used to update the position of the model. Force is then fedback to the user. As a result, the user obtains an immersive sense of pushing the mesh model.

Figure 6.1: Basic architecture of the simulation system

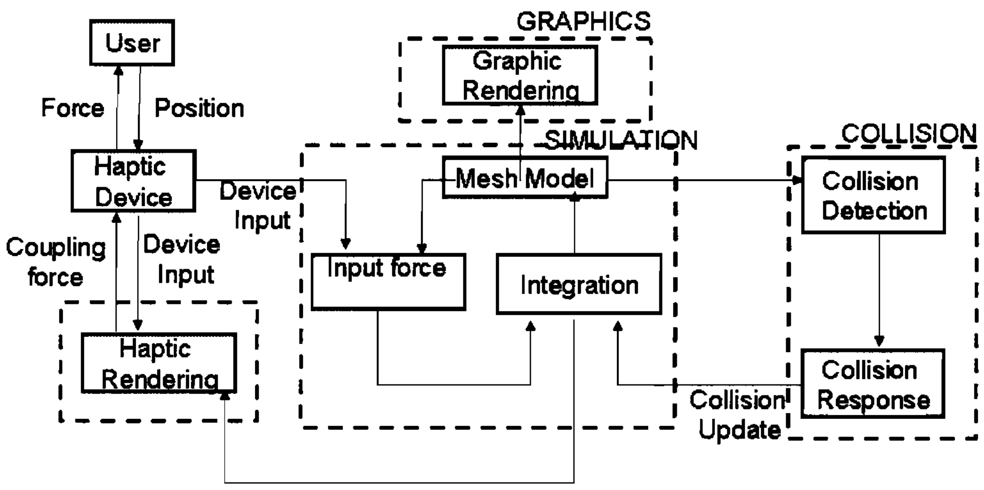


The haptic cycle in the simulation involves computation of the repulsive forces that are generated as a result of deformation of the virtual body. This feedback adds additional challenges to virtual simulation systems as it needs high update cycles (500-1000 Hz) for realistic user interactions.

6.2 GPU and Haptics

Haptics-based deformation using the GPU has been implemented in [60] [61] using shape functions that distribute the deformation forces of the neighboring nodes. These shape functions represent the shape that the surface will assume in the neighbourhood of the contact point due to deformation by external forces. A GPU-based simulation with haptic interaction was implemented by [83]. Both probing and grabbing operations were implemented with force computations being done entirely on the GPU. This method achieves an overall graphics frame rate of 30 Hz with the haptic frame rate or simulation rate of 450 Hz.

Figure 6.2: Detailed view of the haptics and graphics loops with virtual coupling



This thesis focuses on point-based haptic interaction with local deformation. A point-based method is implemented as a preliminary framework to include haptic interaction. Probing is the process of touching the virtual organ with a surgical tool. The haptic interaction in this thesis is provided through probing. The aim of implementing integration equations on the GPU in Chapter 5 was to improve simulation performance. With haptic interactions, the CPU needs to interface with the device and care must be taken to avoid bottlenecks in CPU-GPU data transfer.

6.3 Virtual Coupling

Virtual coupling is a technique frequently used to provide stability in interactive haptic simulations. This technique [74] [50] handles the communication between the controller of the haptic device and the simulation of the grasped object, enabling bidirectional interaction. Virtual coupling is implemented using a spring damper between the simulated object and the device (Figure 6.2).

The approach taken in this thesis to realize the software-based interaction was to introduce a damper effect everytime the haptic makes contact with the mesh model. A virtual positional anchor at the tip of the tool simulates the damper and updates its position at each time-step of the simulation. The simulation also obtains the haptic pointer position at each simulation step which is then used to compute the force that is applied to the virtual mesh model. The graphics loop typically runs at the rate of 30 Hz, which is quite low in comparison to the haptics loop (1 Khz). This means that the loops needs to synchronize with each other. This can be done by means of a synchronization mechanism that updates the positional inputs used by the virtual coupling. Each loop deals with a sampling of the corresponding damper positions. The damper used by the haptic device is attached to a virtual anchor that updates its position every time the graphics loop is stepped. Similarly, the graphics loop samples the device position before each step so that it can compute an input force to apply to the simulated body. Interpolation or extrapolation of the anchor position values prevent jerks. Such vibrations usually occur due to rapid fluctuations in force magnitude or direction.

We now describe the sequence of operations that run at each time-step, N, of the haptic cycle:

- Read 6 degree-of-freedom input of the haptic device at time t_N.
- Obtain the coupling force and torque at time t_{N-1}.
- Obtain the contact force and torque at time t_{N-1}.
- Compute the position of the virtual tool at time t_N.
- Compute the coupling force and torque at time t_N.
- Send the coupling force and torque to the device at time t_N

This sequence of operations is reflected in Figure 6.2.

On the simulation cycle front, the position of the virtual tool is obtained. Collision detection is then performed. If a collision is detected between the end-effector of the haptic stylus and the deformable model, the collision response loop computes the reaction forces that need to be sent to the haptic device.

Chapter 7

Results

Premature optimization is the root of all evil.

Donald Knuth

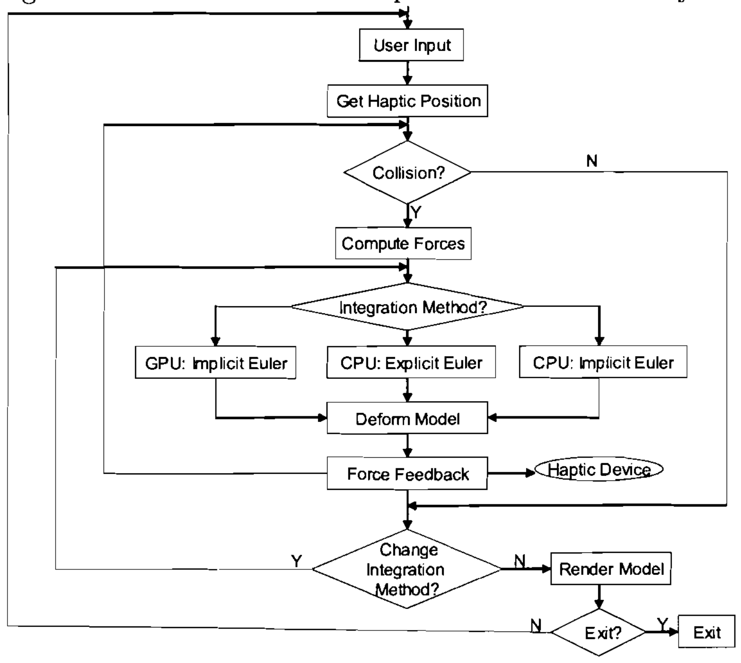
This chapter presents the results of the GPU-based surgical simulation framework. Comparison with the traditional CPU-based simulation systems will also be presented.

7.1 Overview

The simulation system was implemented in C++ using OpenGL and GLSL for the shader programs. Visual C++ was the development environment of choice. The system flowchart is presented in Figure 7.1. As mentioned previously, the integration method in VTE was a major bottleneck to real-time performance. To improve the system performance, explicit integration was replaced by an implicit integration scheme that was further parallelized using a GPU.

Three different simulation methods were implemented (a) simulation with GPU-based implicit euler integration, (b) simulation with CPU-based implicit euler integration and (c) simulation with CPU-based explicit euler. Such an implementation allows us

Figure 7.1: Flowchart of the Implemented Simulator System.



to compare performances of the three different methods and to analyze them. At any time of the simulation, the user can dynamically change the integration method of the simulation to qualitatively gauge the performance differences. These differences are quantitatively captured using various metrics as described later in this chapter.

The system initializes with a virtual instrument (representing the current haptic device position) and mass-spring model (representing the virtual organ). In what follows a single computation cycle will be explained.

In a given cycle, the system checks for a collision between the virtual instrument and the deformable organ. If a collision is detected, as explained in Chapter 2, the total force based on the mass-spring system equations is computed at the point of deformation. At this point an integration should occur to compute the deformations of the entire mass-spring system model. As explained earlier, the user is given a choice of integration scheme and processor for the same (this defaults to GPU-based implicit integration). The user-selected choice guides the integration and the resultant position and velocities for the mass-spring model causes it to deform. The resultant repulsive force due to this deformation is sent as feedback to the haptic device. Visual feedback is then provided through rendering of the model.

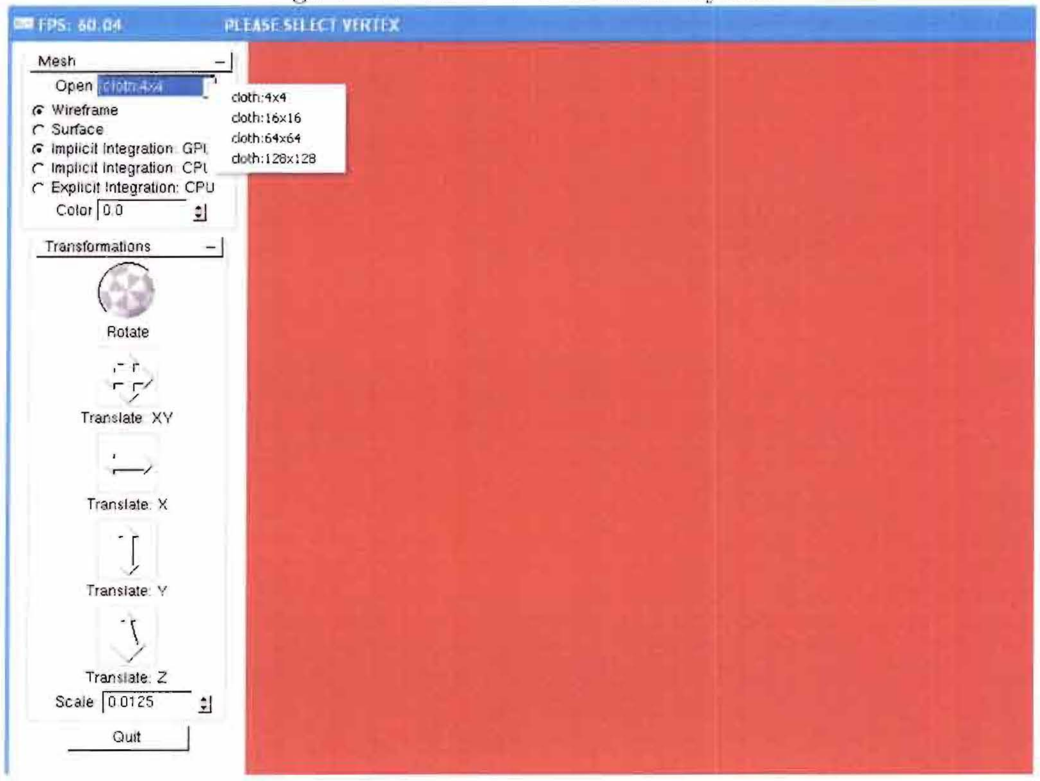
In the event of no collision being detected, the system continues to render the mesh on screen.

7.2 Simulation snapshots

This section describes the user interface and the mesh-model renderings of the simulation software. The user can begin with choosing the size of the mesh-model which ranges from 16×16 nodes to 256×256 nodes. In addition, there are two model rendering options: wireframe and surface. As mentioned earlier, the GUI provides three integration options to the user, namely "Implicit Integration:GPU", "Implicit

Integration: CPU" and "Explicit Integration: CPU". Finally, to navigate through the scene consisting of the mesh model, transformation controls such as 3D rotation, translation and scale are provided. These options can be seen in the left panel of the snapshot in Figures 7.2. Snapshots of various model sizes can be seen in Figures 7.3, 7.4 and 7.5.

Figure 7.2: GUI of the Simulator System.


FPS: 60.04

PLEASE SELECT VERTEX

Open cloth:4x4

[✓ Wireframe]

Surface

[✓ Implicit Integration: GPU]

Implicit Integration: CPU

Explicit Integration: CPU

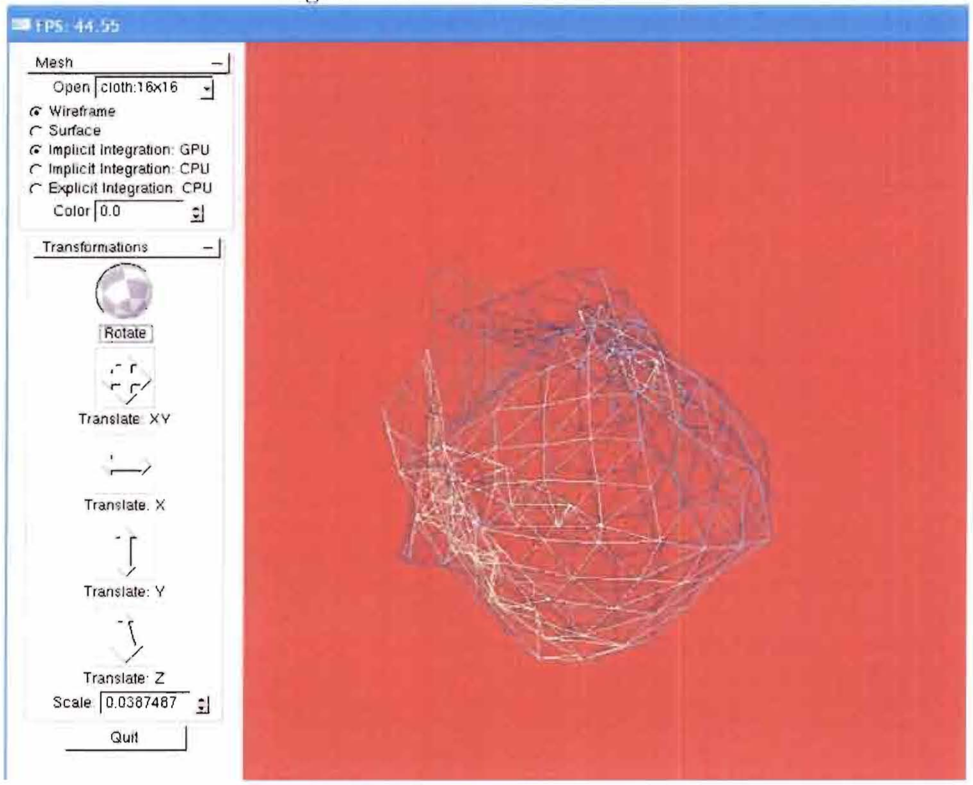
Color 0.0

Scale 0.0125

7.3 Performance

To measure performance of the simulator developed on the GPU, we make use of several benchmarks. One of them is to compare the integration method with the two

Figure 7.3: Initial 256-node Mesh.



FPS: 44.55

Mesh

Open cloth:16x16

[✓ Wireframe]

Surface

[✓ Implicit Integration: GPU]

Implicit Integration: CPU

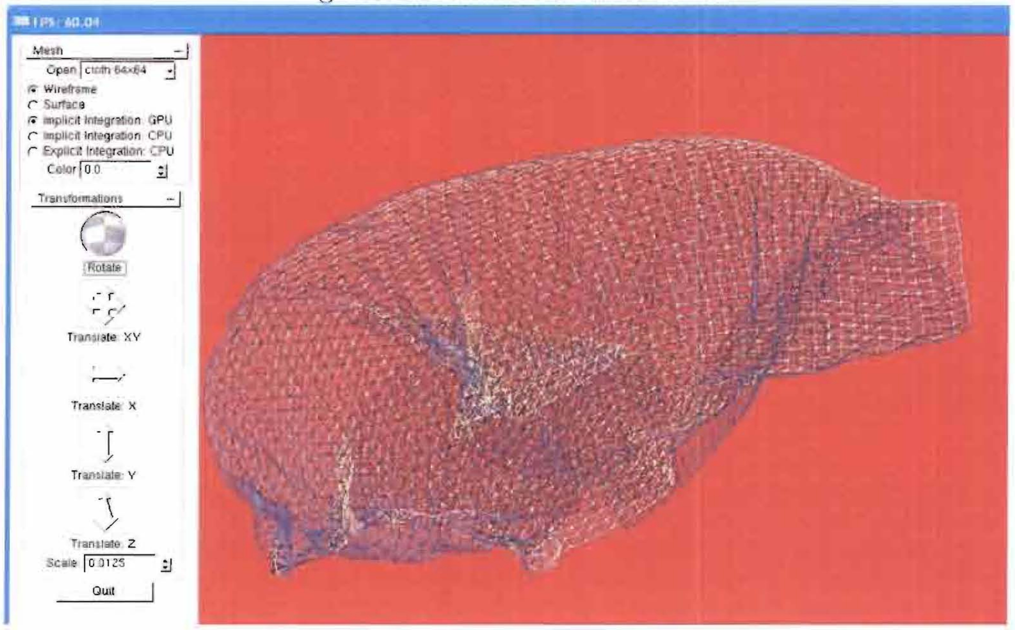
Explicit Integration: CPU

Color 0.0

Transformations

Scale 0.0387487

Figure 7.4: Initial 4096-node Mesh.


FPS: 60.04

Open: cloth 64x64

[✓ Wireframe]
Surface
[✓ Implicit Integration: GPU]
Implicit Integration: CPU
Explicit Integration: CPU
Color: 0.0

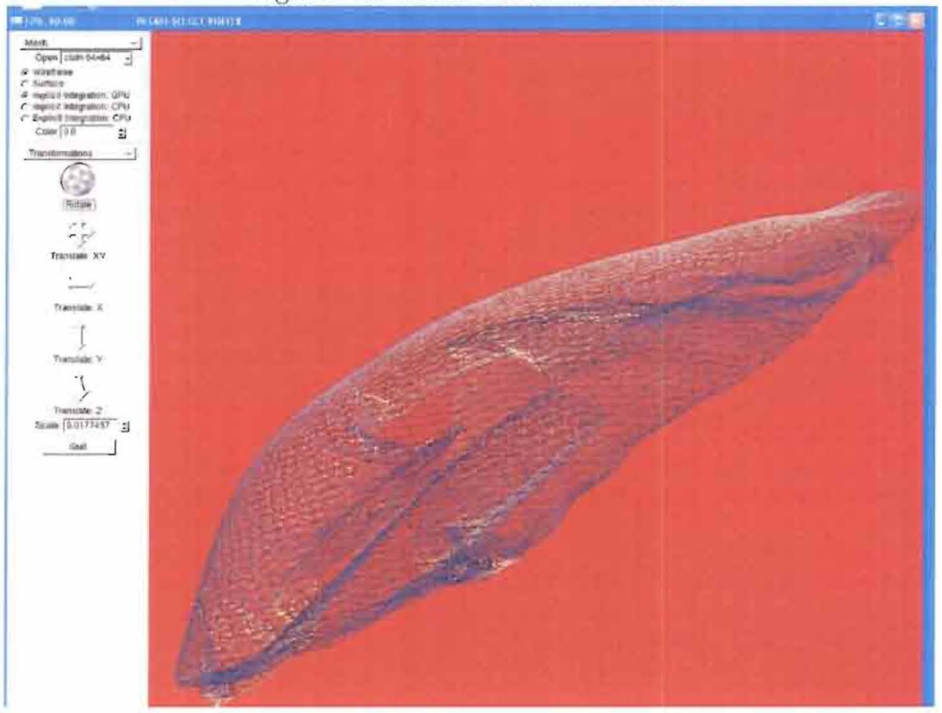
Scale: 0.0125

CPU-based integration schemes: implicit euler and explicit euler integrations. Explicit Euler method has been taken into account due to its usage on the VTE. Comparison of GPU-based implicit Euler with its CPU-based counterpart gives a measure of GPU speedup over CPU. On the other hand, comparison of CPU-based explicit Euler with GPU-based implicit Euler gives a measure of the differences between them.

The simulations were run on a 3.0 GHz Pentium 4 CPU with 1.0 GB RAM and an NVIDIA GeForce 7600 GS. To compute the runtime statistics, simulations were run over 10000 trials from which the mean runtime and its variance are computed and presented.

Table 7.2 shows the comparative results of the runtime for the three integration schemes: explicit euler on the CPU, implicit euler on the CPU and implicit euler on the GPU. For each of these schemes we report the integration runtime statistics for model sizes ranging from 16×16 nodes to 256×256 nodes. It should be noted

Figure 7.5: Initial 16384-node Mesh.



that for meaningful comparison, the GPU-based implicit euler runtime includes data-transfer overhead time (between CPU and GPU) as well. Details of this breakdown for the same node-sizes is shown in Table 7.3.

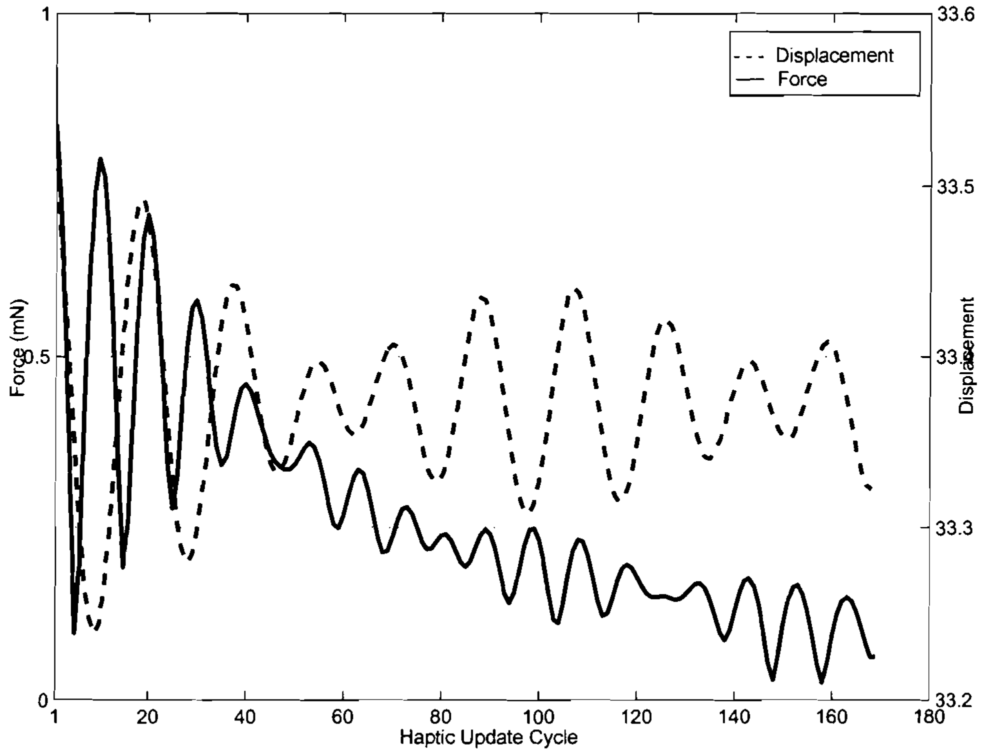
The Frame Buffer Object (FBO) architecture is an extension available in OpenGL for off-screen rendering. As explained in Chapter 5 this thesis makes use of the FBO for rendering to textures. The traditional framebuffer is turned off and the off-screen buffer, framebuffer object, captures the image data being rendered. The velocities and positions of the data model computed at each simulation are fed-back to the textures. These values are then read-back from the FBO and are used as input to the next simulation step, resulting in fast multiple rendering passes. This essentially breaks down to OpenGL's glReadPixels() function call. FBO is efficient as it does not use context switching and can make use of additional buffers like depth and stencil. The FBO also provides full precision and eliminates clamping issues as in the traditional framebuffer. Table 7.3 provides the time taken to read/write the position/velocity data from the GPU at each simulation time-step.

To add further insight into the runtimes of other components of the simulator, we have provided similar runtime statistics for broad-phase and narrow-phase collision-detection and the off-line Hessian matrix computation. Broad phase identifies subsets of objects that may be colliding and excludes those that definitely are not colliding. Narrow phase performs pair-wise intersection tests within these subsets.

We also evaluated the simulations by acquiring the force and displacement values during deformation for each of the three mesh sizes using each of the three integration schemes. For each simulation, a part of the mesh was probed. Upon deformation, the magnitude of the force and displacement values of the mesh node closest to the virtual probe tool were acquired. The results are presented as force-displacement dual axes graphs i.e. force values along a Y axis, displacement values along another Y axis and haptic update cycle against the X axis. Figures 7.6, 7.7 and 7.8 depict graphs plotted using GPU-based Implicit Euler integration scheme. Figures 7.9, 7.10 and

7.11 depict graphs plotted using CPU-based Implicit Euler integration scheme and Figures 7.12, 7.13 and 7.14 depict graphs plotted using CPU-based Implicit Euler integration scheme. As can be seen from the graphs, the fluctuations in displacement and force occur at a much faster rate with GPU-based implicit integration than with their CPU counterparts. Moreover, fluctuations with the CPU-based implicit scheme occur at a faster rate than with the CPU-based explicit scheme.

Figure 7.6: Force and Displacement Graphs during Probing (probe tool used to pull a single mesh node) of a 256-node mesh for GPU-based Implicit Euler Scheme



Haptic update rate refers to the frequency at which force is computed and sent to the human user via a haptic device. Our simulation system interfaces the algorithm with the haptic device and therefore, the haptic update rate is a key parameter that synchronizes the haptic loop with the graphics loop and influences the simulation

Figure 7.7: Force and Displacement Graphs during Probing (probe tool used to pull a single mesh node) a 4096-node mesh for GPU-based Implicit Euler Scheme

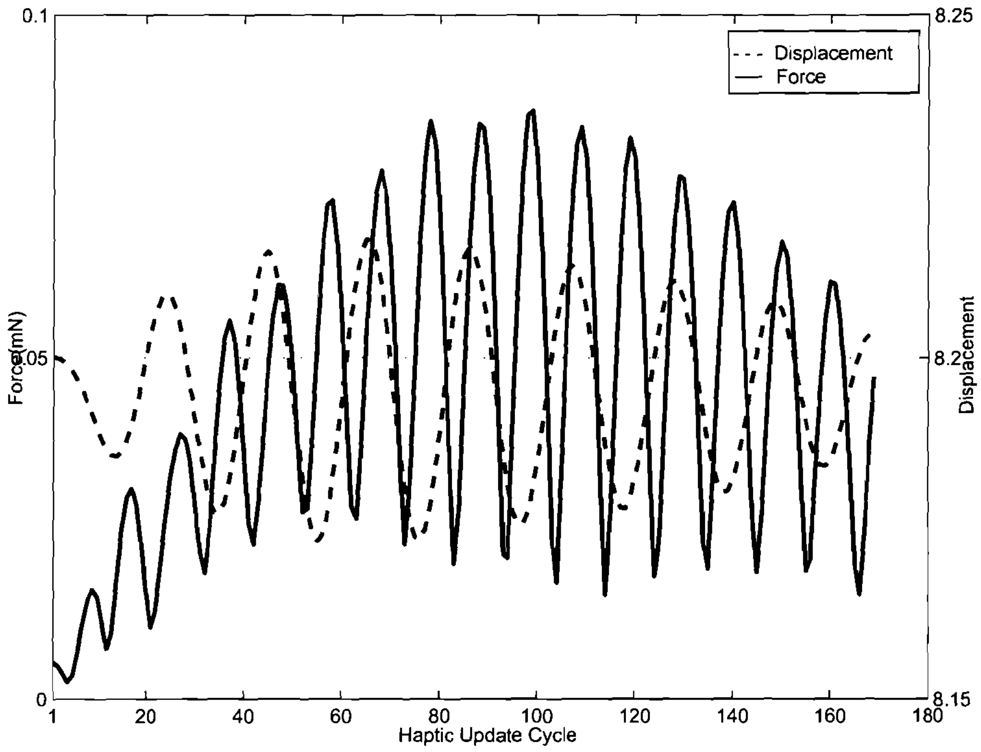


Figure 7.8: Force and Displacement Graphs during Probing (probe tool used to pull a single mesh node) a 16384-node mesh for GPU-based Implicit Euler Scheme

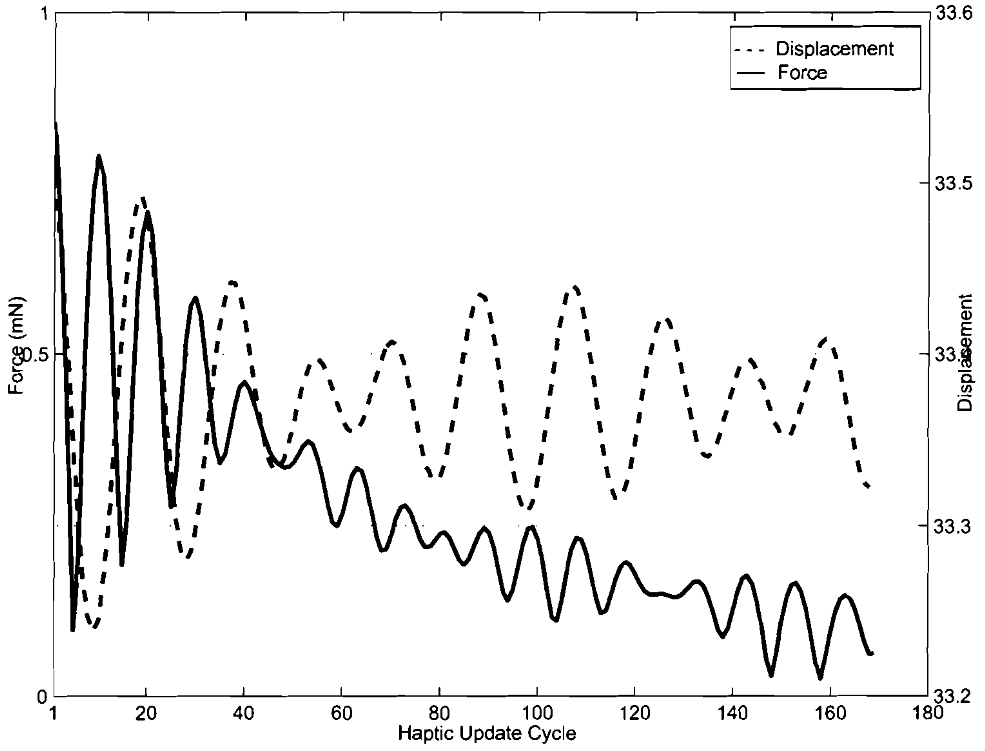


Figure 7.9: Force and Displacement Graphs during Probing (probe tool used to pull a single mesh node) a 256-node mesh for CPU-based Implicit Euler Scheme

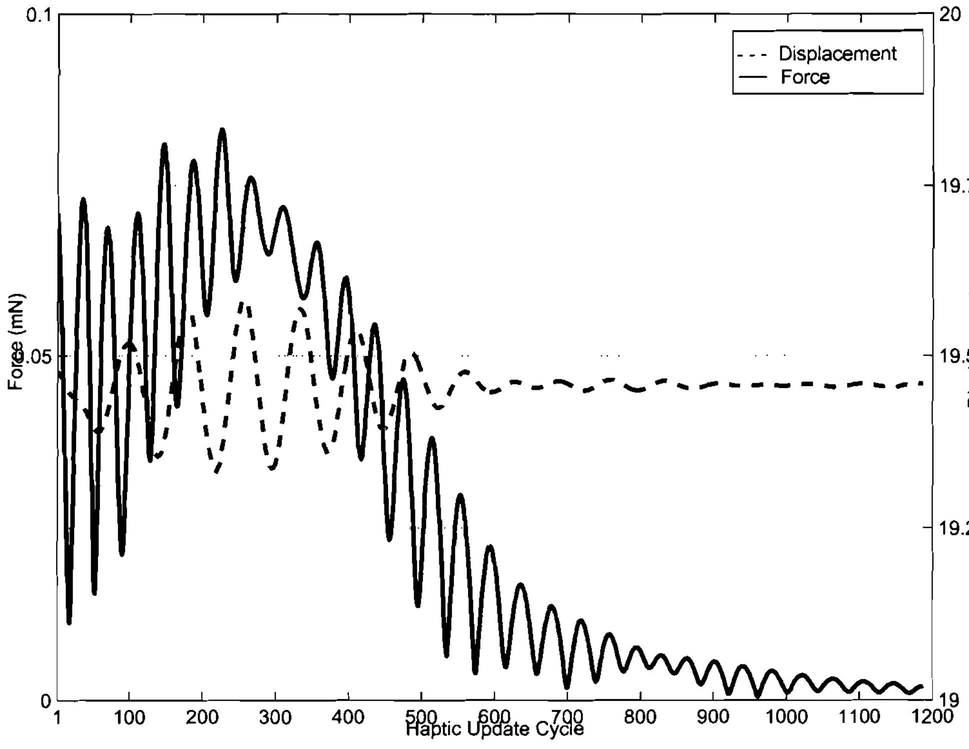


Figure 7.10: Force and Displacement Graphs during Probing (probe tool used to pull a single mesh node) a 4096-node mesh for CPU-based Implicit Euler Scheme

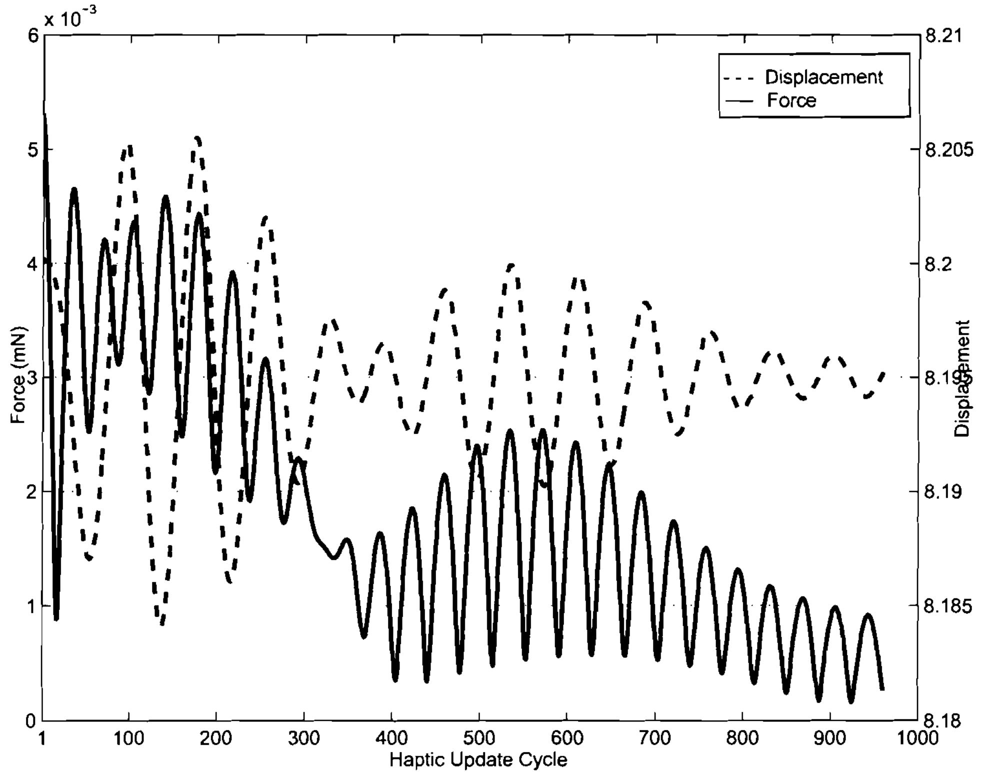


Figure 7.11: Force and Displacement Graphs during Probing (probe tool used to pull a single mesh node) a 16384-node mesh for CPU-based Implicit Euler Scheme

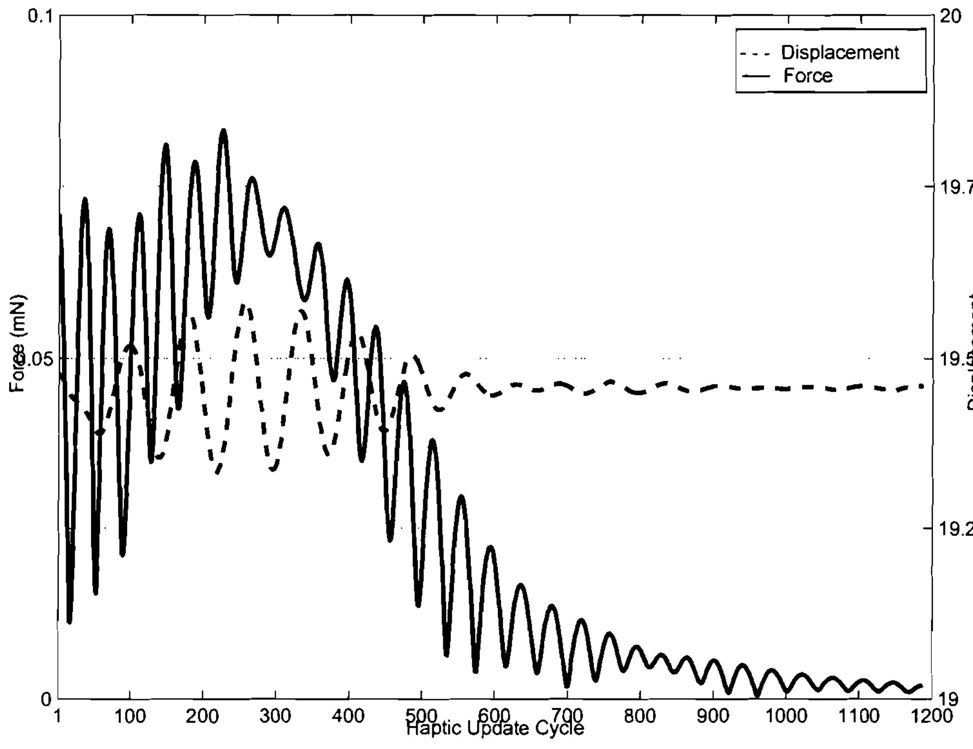


Figure 7.12: Force and Displacement Graphs during Probing (probe tool used to pull a single mesh node) a 256-node mesh for CPU-based Explicit Euler Scheme

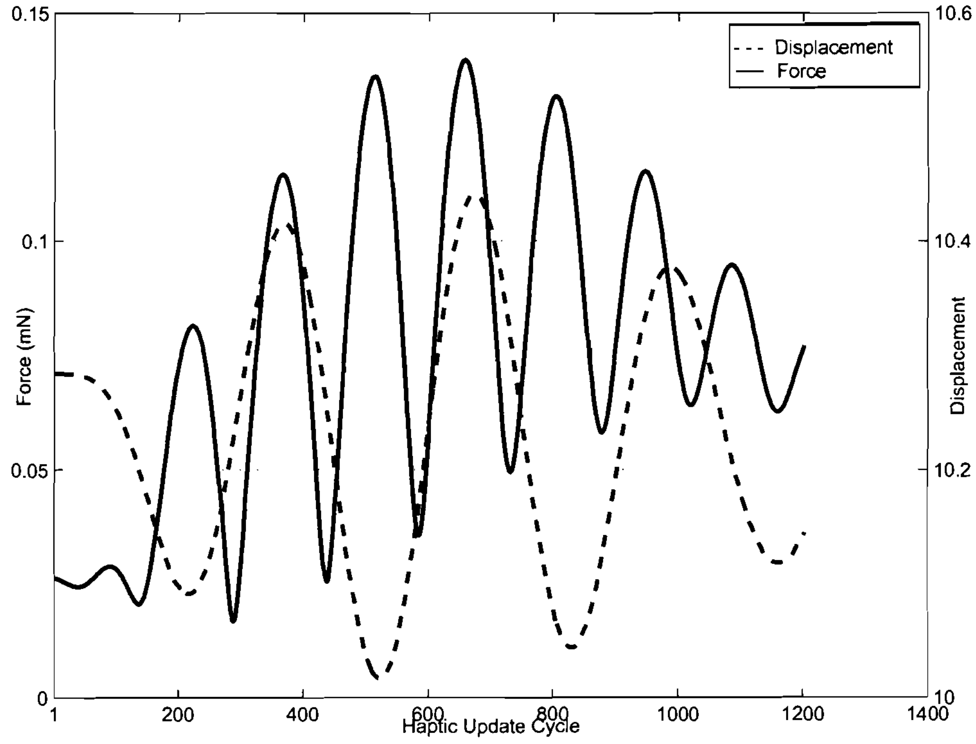


Figure 7.13: Force and Displacement Graphs during Probing (probe tool used to pull a single mesh node) a 4096-node mesh for CPU-based Explicit Euler Scheme

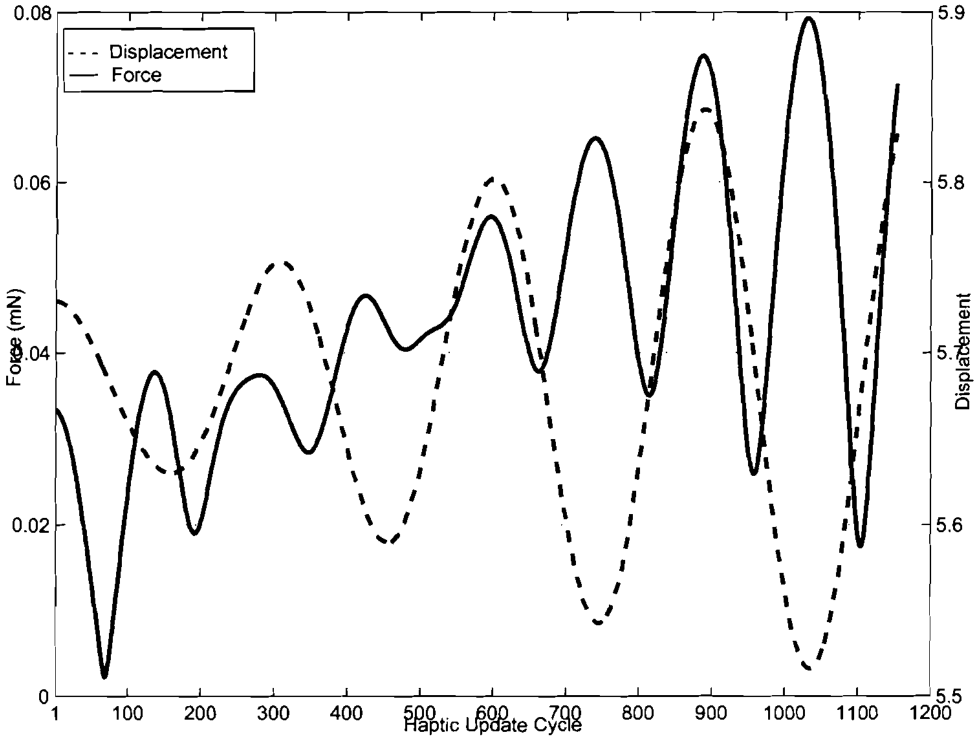


Figure 7.14: Force and Displacement Graphs during Probing (probe tool used to pull a single mesh node) a 16384-node mesh for CPU-based Explicit Euler Scheme

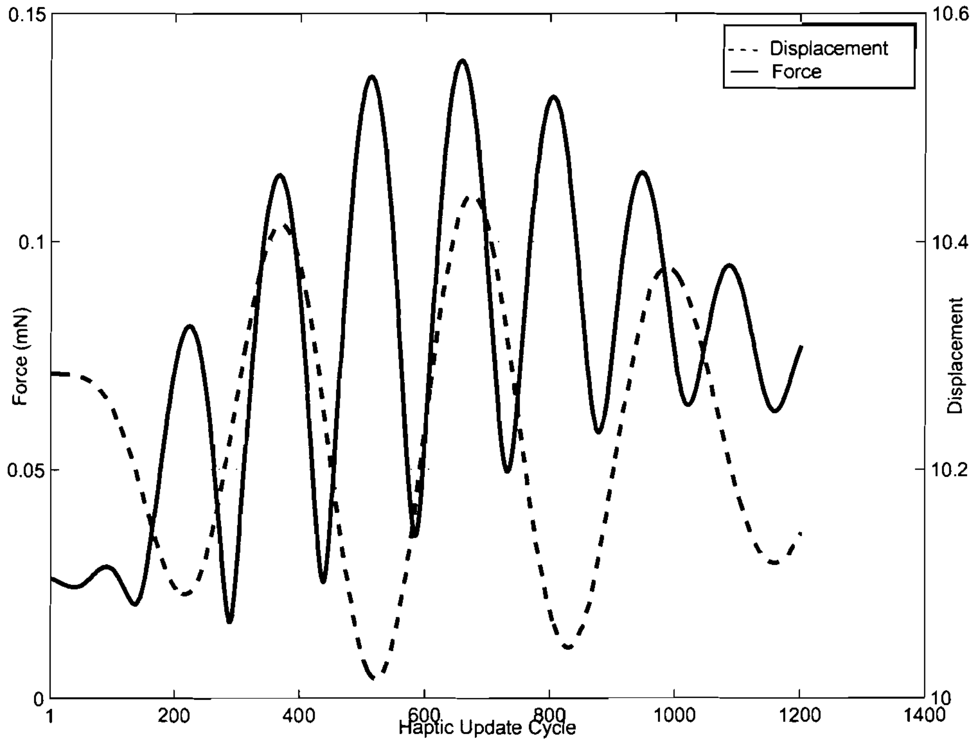


Table 7.1: Haptic Update Rate (in Hz) for Each Integration Scheme Against Each Mesh Size.

Integration Scheme	256 nodes	4096 nodes	16384 nodes
GPU-based Implicit Integration	560	504	305
CPU-based Implicit Integration	611	347	212
CPU-based Explicit Integration	597	271	148

performance. Previous research suggests that a minimum update rate of 1 kHz is required for rendering rigid objects, but lower rates may suffice for deformable ones [60]. We have computed the haptics update rates for each of the integration methods and each mesh-model and the results are presented in Table 7.1. As can be seen in the table, a maximum update rate of 560 Hz was achieved with the GPU-based implicit integration scheme.

As can be seen from these tables and figures, the general trend is that

- For all node sizes, implicit Euler integration is faster than explicit Euler integration. This is consistent with previous reports in literature[63][84].
- For all node sizes, GPU-based implicit Euler integration is faster than its CPU counterpart. The maximum speedup achieved is about a factor of 6.5 for model size of approximately 16000 nodes, which is the node size of the largest simulated mesh model.
- While table in Figure 7.2 shows that the GPU-based integration is slower than its CPU counterpart for the 256-node mesh size, a closer inspection of table in Figure 7.3 reveals that the data latency time causes this delay. The true integration time on the GPU is in fact faster than that of the CPU. Therefore, for small mesh sizes, data latency is a huge bottleneck and outweighs the benefits of GPU-based integration.
- For larger node sizes, GPU-based implicit Euler integration allows faster haptic update rates over its CPU counterpart. The maximum update rate achieved is

Table 7.2: Mean and variance of the time taken (in ms) over 10000 simulation trials to solve each integration method on the corresponding processor.

Integration	Processor	Mesh Size					
		256		4096		16384	
		Mean	Variance	Mean	Variance	Mean	Variance
Explicit Euler	CPU	1.48	0.19	5.48	1.88	20.74	0.12
Implicit Euler	CPU	0.73	0.001	2.23	0.02	8.88	0.07
Implicit Euler	GPU	1.77	0.010	1.98	0.018	3.27	0.071

560 Hz, bringing it closer to the target 1 KHz update rate.

- For each of the integration schemes and processors, runtime increases with increase in node size. This is a result of increase in integration times in general as well as data transfer times specifically in the case of GPU-based scheme.

- The benefits of parallelism (GPU) increases with increase of node, benefit being used here to mean the ratio of VTE's CPU-based Explicit Euler runtime to GPU-based Implicit Euler runtime. This is because small node sizes can be impeded with overheads of graphic drivers as well as data transfer rate.

- Reading back data from the GPU is slower than sending data to the GPU. This is due to different openGL calls being made for these operations - glReadBuffer() and glReadPixels() for reading back GPU data and glTexSubImage2D() for writing data to the GPU. This is in agreement with established research [19] and therefore is an important factor to consider in the design of an algorithm for GPUs.

Table 7.3: Mean and variance of the time taken (in ms) over 10000 simulation trials to solve each sub-step of GPU-based implicit Euler integration.

Sub-step	Mesh Size					
	256		4096		16384	
	Mean	Variance	Mean	Variance	Mean	Variance
Data transfer to GPU	1.18	0.007	1.22	0.011	1.68	0.014
Data transfer from GPU	0.023	0.00017	0.063	0.0011	0.391	0.011
Integration Time	0.57	0.0037	0.703	0.0059	1.20	0.046

Table 7.4: Mean and variance of the time taken (in ms) over 10000 simulation trials to solve sub-steps on the CPU for GPU-based implicit Euler integration. Variance $\geq 10^{-4}$ ms is given as 0.Broad phase collision detection identifies subsets of objects that may be colliding and excludes those that definitely are not colliding. Narrow phase collision detection performs pair-wise intersection tests within these subsets.

Time Taken (ms)	Mesh Size					
	256		4096		16384	
	Mean	Variance	Mean	Variance	Mean	Variance
Broad Phase	0.001	0	0.001	0	0.002	0
Narrow phase	0.003	0	0.003	0	0.003	0
Implicit Euler	0.11	0	1.28	0	7.29	0.1

Chapter 8

Conclusion

The path comes into existence only when we observe it.

Heisenberg

This chapter concludes this thesis.

8.1 Conclusion

This thesis presents the development and analysis of a GPU-based simulation system to counter current performance bottlenecks in the *Virtual Training Environment (VTE)* developed at Simon Fraser University.

Specific contributions towards this are:

- Analysis of *VTE* and identification of its performance bottlenecks. It was shown that the collision detection algorithm and the use of a single thread for CPU-intensive tasks were the main bottlenecks.
- Design and development of a GPU-based simulation system using Implicit Euler integration scheme. This integration scheme was chosen to achieve stable simulation of deformable anatomical structures, thereby improving the accuracy

of the simulations. To further enhance performance of the simulation, modern GPU architecture was investigated and its parallel capabilities were exploited. Further research led to the understanding and use of suitable GPU techniques such as Framebuffer Objects (FBO), MRT (Multiple Render Targets), Multi-pass Rendering and Ping-pong Technology.

- Implementation and comparison of three integration schemes for the simulator viz. GPU-based implicit euler, CPU-based implicit euler, and CPU-based explicit euler schemes. To quantitatively measure performance gains using the GPU and implicit integration, we used several metrics, including integration time and haptic update rate.

- Qualitative comparison showed definite performance gains by the GPU-based implicit scheme followed by CPU-based implicit scheme over CPU-based explicit scheme.

- Demonstration of fastest integration time using the GPU-based implicit Euler scheme with a maximum speedup of 6.5 achieved for model size of approximately 16000 nodes. This speedup was achieved in spite of data-transfer latencies. The GPU-based system consistently outperformed the CPU-based systems across multiple mesh-sizes.

- Implementation of GPU-friendly haptic interaction with damper-based virtual coupling. It was demonstrated that the haptic update rate was the highest for the GPU-based implicit scheme with the maximum rate achieved being 560 Hz. The update rate is almost double that of the CPU-based explicit simulation system.

- Use of a semi-automated surface wrapping algorithm to obtain closed surface meshes from 2D planar square meshes. Relaxation methods minimize distortion and overlapping in the closed surface mesh.

From the results, we can conclude that for most simulation systems the limiting factor will be the cost of CPU-GPU communication (even for systems with small mesh-sizes).

Following the successful implementation of the integration scheme on the GPU and having obtained better performance over its CPU counterparts, we expect haptic update rates of much closer to 1 KHz if the entire simulation is implemented on the GPU. A higher performance gain is also expected if data latency were reduced or if multiple GPUs were used as co-processors.

8.2 Future Work

With the proposed GPU-based simulation system, several areas can be explored.

The focus of the thesis was to utilize the GPU and improve the performance time of the existing explicit euler simulation. Hence, a crude method was used for wrapping a 2D square object mesh around a three-dimensional model. The wrapping method used in this thesis has the potential to introduce wrinkles and foldovers in the mesh. An accurate alternative approach would be to reconstruct the three-dimensional model using 2D surface patches[30]. Although it would be very valuable in the long run for further extensions to this project, it is not within the scope of this thesis and therefore was not explored extensively.

The realism in the modeling of deformable bodies can be further extended by importing MRI images of patients into the simulation system [76].

In this thesis, we have only briefly discussed collision detection. As has been shown in previous work [12], [8] collisions between moving and complex polygonal objects can be determined quite efficiently on the GPU. In addition, self-collisions can also be included.

Scalable graphics systems, like CPU clusters, can be used to further performance of the simulation system. They are advantageous due to their additional hardware capabilities. A few years ago, NVIDIA and ATI introduced dual-GPU configurations through SLI(Scalable Link Interface) [8] and Crossfire [29] respectively. However, the drawback in using these hardware systems was the lack of reliable software and programming models to utilize their capabilities. Also, these systems scaled pixel and triangle rates but not texture memories posing challenges typical of shared memory data parallel systems. For example, boiling simulations implemented using multiple GPUs obtained lower performance than their single-GPU counterparts due to increased communications over texture accesses [13]. In [32], the focus of their unsteady flow visualization implementation was on improving its quality and accuracy and they achieve frame-rates of a mere additional 10 to 50 frames with dual GPUs. However, [57] scaled applications using the recently introduced Havok physics API for GPUs. An entire GPU Card was dedicated to non-gameplay physics calculations obtaining results 10 times faster than their corresponding CPU implementations. Therefore, multiple GPU systems can be explored to provide higher performance gains if provided with strong software support like Havok.

Appendix A

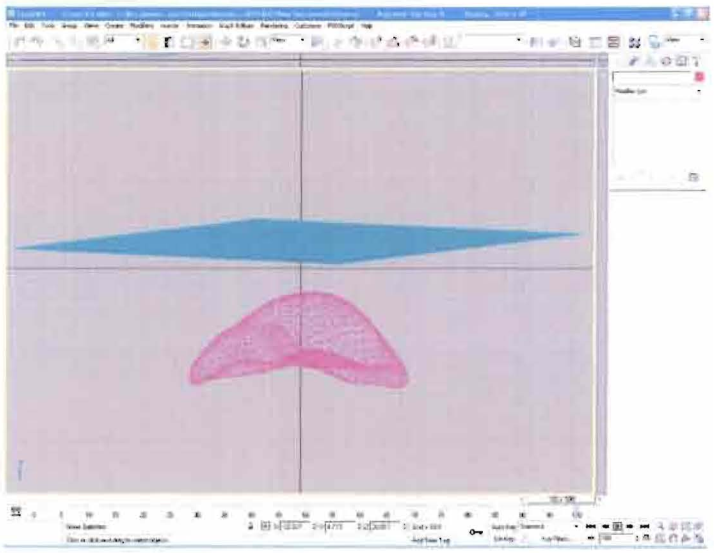
Surface Wrapping with Autodesk 3DS Max

The *Cloth Modifier* available in Autodesk 3DS Max is used in an interactive fashion to implement surface wrapping for closed meshes. The details of the implementation are provided below.

A.1 Implementation steps

Step 1:
The scene in Figure A.1 contains an imported VRML (Virtual Reality Modeling Language) organ file (in this case liver model) and a two-dimensional mesh. In this case, the organ is a triangular closed mesh and of genus zero. The two-dimensional mesh is of size 4096 nodes.
Step 2:
In Figure A.2, the two-dimensional mesh is selected and a *Cloth Modifier* is applied to it from the *Modifier* list. *Object Properties* in its corresponding interface panel can be selected to display and modify the mesh properties. In this case, the *thickness, repulsion, offset* and *cling* parameters are edited. The parameter definitions [3] as well as values used are as follows:
Thickness: the virtual thickness of a mesh for the purpose of detecting mesh-to-mesh

Figure A.1: Initial state of input mesh and closed surface.



collisions.

Thickness value: 0.2

Repulsion: The amount of force used to repel other mesh objects.

Repulsion value: 2.0

Offset: the amount of distance kept between the mesh and the collision object.

Offset value: 0.5

Cling: the extent to which the mesh adheres to a collision object.

Cling value: 500

Step 3:

In Figure A.3, the mesh describing the 3D object that we wish to wrap (in this case a liver) is added to the *Cloth Modifier* and set as a *collision object.* This is done by selecting *Cloth* radio-button in the *Object Properties* interface panel.

Step 4:

In Figure A.4, the *Simulate* button is activated to begin the wrapping process. The

Figure A.2: Input mesh set as *Cloth* object.

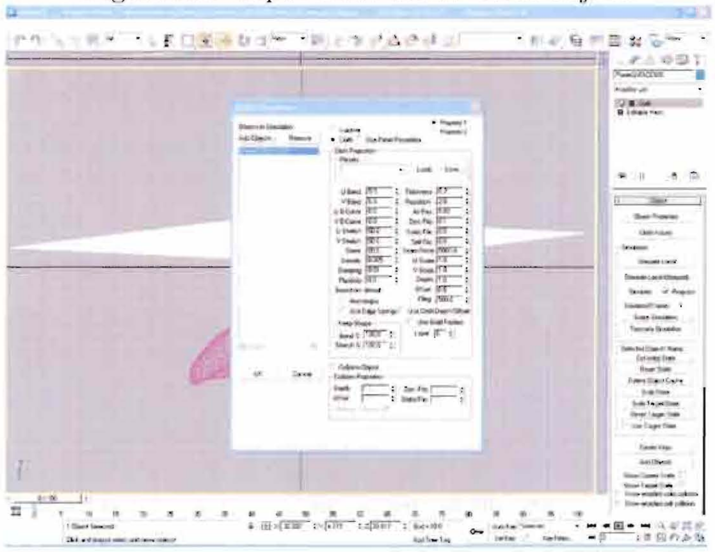
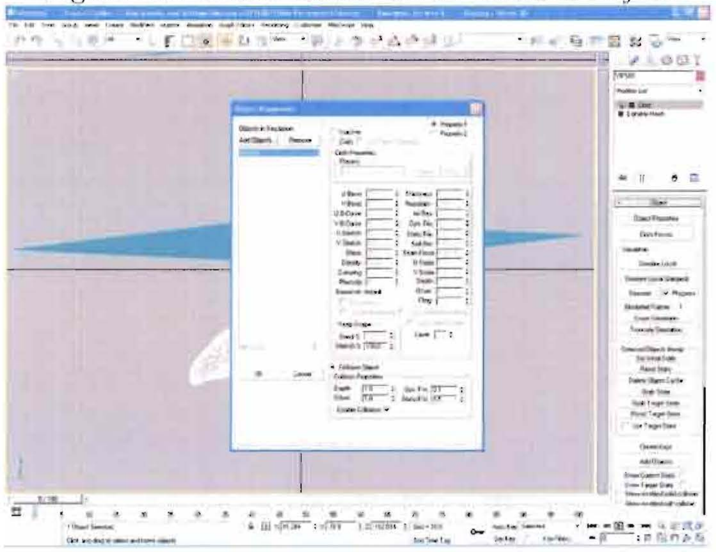
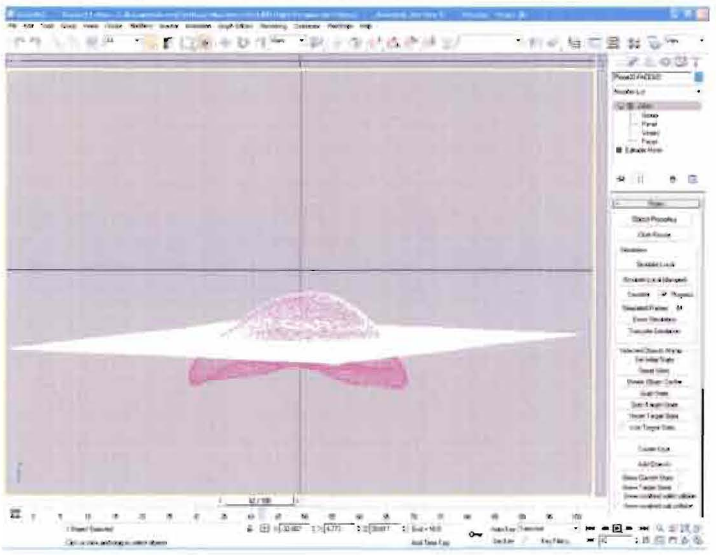


Figure A.3: Closed surface set as *Collision Object.*



first simulation can cause errors due to inappropriate simulation settings. Trial and error fixing of setting-values can rectify the simulation. In case of erroneous simulation, *Erase Simulation* can be activated and the simulation restarted.

Figure A.4: *Simulate* achieves wrapping.

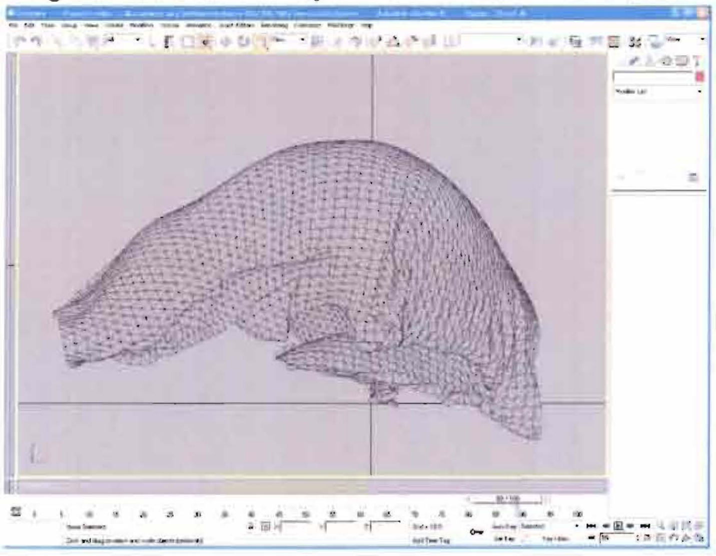


Step 5:

In Figure A.5, relaxation steps can be manually adjusted to achieve desired closed surface shape. A *Relax Parametric Modifier*, with *Keep Boundary Pts fixed* unchecked, applied to the converted mesh reduces fold-overs.

In conclusion, simulating cloth in $3D$ can be implemented using 3DS Max. However, it must be noted that trial and error of simulation settings is crucial to obtaining a visually accurate wrapping.

Figure A.5: *Relax Modifier* reduces foldovers in mesh.



Appendix B

Collision Detection Algorithm

The motion of the surgical tool can be tracked as a line segment and the primitives in the deformable bodies are triangles. Hence, collision detection between the tool and the body can be achieved using line-triangle intersection test.

B.1 Line-Triangle Intersection Test

The test involves finding the intersection between the line, that the segment is part of, and the plane containing the triangle. The first step is to ensure that the line and the plane are not parallel to each other. The next step is to determine a point on the line that lies within the triangle.

B.1.1 Implementation

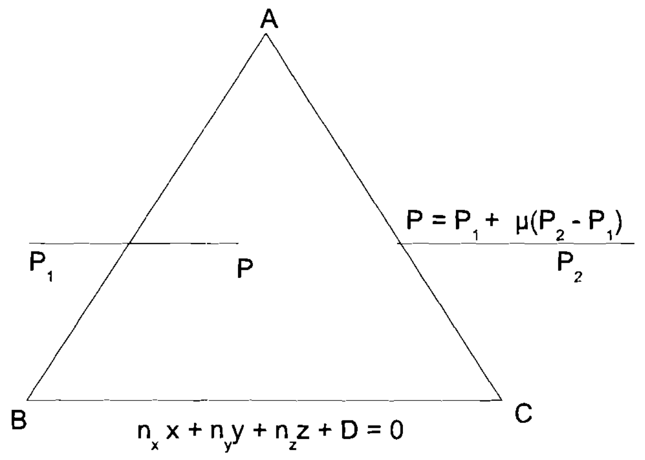
In Figure B.1, $\bar{P_1P_2}$ represents the line segment while ABC depicts the triangle. The equation for the line segment can be given as

$$P = P_1 + \mu(P_2 - P_1) \tag{B.1}$$

where P is the intersection point. The equation for the plane is given by

$$n_x x + n_y y + n_z z + D = 0 \tag{B.2}$$

Figure B.1: Line-triangle Intersection



(n_x,n_y,n_z) being the normal vector of the plane. The cross-product of any two normalized edge vectors will yield this normal.

$$(n_x, n_y, n_z) = (B - A) \times (C - A). \tag{B.3}$$

D can be found by vertex substitution of above equation into B.2.

$$D = -(n_x A_x + n_y A_y + n_z A_z) \tag{B.4}$$

Using the above equations, μ can be expressed as

$$\mu = \frac{D + n_x P_{1x} + n_y P_{1y} + n_z P_{1z}}{(n_x(P_{1x} - P_{2x}) + n_y(P_{1y} - P_{2y}) + n_z(P_{1z} - P_{2z}))} \tag{B.5}$$

If the denominator in B.5 is 0, the line is parallel to the plane and there is no intersection. If μ is between 0 and 1, the intersection point lies on the line segment [36].

In order to test if the intersection point P lies within the triangle ABC, we need to compute the sum of the internal angles. If the total sum is 2π, P lies inside the triangle while if the total sum is lower, P lies outside the triangle.

If the unit vectors p_1, p_2, p_3 are as follows:

$$\begin{aligned} p_1 &= \frac{(A-P)}{|(A-P)|} \\ p_2 &= \frac{(B-P)}{|(B-P)|} \\ p_3 &= \frac{(C-P)}{|(C-P)|} \end{aligned} \tag{B.6}$$

the angles are

$$\begin{aligned} a1 &= a\cos p_1 p_2 \\ a2 &= a\cos p_2 p_3 \\ a3 &= a\cos p_3 p_1 \end{aligned} \tag{B.7}$$

Floating point error of 0.05 is considered for efficiency [36].

Bibliography

[1] http://www.glowcode.com.

[2] http://www.nvidia.com.

[3] 3dsmax. http://www.autodesk.com.

[4] Directx. http://msdn.microsoft.com/directx/.

[5] Opengl. http://www.opengl.org.

[6] Perfhud. http://developer.nvidia.com/object/nvperfhud.html.

[7] Microsoft shader debugger. http://msdn.microsoft.com, 2005.

[8] Nvidia gpu programming guide. http://download.nvidia.com/developer/, July 2005.

[9] Graphic remedy gdebugger. http://www.gremedy.com/, 2006.

[10] Cover S. A., Norberto F. E., OBrien J. F., Rowe R., Gadau T., and Palm E. Interactively deformable models for surgery simulation. In *IEEE Computer Graphics and Applications*, volume 13, pages 68–75, 1993.

[11] Gates E. A. New surgical procedures: can our patients benefit while we learn? In *Am J Obstet Gynecol*, volume 176, page 12931298, 1997.

[12] Kolb A., Latta L., and Rezk salama C. Hardware-based simulation and collision detection for large particle systems. In *Graphics Hardware*, pages 123–132, 2004.

[13] Moerschell A. and Owens J.D. Distributed texture memory in a multi-gpu environment. In *Graphics Hardware*, 2006.

[14] Moller T. A. and Haines E. *Real-time Rendering.* AK Peters, 2005.

[15] Quarteroni A., Sacco R., and Saleri F. *Numerical Mathematics.* Springer, 2007.

[16] Mandayam A.S. and Cagatay B. Haptics in virtual environments: Taxonomy, research status, and challenges. In *Haptic Displays in Virtual Environments and Computer Graphics in Korea*, volume 21, pages 393–404.

[17] U.M. Ascher and L.R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998.

[18] Starfield B. Is us health really the best in the world? In *the journal of the American Medical Association*, volume 284, pages 483–485, 2000.

[19] M. Blom and P. Follo. Vhf sar image formation implemented on a gpu. *Geoscience and Remote Sensing Symposium, 2005. IGARSS '05. Proceedings. 2005 IEEE International*, 5:3352–3356, July 2005.

[20] Basdogan C. and Srinivasan M.A. *Virtual Environments HandBook*, chapter Haptic rendering in virtual environments, pages 117–134. 2001.

[21] Grimm C. Simple manifolds for surface modeling and parametrization. In *Shape Modeling International*, 2002.

[22] Lombardo J. C., Gascuel M. P., and Neyret F. Real-time collision detection for virtual surgery. In *Proceedings of Computer Animation*, pages 33–39, 1999.

[23] Baraff D. and Witkin A. Large steps in cloth simulation. In *Computer Graphics Proceedings, Annual Conference Series*, pages 43–54, 1998.

[24] Terzopoulos D., Platt J.C., Barr A.H., and Fleischer K. Elastically deformable models. In *Computer Graphics*, volume 21, pages 205–214, 1987.

[25] Trebilco D. Glintercept. http://glintercept.nutty.org, 2006.

[26] Anastakis D.J., Wanzel K.R., and Brown M.H. et al. Evaluating the effectiveness of a 2-year curriculum in a surgical skills center. volume 185, page 378385, 2003.

[27] Breen D. E., House D. H., and Wozny M. J. Predicting the drape of woven cloth using interacting particles. In *SIGGRAPH Conference Proceedings*, pages 365–372, 1994.

[28] Hoff K. E., Keyser J., Lin M., Manocha D., and Culver T. Fast computation of generalized voronoi diagrams using graphics hardware. In *Computer Graphics (Annual Conference Series)*, volume 33, pages 277–286, 1999.

[29] Persson E. Programming for crossfire, 2005.

[30] A. Elnagar and A. Basu. From 2d surface patches to 3d reconstructed models: Theory and applications. *PR*, 31(10), October 1998.

[31] Cohen F.S., Walid I., and Chuchart P. Ordering and parameterizing scattered 3d data for b-spline surface approximation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(6):642–648, 2000.

[32] Li G., Tricoche X., and Hansen C. Gpuflic: Interactive and accurate dense visualization of unsteady flows. In *Eurographics*, pages 1–6, 2006.

[33] Shen G., Zhu L., Li S., Shum H.Y., and Zhang Y.Q. Accelerating video decoding using gpu. volume 4, pages 772–775, 2003.

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

[35] Massie T. H. and Salisbury J. K. The phantom interface: A device for probing virtual objects. In *Proc. ASME Winter Annual Meeting, Symposium on Haptic Interfaces for a virtual environment and teleoperator systems*, 1994.

[36] Zhang H. *Simulating Tissue Dissection for Surgical Training.M.A.Sc. Thesis.* Simon Fraser University, 2004.

[37] Buck I., Fatahalian K., and Hanrahan P. Gpubench: Evaluating gpu performance for numerical and scientific applications. In *ACM Workshop on General Purpose Computing on Graphics Processors*, 2004.

[38] Rudomin I. and Castillo J. Realtime clothing: geometry and physics. In *Journal of Winter School of Computer Graphics*, pages 45–48, 2002.

[39] Amirbayat J. and Hearle J. W. S. The complex buckling of flexible sheet materials, part i: Theoretical approach. In *International Journal of Mechanical Science*, volume 28, pages 339–358, 1986.

[40] Amirbayat J. and Hearle J. W. S. The complex buckling of flexible sheet materials, part ii: Experimental study of three-fold buckling. In *International Journal of Mechanical Science*, volume 28, pages 359–370, 1986.

[41] Bolz J., Farmer I., Grinspun E., and Schroder P. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *ACM Transactions on Graphics*, volume 22, pages 917–924, 2003.

[42] Choi Y. J., Kim Y. J., and Kim M. H. Self-cd: Interactive self-collision detection for deformable body simulation using gpus. In *Asian Simulation Conference*, 2005.

[43] Kent J., Carlson W., and Parent R. Shape transformation for polyhedral objects. In *ACM SIGGRAPH*, pages 47–54, 1992.

[44] Kessenich J., Baldwin D., and Rost R. *OpenGL Shading Language v 1.10*. 2004.

[45] Krger J., Kipfer P., Kondratieva P., and Westermann R. A particle system for interactive visualization of 3d flows. In *Transactions on Visualization and Computer Graphics*, volume 11, 2005.

[46] Krger J. and Westermann R. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM Transactions on Graphics*, volume 22, pages 908–916, 2003.

[47] Moosegaard J. and Sorensen T.S. Gpu accelerated surgical simulators for complex morphology. In *Proceedings of Virtual Reality*, pages 147–153, 2005.

[48] Segerlind L. J. *Applied Finite Element Analysis*. Wiley, 2nd edition, 1976.

[49] Thompson C. J., Hahn S., and Oskin M. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, page 306317, 2002.

[50] Colgate J.E. and Schenkel G.G. Passivity of a class of sampled-data systems: Application to haptic interfaces. 1994.

[51] Fatahalian K., Sugerman J., and Hanrahan P. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Graphics Hardware*, 2004.

[52] Salisbury K., Brock D., Massie T., Swarup N., and Zilles C. Haptic rendering: Programming touch interaction with virtual objects. In *Proceedings of the ACM Symposium on Interactive 3D Graphics*, pages 203–210, 1995.

[53] Dent T. L. The impact of laparoscopic surgery on health care delivery. the learning curve: skills and privileges. volume 3, page 247249, 1993.

[54] Lian L.L. and Chen Y.H. Haptic surgical simulation: An application to virtual suture. In *Computer-Aided Design and Applications*, volume 3, pages 203–210, 2006.

[55] Anitescu M. and Hart G.D. A constraint-based time-stepping approach for rigid multibody dynamics with joints, contact and friction. volume 60, page 23352371, 2004.

[56] Bridges M. and Diamond D.L. The financial impact of teaching surgical residents in the operating room. *The American Journal of Surgery*, 177:28–32(5), January 1999.

[57] Harris M. Gpu physics. In *International Conference on Computer Graphics and Interactive Techniques*, 2007.

[58] Moore M. and Wilhelms J. Collision detection and response for computer animation. In *Computer Graphics*, volume 22, pages 55–66, 1988.

[59] Paisley A. M., Baldwin P.J., and Paterson-Brown S. Accuracy of medical staff assessment of trainees' operative performance". *Medical Teacher*, 27(7):634–638(5), November 2005.

[60] Pascale D. M., Pascale D. G., Prattichizzo D., and Barbagli F. A gpu-friendly method for haptic and graphic rendering of deformable objects. In *Proceedings of Eurohaptics*, pages 44–51, 2004.

[61] Pascale D. M., Sarcuni G., and Prattichizzo D. Real-time softfinger grasping of physically based quasi-rigid objects. In *Proceedings of World Haptics Conference*, pages 545–546, 2005.

[62] Pharr M. and Fernando R. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley Professional, 2005.

[63] Shinya M. Stabilizing explicit methods in spring-mass simulation. *cgi*, 00:528–531, 2004.

[64] Srinivasan M.A. and Basdogan C. Haptics in virtual environments: Taxonomy, research status, and challenges. In *Computers and Graphics*, volume 21, pages 393 - 404, 1997.

[65] Goodnight N., Woolley C., Lewin G., Luebke D., and Humphreys G. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 102–111, 2003.

[66] Govindaraju N., Redon S., Lin M., and Manocha D. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware*, volume 32, 2003.

[67] Ray N., Ulysse J., Cavin X., and Lvy B. Generation of radiosity texture atlas for realistic real-time rendering. In *Eurographics*, 2003.

[68] Madsen N.K., Rodrigue G.H., and Karush J.I. Matrix multiplication by diagonals on a vector/parallel processor. In *Information Processing Letters*, volume 5, pages 41–45, 1976.

[69] Kipfer P., Segal M., and Westermann R. Uberflow: A gpu-based particle engine. In *Graphics Hardware*, pages 115–122, 2004.

[70] Moore P. and Molloy D. A survey of computer-based deformable models. In *International Machine Vision and Image Processing Conference*, pages 55–66, 2007.

[71] Liu Q., Prakash E.C., and Srinivasan M.A. Interactive deformable geometry maps. In *Visual Computing*, pages 119–131, 2007.

[72] Courant R., Friedrichs K., and Lewy H. ber die partiellen dierenzengleichungen der mathematischen physik. volume 100, pages 32–74, 1928.

[73] Mark W. R., Glanville R. S., Akeley K., and Kilgard M. J. Cg: A system for programming graphics hardware in a c-like language. In *ACM Transactions on Graphics*, volume 22, pages 896–907, 2003.

[74] Adams R.J. and Hannaford B. A two-port framework for the design of unconditionally stable haptic interfaces. In *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1998.

[75] Sidhu R.S., Grober E.D., and Musselman L.J. et al. Assessing competency in surgery: where to begin? volume 135, page 620, 2004.

[76] Choi S. Towards realistic haptic rendering of surface textures. In *PhD. Thesis, Purdue University*, 2003.

[77] Ferguson S., Davison J., and Lewin J. Surface wrapping technology for industrial cae. World Wide Web electronic publication, 2008.

[78] Floater M. S. and Hormann K. Surface parameterization: a tutorial and survey. In *Advances in Multiresolution for Geometric Modelling*, pages 157–186. Springer-Verlag, 2005.

[79] Haker S., Angenent S., Tannenbaum A., Kikinis R., Sapiro G., and Halle M. Conformal surface parameterization for texture mapping. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):181–189, April 2000.

[80] Payandeh S., Dill J., and Cai Z.L. On interacting with physics-based models of graphical objects. In *Robotica*, volume 22, pages 223–230, 2004.

[81] Payandeh S., Lomax A. J., Dill J., MacKenzie C.L., and Cao C.L.G. On defining metrics for assessing laparoscopic surgical skills in a virtual training environment. In *Medicine Meets Virtual Reality*, 2002.

[82] Payandeh S. and Azouz N. Finite elements, mass-spring-damper systems and haptic rendering. In *Proceedings of IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pages 224–230, 2001.

[83] Sorensen T. S. and Mosegaard J. Haptic feedback for the gpu-based surgical simulator. In *Medicine meets virtual reality*, pages 523–528, 2006.

[84] Eduardo Tejada and Thomas Ertl. Large steps in gpu-based deformable bodies simulation. pages 703–715, 2005.

[85] Kotamraju V., Payandeh S., and Dill J. Towards a gpu-based simulation framework for deformable surface meshes. In *Canadian Conference of Electrical and Computer Engineering*, pages 1349–1352, 2007.

[86] Osler W. *The Principles and Practice of Medicine.* Bayliss, Toronto,Canada, 4 edition, 1999.

[87] W. S. K. Wong and Baciu G. Gpu-based intrinsic collision detection for deformable surfaces. In *Computer Animation and Virtual Worlds*, volume 16, page 153161, 2005.

[88] Tutte W.T. How to draw a graph. In *Proceedings of London Math Soc.*, volume 13, page 743768, 1963.

[89] Gu X., Gortler S.J., and Hoppe H. Geometry images. In *ACM Transactions on Graphics*, volume 21, pages 355–361, 2002.

[90] C. B. Zilles and J. K. Salisbury. A constraint-based god-object method for haptic display. pages 146–151, 1995.