Progressive Cutting with Minimal New Element Creation of Soft Tissue Models for Interactive Surgical Simulation

Andrew B. Mor

CMU-RI-TR-01-29

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

October 11, 2001

Thesis Committee

Takeo Kanade, Chair
Omar Ghattas
Branislav Jaramaz
Sarah F. Frisken

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics

for Maria, Euler, and Archimedes

Abstract

This thesis deals with the modification of finite element models used in surgical simulation.

Surgical simulation offers the promise of enhanced medical training and education. It can provide a more realistic learning environment than many of the methodologies employed today while reducing costs. It also increases the variability of pathologies presented to the student, and can be used for continuing medical education. Simulators can also gain a place in the medical practice, to rehearse difficult or uncommon procedures. While a good deal of work has been done on the underlying soft tissue simulation, cutting and interacting with the model has been relatively ignored.

In this thesis, we present a method for simulating cutting of soft tissue within a physically based surgical simulator. The technique works on subdividing tetrahedral meshes while impacting model and simulator efficiency as little as possible. Model stability is addresses so that the new, cut, model does not cause the simulation to become unstable. Also, within the framework the interactive simulator demonstrated, the user is able to palpate, grasp, and puncture the model.

Acknowledgements

Thanks to my advisor Takeo Kanade, for helping and encouraging me throughout the years working on this research in the Robotics Institute. Also, thanks to my thesis committee members, Omar Ghattas, Branislav Jaramaz and Sarah Frisken.

I would also like to thank the many members of the Institute who have helped me reach this milestone, and some of my classmates over the years: Zack Butler, Sundar Vedula, Patrick Rowe and Terry Fong.

Lastly, I would like to thank my family for helping to get me here. Most of all, thanks to Maria (and Euler and Archimedes) for putting up with me these many years.

Table of Contents

1 Introduction 11

2 Motivation and Background 15

2.1 Difficulties of an Interactive Surgical Simulator 15

2.2 Cutting as an Integral Part of Surgery 17

3 Previous Related Work 19

3.1 Model Modification 19

3.2 Deformable Modeling 21

3.2.1 Surface-Based Models 21

3.2.2 Volume-Based Models 22

3.3 Haptic Interface 25

4 Experimental Simulator Overview 27

5 Cutting 31

5.1 General Cutting Procedure. 31

5.2 Minimal New Element Creation 32

5.2.1 Element Subdivision 32

5.2.2 Generation of the Minimal Set. 33

5.2.3 Comparison of Minimal Sets to General Subdivision 35

5.2.4 Intersection Detection and Propagation. 35

5.2.5 Intersection Testing 36
5.2.6 Intersection Coordinates 37
5.3 Progressive Cutting 38
5.3.1 Progressive Cutting Between Elements 38
5.3.2 Progressive Cutting with Temporary Subdivisions 41
5.3.3 Different Possible Cases for Temporary Progressive Cuts . . . 42
5.3.4 Progressive Cutting: Cutting Tip Within Model 43
5.3.5 Progressive Cutting: Topology Change 44
5.4 Stable Cutting With Snapping 45
5.4.1 General Concept. 45
5.4.2 Geometry Test for Element Stability 46
5.4.3 Where Intersection Points Snap 47
5.4.4 Ordering of Possible Cases to Find Stable Subdivision. 49
5.4.5 New Vertex Types 51
5.4.6 Paired Vertex Above or Below the Plane 52
5.4.7 Different Possible Cases For Snapped Cuts. 56

6 Soft Tissue Modeling 61

6.1 Tensor Mass System 61
6.1.1 Element Properties 62
6.1.2 Nodal and Edge Properties. 63
6.2 Position Integration 64
6.2.1 Nodal Dynamics. 65
6.2.2 Euler Integration 66
6.2.3 Runge-Kutta Integration 67
6.2.4 Verlet Integration 68
6.2.5 Element Stability 70
6.2.6 Computational Efficiency 71

7 Object Interaction 73

7.1 Palpation of Model. 73
7.1.1 Implicit Sphere - Node Interaction 74

7.1.2 Implicit Sphere - Surface Interaction 76
7.1.3 Implicit Cylinder - Surface Interaction 83

7.2 Grasping of the Soft Tissue Model 85

7.2.1 Grasping a Single Vertex 86
7.2.2 Grasping of a Surface Triangle 87
7.2.3 Grasping a Point Within a Triangle 88

7.3 Needle Puncture Modeling 89

7.3.1 Needle Sharpness Model 90
7.3.2 Propagation of the Needle Path 91
7.3.3 Transverse Forces Generated by a Needle 92

8 Haptics 95

8.1 Haptic Feedback in a Surgical Simulator 95

8.2 Intermediate Representation 96

8.2.1 Setpoint Local Model 97
8.2.2 Constraint Plane Local Model 99
8.2.3 Line Constraint Local Model 100

9 Implementation Details 101

9.1 System Setup and Implementation 101

9.2 Linear Elastic Soft Tissue Modeling 103

9.2.1 Tissue Parameters 103
9.2.2 Object Construction within the Simulator 103

9.3 Interaction Routines 105

9.4 Haptics System 106

9.5 Graphics System 106

10 Examples and Performance 107

10.1 Changes in Update Rate 107

10.2 Distance of Cut Surface from User’s Path 109

10.3 Progressive Cutting with Temporary Subdivisions 113
10.4 Interaction with a Rectangular Model 113
10.5 Simulation of Liver Model 117

11 Conclusions 123

11.1 Contributions 123
11.2 Future Work 125

List of Figures

Figure 1 System Diagram. 29

Figure 2 General cutting process.. 32

Figure 3 The five cases of tetrahedron subdivision after a completed cut. . . . 33

Figure 4 Minimal element subdivision. 33

Figure 5 Minimal element subdivision, exploded view. 34

Figure 6 General tetrahedron subdivision. 35

Figure 7 Cutting edge intersection with a tetrahedron. 36

Figure 8 Progressive cutting between elements example. 39

Figure 9 New vertices not connected to unintersected element. 40

Figure 10 Progressive cutting with temporary intersections example. 41

Figure 11 Different types of intersections for progressive cutting. 42

Figure 12 Eleven different progressive cutting cases. 43

Figure 13 Temporary subdivision, with two intersections on one face.. 44

Figure 14 Temporary cut opening up between face intersections.. 44

Figure 15 Flow of stable subdivision routine. 46

Figure 16 Example of stable snapping. 47

Figure 17 Element edge length and vertex height. 47

Figure 18 Where edge and face interaction points snap to. 48

Figure 19 Face snapping to vacated edge. 49

Figure 20 Sorting of possible permutations. 50

Figure 21 Paired vertices in an unsnapped intersection. 52

Figure 22 Example of paired vertices in snapped intersections. 53

Figure 23 Example showing two different cases of paired vertices. 54

Figure 24 Non-trivial combinations of vertex distances above, below, or on the plane.. 55

Figure 25 Volume fraction above the cutting plane. 56

Figure 26 Snapped progressive cutting cases. 58

Figure 27 Tetrahedral element shape and nodal ordering. 63

Figure 28 Summing of stiffness matrices. 65

Figure 29 Tensor mass system deformation example. 67

Figure 30 Cross-section of implicit sphere interacting with the nodes of a model. 75

Figure 31 Example of hole generated by implicit sphere, in cross-section.. . . . 76

Figure 32 Cross-section of implicit sphere interacting with surface, showing the volume of intersection 77

Figure 33 Closest point on a triangle.. 78

Figure 34 Sphere intersected with surface triangle, and projection of the sphere onto the plane of the triangle.. 79

Figure 35 Different types of intersections between projected circle and triangle. 79

Figure 36 Area of lopped circle.. 80

Figure 37 Equivalent forces at nodes to force at centroid.. 82

Figure 38 Model deformed by an implicit sphere touching the surface. 83

Figure 39 Intersection of a finite edge with a finite, implicit cylinder. 84

Figure 40 Grasping of a node. 86

Figure 41 Grasping of a triangle. 87

Figure 42 Grasping the closest point on the model to the grasping tool. 88

Figure 43 Example of soft tissue model being grasped and pulled upon. 90

Figure 44 The beginning of puncture with a needle. 91

Figure 45 Checking the backward or forward motion of the needle.. 92

Figure 46 Force generation due to needle deflection. 93

Figure 47 Example of object deformation caused by needle deflection. 93

Figure 48 Basic haptic feedback loop. 96

Figure 49 Typical force levels with and without intermediate representation with slow update rates.. 97

Figure 50 Example of servoing to a point. 97

Figure 51 Extrapolated setpoint example. 98

Figure 52 Constraint plane for local modeling. 99

Figure 53 Line constraint for local modeling. 100

Figure 54 Block diagram of system flow. 102

Figure 55 Basic data structures for the soft tissue model. 104

Figure 56 Completed cuts for showing changes in update rates. 108

Figure 57 Cut surfaces generated by the same motion as in Figure 56. 110

Figure 58 Second example of completed cuts demonstrating different cutting methods. 110

Figure 59 Cut surfaces generated by the same motion as in Figure 58. 111

Figure 60 Liver model example (Exact, snapped, clearing cuts). 111

Figure 61 Cut surfaces of model shown in Figure 60. (Exact, snapped, clearing cut). 112

Figure 62 Progressive cutting, with temporary subdivisions, of a rectangular model. 114

Figure 63 Undeformed image of basic rectangular model 115

Figure 64 Palpating and grasping the rectangular model. 115

Figure 65 Partial cut of rectangular object. 116

Figure 66 Complete cut of rectangular object. 116

Figure 67 Undeformed model of the liver. 117

Figure 68 Model of the liver under the effect of gravity. 118

Figure 69 Liver model palpated by implicit sphere. 118

Figure 70 Palpating the liver model with an implicit cylinder. 119

Figure 71 Partial cut of the liver, front view. 120

Figure 72 Partial cut of the liver, bottom view. 120

Figure 73 Complete cut of the liver, frontal view. 121

Figure 74 Complete cut of the liver, bottom view. 121

List of Tables

Table 1 Enumeration of different cases for temporary intersections. 42

Table 2 Sorting of possible permutations. 51

Table 3 Enumeration of different cases for snapped intersections. 57

Table 4 Computational efficiency vs. numerical stability for integration. 72

Table 5 Changes in update rate and number of elements based on cutting method. 108

Table 6 Mean distance from ground truth. 112

Chapter 1

Introduction

Rising health care costs and reduced reimbursements over the past few years have spawned a quest to reduce the costs associated with medical care and education. The number of minimally invasive procedures performed has drastically increased, while surgeons are developing new methods for applying these procedures to regions of the body formerly unheard of. For example, while arthroscopic knee surgery has commonly been used for the past 10 years, only recently have minimally invasive total joint replacements become common. Recent advances in cardiac surgery have led to minimally invasive procedures for coronary artery bypass.

Minimally invasive procedures are, generally, cheaper to perform than open procedures. They also generate less scarring and risk to the patient and reduced healing time because the surgical field is not fully opened, but accessed through small ports, usually less than half an inch in length. Visualization of the surgical field is done with long, slender video cameras, and surgical tools are of similar shape, to reach the remote surgical field. These procedures, and the many other minimally invasive and new open surgical techniques, require extensive training to be performed reliably and safely.

In the past five years, surgical simulation has become an area of heavy research in the academic world. Surgical simulation, using computers and robotics, can supplant and improve traditional training techniques. Traditionally, surgical training has been performed on plastic models, cadavers, or on actual patients. While mechanical models are relatively cheap and can be used many times, the sensations they generate are unrealistic at best. They can also only demonstrate a limited range of anatomical sizes and pathological situations. Additionally, different models must be acquired for every type of procedure to be learned. Cadaveric training presents the most realistic anatomy possible, but tissue responses are affected by the preserving techniques and differences in temperature and fluid flow, and the number of pathologies presented is usually not very broad. Additionally, there is a limited supply of cadavers for medical training and the cost to provide extensive training with them would be high.

Training students by performing procedures on actual patients is the gold standard for training, but there are ethical considerations. Novice students may not have the experience to reliably and safely perform procedures; the potential for complications may cause an attending surgeon to take over. Also, similar to cadaveric training, the procedures related to particular pathologies is limited to what may come through the operating room at a particular time, and may not fully cover the breadth of techniques a student would need to learn. Lastly, when the difficult and interesting cases do come into surgery, the student may be only allowed to observe the procedure, and not actually participate.

Surgical simulation can improve on these limitations in many ways, although they can not completely substitute for practicing on live patients. Depending on the fidelity of the model, the simulator can be as realistic as the plastic model up to close to the realism of living tissue. The simulator, unlike plastic models or cadavers, can be programmed to show an unlimited variety of pathologies, both in type and placement. They can often be used for surgical procedures on many different parts of the body, with only changes to the modeled part of the body in software required. Patient models can be adjusted to account for the sex and size of the simulated patient to be operated upon. The student can also practice on a particular pathology as many times as she feels is necessary. Students can also be easily scored on the simulators, with data gathered on time of the procedure, accuracy of the results, and damage to surrounding tissue. Lastly, training using a surgical simulator can reduce costs. Since there is only the upfront cost of the simulator, ongoing costs are reduced while realism and amount of practice are increased.

Surgical simulators also have a place outside of medical schools. Surgeons often attend medical conferences to learn new techniques and to continue their medical education. Simulators can be used at these conferences in a much more versatile and realistic manner than plastic models for practicing surgeons. Simulators can also be used within the surgical practice for rehearsing a procedure before operating on the patient. Patient specific models can be acquired with current scanners and object segmentation techniques. Tissue parameters can be assigned, and then the surgeon can use that model to verify that a procedure is safe and effective.

Much of the recent research in surgical simulation has focused on increasing the realism of modeling soft tissue. While this is very important to the overall applicability of surgical simulators, other aspects of surgery must also be realized. This thesis addresses another very important part of surgical procedures, cutting of soft tissue within the framework of a physically based, interactive simulator. Since cutting is such a common and important task within surgery, it needs to be addressed as an issue as important to the realism of the simulation as the soft tissue model. Cutting of the models must be done accurately, following the path traced out by the user as close as possible, while maintaining the stability and efficiency of the overall simulation. A minimal set of new elements, to reduce subsequent computational load, should be generated for each cut element in the original model. The cutting should also occur progressively, as the user moves through individual elements and the overall model, so that the model is updated iteratively as the cut occurs, not after the cut is completely finished. The cutting should also affect the tissue model in the expected fashion, not altering tissue parameters or model size unnecessarily.

The simulator described in this thesis is built upon a linear elastic finite element based tissue model using tetrahedral elements. In addition to cutting, we also implemented other interaction techniques, palpation, grasping, and puncture, to demonstrate the requirements of a general simulator.

In the following chapters, we first describe the motivation and background for this work and prior work in the area. A brief description of the surgical simulator is introduced in Chapter 4. Chapter 5 describes the main thrust of this work on cutting. The soft tissue modeling is introduced in Chapter 6, and Chapter 7 describes other interaction techniques in the simulator. Next, the haptics routines and implementation details of the overall simulator are described. Lastly, examples and results are shown in Chapter 10, and these results are discussed in the final chapter, Chapter 11.

Chapter 2

Motivation and Background

Performing surgical procedures requires a great deal of skill, involving proficiency in a large number of different techniques. Since surgery most often entails modifying in some manner the macroscopic tissue structure of a patient, one of the most common aspects of surgery is cutting, to either reach or repair internal structures. Because of the importance cutting of tissue has in surgery, this thesis largely addresses the issue of cutting accuracy and efficiency in surgical simulators. Why surgical training is necessary and how simulators can improve current methods was described in the previous chapter. In this chapter, we will go into a little detail about the difficulties in the development of an interactive surgical simulator and the importance of cutting in surgery.

2.1 Difficulties of an Interactive Surgical Simulator

There are many challenges in the development of a realistic surgical simulation system, including the modeling of the tissue, interaction between tools and the model, and the user interface. A realistic model of the soft tissue of the body is required, which behaves in a manner that surgeons would expect, given their background and experience. A fast, physically based algorithm is required to generate the deformations of the modeled tissue. Interaction between simulated tools and the model is also required for a training system, as is a methodology for modifying the topology of the tissue in an efficient manner. Lastly, the best way to train a novice user is to have her practice the actual motions, which requires the use of a haptic interface. Update rate requirements for force feedback devices are much higher than for graphical interfaces: around 500-1000Hz instead of 15-30Hz, respectively.

The problem of how to model soft tissue for a real-time simulation is a difficult one. Different methods have been used in the past, each with its own strengths and weaknesses. In general, they can be broken down into either surface-based models or volume-based models. Surface models that have been implemented range from continuous, snake-based models, to discrete mass spring models based on triangle meshes. The problem with all surface-based models, when used to simulate tissue with interior structure, is that they do not explicitly model the interior. Therefore, complex interactions between the surface of the tissue and the interior structure can not be modeled. Volume-based models can simulate interior structure because they encode the entire object, thereby modeling the interactions between the interior structure and the exterior of the object. Discrete mass spring models are very popular, and can be implemented in an efficient manner, but they do not model the tissue in a physically based manner. A more physically based method is to use finite element techniques. While standard finite elements can generate very accurate results, they can be quite slow. There are techniques to speed up finite element models, but most of those impose the requirement that the topology of the model does not change. Since cutting is such an important part of surgery and it changes to topology of the model, none of the standard precomputation techniques are applicable.

Modifications to the patient model caused by cutting are very important in a surgical training system. While the ability to learn how to move and navigate around a new anatomical region is very important, a system which does not include the capability to modify the simulated tissue has limited utility. Modifying the soft tissue can be viewed as occurring when the local shear stress passes above some threshold for the simulated material, and all modification techniques, including cutting, puncture, and tearing, have the same underlying basis in physics. For simulation however, it is advantageous to model the different activities in different ways, both for simplicity and efficiency. Cutting methods also should be accurate, volume preserving, and update as the user moves the cutting tool through the modeled tissue. Progressively cutting through the model requires a very efficient underlying technique to update the model as the user moves a scalpel. Finding stable subdivisions of individual elements to ensure the stability of the overall model is also necessary.

Training with a simulator requires both a graphical display and a haptic display. A purely graphical display can tolerate low frame rates, lag, and dropouts. A haptic display, on the other hand, would not work with any of those deficiencies. While there are techniques for interfacing low rate simulations with high rate devices that will provide a smooth sensation to the user, there is still a minimum update rate that needs to be maintained to provide realistic sensations to the user. The haptic interface itself must run at a minimum of 500Hz, while a simulation running at 100Hz, in most cases, is fast enough to interface well with the haptic routines. Real-time interaction between the user and the simulation also requires modeling tools that the user can hold and how they interact with the soft tissue. Tools for modification, as described above, and palpation are required. All interaction techniques require fast collision detection to detect object-tool intersections and to generate the forces that arise.

2.2 Cutting as an Integral Part of Surgery

The main actions that surgeons utilize are cutting, carving, and sewing. While there definitely is a great deal of knowledge and skill behind their hands, surgeons mainly perform these three tasks. This thesis demonstrates a technique that accurately models one of those three techniques, cutting. Cutting is very common in both open surgery and minimally invasive surgery. It is used in all parts of the body, from the brain, to the gall bladder, to orthopedic procedures. It is important to know where and how to cut, because the action is often non-reversible. It is also important to be able to visualize the results of the cutting actions.

Some simulators process the cutting action after the user has completed tracing the cut through the object. This does not provide the immediate feedback that is needed to see how a cut is progressing. It also does not allow the user to change the path that is being traced to account for any problems or mistakes encountered while cutting through the tissue. Because of this, it is better to update the model as the user cuts through the object, providing almost instantaneous updates as to the progress of the cut.

This thesis demonstrates two types of progressive cutting: cuts that occur as the user completes his motion through individual elements of the model; and cuts that are modeled as the user moves within individual elements. The first method generates a small amount of lag, on the order of the typical edge length within the model. The second method updates at the rate of the simulation, but can suffer from difficulties with model stability and determining when an edge is definitively cut.

There are multiple ways to modify the underlying model of the simulated object, each with its own drawbacks. The main requirements for accurate cutting are to faithfully follow the path traced out by the user and to impact the overall simulation as little as possible. We want to follow the path of the user, and not to modify the object in a way that the user didn’t intend. The path should accurately reflect the path traced out, and not modify the mass or volume of the model. The generated cut should also not impact the computational load of the simulator severely, either through the computation required to generate the cut, or through a large increase in the model size, i.e. the number of elements, after the cut is completed.

One very simple technique is to remove all elements that are contacted by the cutting tool. Another technique is to find the element boundaries closest to the surface traced by the cutting tool, and then split elements apart along that boundary. Lastly, we can track the actual intersection points between the cutting tool and the individual elements, and generate the cut surface between these intersection points.

The first two methods have clear drawbacks. The first technique changes the overall topology, not to mention mass and volume, of the object. If the average element size is not quite small, a large, irregularly shaped channel will be carved through the object. The problem with the second method is that there are not always element boundaries that closely follow the path of the cutting tool. Therefore, a rather jagged and irregularly shaped cutting path is created, which may zig zag back and forth across the path that the user traces out. In this way, the generated cut surface can be quite different from the path traced by the user.

For example, if we have a model where the average edge length is 30mm, removing elements completely, or splitting along boundaries, would place the vertices on the cut surface a large distance away from the path traced out by the user. This deflection would be roughly half the average edge length, or 15mm. This distance in a surgical simulation would be quite noticeable. If we wanted to bound the error at a smaller distance we would have to decrease the element size. If the error is bounded to 3mm, then we would require an average edge length of 6mm, which is 5 times smaller than the initial mesh and would increase the number of elements by approximately 125 times. Even with this reduction in error, the cut surface will still not be smooth and will zig zag back and forth across the user’s path. Meanwhile, the computational load will have to increase by more than two orders of magnitude to achieve this increase in accuracy. Conversely, if we follow the path that the user traces out, we will be able to achieve the accuracy requirement while only increasing the number of elements locally around the cut surface, a much smaller subset of the initial mesh.

The last method, using actual intersection points, does not have any of the difficulties mentioned with the first two techniques. The cut surface exactly follows the path of the user, as it is traced across the original elements of the model. It does not remove any portion of the model, and therefore maintains object volume. The main drawback of this method, however, is that it can generate a large number of new elements to model the intersected elements, which can be prohibitively expensive computationally to model. This is the problem addressed by the minimal new element creation cutting technique described in this thesis.

Chapter 3

Previous Related Work

A large amount of work has been done in areas related to this research. In the area of soft tissue modeling, research has been done on both surface-based and volume-based modeling, with both physically and non-physically based representations. Haptic methods for various applications have started to receive attention in the research community, and work related to this thesis is also described. Work on modifying soft-tissue models has not been as prevalent as work on the underlying soft tissue model or on haptics. The work on model modification that is related to this thesis is described in the following section.

3.1 Model Modification

Modification of object topology can take many different forms; cutting, tearing, and puncture make up a large subset. Puncture simulation, where penetration of the object occurs, is probably the most tractable form of interaction. Singh, et al. [45] and Popa and Singh [40] demonstrate a lumbar puncture simulator using impedance control to model the insertion through the layers of tissue of the back. Reinig [42] describes a volume-based puncture simulator where the resistance to penetration is based on actual tissue parameters. Reinig uses segmented visible human data to determine the type of tissue being punctured, and integrates the frictional resistance of the shaft of the needle along its path through the back.

Tearing can also be easily detected by looking at the internal forces being generated within the model. Miyazaki, et al. [29] support tearing by looking at the elongation of the springs in their mass spring system. If the length of the spring exceeds a threshold, then one end of the spring is separated from its neighbor. Cotin, et al., in their volumetric tensor mass system [13], support both cutting and tearing. Tearing occurs if one of three geometric measures of the tetrahedron exceeds a threshold. Cutting occurs when the model of a bipolar cautery instrument touches an element. Any element that is touched is removed from the model.

Cutting of surface-based models was demonstrated by Song and Reddy [47], where, in 2 dimensions, they moved a finite element mesh around with a cutting tool. The cutting force exerted by the user is introduced to the object as a nodal force. Once it exceeds the shear strength of the material, cutting occurs. Tanaka, et al. [51] use boolean operations on polygonal objects. The cutting tool and object are represented as polygonal objects, and the intersection of the two causes the polygons of the modeled object to be cut. Tanaka uses a haptic interface, and the cutting force displayed to the user consists only of a viscous force, based on velocity. Since cutting is modeled as a purely geometric activity, there is no internal model of a minimum cutting force.

O'Brien and Hodgins [35] demonstrated a method for propagating fracture through a brittle volumetric finite element model. They detect when the total forces, formed into a tensor, acting on individual nodes within the model would initiate fracture. Once a node has fractured, they determine the direction of travel of that fracture, and subdivide the element based on that direction. They also insure that the fracturing routine will not create ill-conditioned tetrahedra by snapping intersection points to the closest nodes. This method generates a small number of different subdivisions, due to the fact that all fractures are initiated at nodal locations. Fracture propagation is not explicitly modeled, but is inherent in the model due to stress risers caused by fracturing at previous time steps.

Cutting through volumetric objects is more difficult due to the more complicated connectivity between nodes or vertices, and the greater number of cases due to the arbitrary locations of intersections between the model and the cutting tool. Mazura and Siefert [27] create a cutting surface by specifying the beginning and end points of the cutting edge, and then interpolating between them. This creates a set of triangles which are intersected with the tetrahedral mesh. Once an element is detected to have been cut, it is split according to the number of intersected edges in the element. They were able to process a model with 15,000 elements in about 6 minutes. Bielser, et al. [7] cut through a mass spring object by tracking the tip and the direction of the cutting edge through the object. A small cutting plane is generated at every time stop between the previous edge position and the current edge position. Cutting occurs whenever an element has an edge or face that has been intersected, but that no longer has the cutting edge passing through it. In this manner, only one cut occurs in each tetrahedron, and the cutting action occurs the equivalent of one tetrahedron length behind the cutting edge position, introducing some apparent latency. The cutting procedure always subdivides an element into 17 smaller elements using midpoint subdivision. If one of the edges or faces has been cut, then that position is substituted for the midpoint. Split edges are replaced with two edges, with two vertices at the new location, while unsplit edges get one vertex. Also, all possible intersections are just mirrors and rotations of 5 basic cases.

Bielser and Gross [6] extended the work of [7] by reducing the number of new elements generated for each cut element. Instead of inserting new vertices on uncut edges and faces, they only insert vertices at intersection locations, plus new vertices on faces split in two. This reduces the number of new elements created, but does not minimize it. Ganovelli, et al. [18] demonstrate a similar system, where the true minimal sets of new elements are created when an element is cut, based on which of the five basic subcases that element corresponds to. They determine the subdivision function to call by generating a code based on which edges are intersected. This is a fast and efficient method of determining how to subdivide each cut element. Ganovelli also implements tearing, where an spring elongation factor is computed. Once a certain number of edges have factors past a threshold, then a tearing path is propagated out from the most stretched edge. This propagation insures that the split will generate only legal subdivisions using the same routines as the cutting process. Mor and Kanade [33] also demonstrate a system that generates a minimal set of new elements during cutting, with progressive updates of the cut elements while the cut is occurring.

3.2 Deformable Modeling

The area of deformable modeling can be broken down into two main types of models: surface-based and volume-based. Surface-based models only represent the exterior of an object, and therefore should be limited to areas where complex interior structure is not present; for example, the gallbladder. Volume-based models can simulate the interior structure of an object, and are therefore more powerful, although computationally more expensive. Additionally, novel methods for integrating the motion of the model are also presented.

3.2.1 Surface-Based Models

Physically-based deformable object simulation began with the models based on elasticity theory developed by Terzopolous [53] to fit models to images. The models combined an internal energy term with external forces to generate a smooth model that was attracted to edges in an image. These models were continuous and not suitable for real-time interaction. Terzopolous and Waters [54] also developed a discrete mass-spring system for facial modeling. Their system consisted of a three-layer mesh with anisotropic behavior to model muscle below a fatty tissue layer below the epidermis. The model had 6500 springs and was animated at interactive, graphical, update rates.

Swarup [49] developed a mass spring system that would run at rates suitable for a force feedback device. His system was based on a two layer mesh, a top layer that the user could interact with and would deform, and a bottom stationary layer that didn't move. The nodes in the top layer were interconnected with springs, connected to the bottom layer with springs, and also connected to a "home" position with a spring. Only elements within a certain distance of the user's position would have their state updated. One serious problem with this system was the need for the bottom layer, whose only purpose was to ground the surface layer and to utilize existing finite element theory. The model also was not partitioned into an octree or with bounding spheres, so computation time increased quickly with the number of nodes. Tarr and Salisbury [52] addressed these problems, by utilizing a dynamically remeshed surface mesh to present to the user. The mesh was also partitioned within an octree, so only local nodes were intersected with the sphere representing the users position. They removed the bottom layer of the model by assuming the density of nodes on the mesh was suitably high and homogeneous.

Keeve, et al. [24] presents a similar surface model for generating facial tissue deformations after craniofacial bone surgery. He demonstrates both a multi-layer mass-spring model with biphasic springs, and a single layer non-linear finite element model. Both methods produced results that predominately agreed with a test case, although the two demonstrate the trade-off between precision and computation time.

Meseure [28] recently presented a mass spring model that depends on a virtual rigid component to generate bulk translations and rotations. A surface mesh is connected to the virtual component, and motion of the virtual object represents the undeformed desired position of the model. The surface mesh then deforms due to local interactions and collisions. If no forces are present, then the two components line up. If there are external forces, the surface mesh is perturbed away from the rigid component.

Moutsopoulos and Gilles [34] utilized a coarse-fine finite element model to simulate a gallbladder in a laporascopic surgery simulation. The global model was coarse enough (few enough nodes) to run at real-time rates, while the local model, around where the user was interacting, would subdivide the mesh to generate a finer, smoother response to the user. They also utilized a B-spline surface to interpolate the nodes of the finite element model to represent the "skin" of the object. Their system was able to interact at real-time rates (25 Hz) utilizing a coarse model with 60 nodes.

3.2.2 Volume-Based Models

Mass Spring Models

Two groups that have implemented three-dimensional, volumetric mass spring models are Reznik and Laugier [43] and Miyazaki, et al. [29]. Reznik and Laugier implemented a basic homogenous mass-spring model, using Euler's method for the numerical integration. The volume is sampled into a cubical lattice, with each node connected to its 26 nearest neighbors. Spring stiffness is determined by the Young’s modulus for the material, and nodal mass is similarly determined by the density of the material. Nodes are interconnected, but not attached to a home position. The authors claim real-time simulation, although they do not state the number of nodes in the simulation or what update rates they achieve. Miyazaki, et al., use a very similar system, although they also simulate model modification through tearing or cutting. If edges are cut, or the edges become too elongated, the spring connecting two nodes is removed. They also present a method to prevent divergence of spring oscillations when large forces are applied. Bielser, et al. [7] also recently demonstrated a system for soft tissue simulation. They utilized a tetrahedral mass spring system to run the simulation, with rigorous treatment of tracing a cutting surface through the simulated object. They demonstrated update rates of a few Hertz on models ranging in size from 48 to 576 elements before cutting, to 354 to 2446 elements after cutting.

Radetzky, et al. [41] show a mass spring model where the spring constants are generated by a neural network so that prespecified deformations match those acquired from experiments on real tissue. The parameters can also be adjusted by a neuro-fuzzy system for user feedback on whether tissue properties feel correct.

Kühnapfel, et al. [25] have developed a system based on their simulation software KISMET, to simulate endoscopic surgery. The soft tissue system is a volumetric mass spring model. The novel feature of their system is a spring stiffness value based on a third degree polynomial. Their research into living tissue showed that the non-linear shape of the stress-strain curve can be well approximated by this polynomial.

Suzuki, et al. [48] developed a system to model deformable tissue by filling a volume with rigid spheres. Forces are generated and deformation occurs when the user, modeled as a larger sphere, intersects the outer layer of spheres. Those spheres are pushed back, and the spheres behind them are pushed back in turn; forces are generated by each sphere wanting to return to its home position and contact with neighboring spheres. Modification of the model was not demonstrated.

Physically Based Models

Bro-Nielsen and Cotin [8] developed a system to utilize classical, three dimensional solid finite element models that would run at real-time rates. Real-time performance was achieved by the use of condensation, precalculation of the inversion and exploitation of the sparse structure of the force vector. Their technique is based on two assumptions: that the topology of the model could not change and that the only deformations seen by the user would occur at the surface nodes. First, the sparse global stiffness matrix is condensed so as to only calculate the displacement and forces at the surface nodes. In condensation, the effect of the interior nodes is taken into account. The idea is that the actual value of the deformation of the interior nodes is not important, only the deformations of the surface nodes matter. They then preinverted the condensed stiffness matrix. Lastly, in most situations in interactive simulations, the user is only touching a few of the surface nodes at a time. Due to this, the force vector is made up of mostly zeros. Bro-Nielsen and Cotin precompute deformation vectors based on a unit force at each node. At run-time, deformations are linear sums of the precomputed vectors based on the force vectors applied by the user. They achieved update rates of 20Hz on models with 250 surface nodes without utilizing the sparse nature of the force vector, and 20Hz on models with 700 nodes utilizing that sparseness. Cotin and Delingette [12] more recently demonstrated update rates of 100Hz on models with 1400 nodes.

Condensation is a popular technique for modeling soft tissue that will not be modified. Kühnapfel, et al. [25] have used this technique, in addition to their mass spring model, to generate faster updates. Berley, et al. [5] show a simulator for suturing skin that simulates models with up to 13,300 nodes using a banded matrix technique that uses condensation as a preprocessing technique. The user can interact, with a force feedback device, with up to 285 nodes at a time. Frank, et al. [17] demonstrates results for a banded system, and predicts the computational power required before condensed finite element system can run at 500Hz and directly generate forces for a haptic device. The results were shown for different techniques of solving the state equations of the model, with only iterative solutions of small (125 nodes) elements capable of update rates of 500Hz.

More recently, Cotin, et al. [13] developed a new representation they named tensor mass based on linearly elastic continuum mechanics to model soft tissue. It is based on finite element theory, but the models are solved in a dynamic fashion over time. Stiffness matrices are calculated in the same manner as for typical finite element models, but instead of forming a global stiffness matrix and solving a global solution at each time step, the stiffness matrix is stored locally, at each nodal point and for each edge. They achieved update rates of 40Hz with a mesh made up of 760 vertices and approximately 4000 edges, which was similar to the rate obtained for a mass-spring system they implemented. Also, as an update to the results in [8], they demonstrated results utilizing precomputation of elementary deformations of a quasi-static mesh of 500Hz on a mesh with almost 8000 tetrahedra.

Due to the limitations of the linear elastic models, Picinbono, et al. [37] have shown the application of a non-linear elastic model to the tensor mass system described in [13]. The non-linear elastic component overcomes the elongation limit of 10% of mesh size to be reasonably accurate. They also add an incompressibility constraint to limit growth of individual elements. For one example, this reduced the growth in the volume of the model under a large scale deformation from 63% to 1%. The main drawback of using a non-linear model is the computation time required. Going from a fully linear model to a fully non-linear model, the update rate for a liver model they show dropped from 45Hz to 8Hz, an 82% drop in update rate. For interactive simulations, this can be a very expensive improvement.

Debunne, et al. [15] demonstrate a system using a multi-resolution model to guarantee a minimum frame rate while still allowing a fine resolution model around deforming areas and where the user is contacting the model. They built a model with multiple levels of detail, and then switch between the different models based on computational load and a quality criterion. This multi-resolution model, while depending on a good deal of pre-computation, allows their simulator to run at real-time rates sufficient for haptic interfaces, with the multi-resolution model running 5 to 20 times that a single resolution model at the finest level of detail that they use.

Most of the methods described above that generate the state of the model iteratively use either Euler or Runge-Kutta methods. The main constraint with respect to these solvers is that the system of equations for the position of the model is viewed as a stiff system of equations [2]. Bielser and Gross [6], though, use a semi-implicit method to generate results more stably than using a typical explicit solver. The method uses an explicit step to estimate the current position of the object, then that estimated current position to implicitly determine the velocity. The final step is another implicit step to determine the final current position of the object. They do not give comparison numbers, though, for the stability of this model compared to a model updated with an explicit method. They did achieve 30Hz with a mass spring model consisting of 1381 tetrahedra using an SGI Onyx2 with 8 R10000 200MHz processors.

3.3 Haptic Interface

Haptic refers to anything having to do with the sense of touch. A haptic interface is a device that can be used to feel “objects” that are generated by a computer or some other modality, that are not actually present locally. Motors are used to generate forces that can act on the user based on his location within a virtual world. Haptic devices can take many forms. Custom interfaces have been built by many groups to satisfy unique requirements. Singh, et al. [45] implemented a custom device to support lumbar puncture simulation. Berkelman, et al. [4] demonstrate a device that uses a magnetically levitated handle that the user grasps. This removes all friction from the device, and can increase its responsiveness. A commercially available device is the PHANToM [26], which is a 6DOF input mechanical device, with either 3 or 6 active degrees of freedom.

Haptics have been used to feel virtual objects in many types of situations. In [32] and [19], a system for interacting with static voxel based data sets is described. Using segmented voxel data sets from MRI scans of a healthy knee, Gibson and Mor demonstrated the ability to feel medical data sets using a PHANToM haptic interface. Both polygon and volume rendering visualization were implemented. Haptic interaction with quasi-static finite element models of the liver is demonstrated by Cotin and Delingette in [12]. And O'Toole, et al. describe in [36] a training and evaluation system for end-to-end anastomosis using a flexible, spline-based model of a blood vessel and a force feedback system.

Complex simulation systems often run at rates lower than that required for haptic feedback. Adachi, et al. [1] proposed the use of an intermediate representation to model a rigid object between updates from the simulation. Berkelman, et al. [4] use a virtual coupling between the simulated object that the user is holding and the handle position of the haptic device; and can therefore guarantee the stability of the haptic device. When the update rate of the simulation is slow, setpoints for the haptic device are interpolated to smooth the path that the user traces. Interaction occurs between surfaces of completely rigid objects.

Picinbono, et al. [38] show a method of extrapolating forces by projecting the current position of the device onto the line between the positions at the last two forces updates. The force is then the extrapolated based on the distance from the current projected point to the previous updated position, compared to the distance between the two positions at the previous updates. This gave better results compared to the two other methods that they implemented.

d'Aulignac et al. [14] demonstrated a mass spring model that uses a local model of the system to generate updates more quickly than the rate of their underlying simulation. The local model is based on a constraint surface that is continually updated based on the previous history of the local model. The force generation portion of their simulator is based on a penalty based method utilizing the volume of intersection between the simulated object and a model of the tool that is being used. This is similar to most implementations, although the scalar distance of penetration is normally used to generate the magnitude of the force to display.

Chapter 4

Experimental Simulator Overview

Surgical simulators are designed to demonstrate and teach the motion and result of surgical actions. In this manner, they must be able to model soft tissue and the actions that affect it. In this thesis, we describe techniques and methods that we developed to cut through soft tissue within the framework of an interactive simulator. In addition to those cutting methodologies, we developed methods of simulating interaction between surgical tools, modeled as simple shapes, and soft tissue. We tied these interaction techniques into an experimental surgical simulator, using a linear elastic deformable model, and a haptic interface to provide physical feedback to the user.

There are 3 main components of our surgical simulator: soft tissue simulation, tissue modification and manipulation, and the user interface. In this simulator, soft tissue is modeled with a linear elastic finite element model. Tetrahedra are used as the basic element shape, to simplify modifications of individual elements when compared to element shapes with more nodes. The soft tissue model is implemented in such a fashion that models are easily and quickly updated and modified, using interconnected lists of pointers to basic data types: vertices, nodes, and elements.

Tissue modification and manipulation are implemented as routines that are called by the soft tissue simulation. They access the data structures of the model, modify the structure of the model, and generate external forces to modify the current state, both position and velocity, of the model. Intersections between the model and the currently wielded tool are propagated based on the local neighborhood of the currently intersected element, speeding up the determination of which features need to be accurately tested. When interacting with the model, the routines also generate forces to display back to the user through the haptic interface. These routines can push and pull on the model, and cut or puncture the model. Currently implemented interaction tools are: a form of cauterizing knife, that can cut in any direction; an implicit sphere model for palpating the object, which can be viewed as similar to the shape of the fingertip; an implicit cylinder model, also for palpating the soft tissue; a simple grasper tool, which can grab the model and move it; and a needle model, for simulating puncture.

Lastly, all simulations require some method for the user to see and act upon the model. A simple graphical interface is used, along with a PHANToM haptic device. The haptic interface allows the user to feel the model, and act upon it in a more realistic fashion than if there were no forces displayed back to the user. The haptic routines are implemented so that they can receive intermittent, slow updates from a simulation, and generate a smooth, stable flow of forces to display to the user.

Figure 1 shows the basic flow of control in our surgical simulator. After startup, the soft tissue model is initialized and communications between the soft tissue simulator and the haptics server are established. Then, the soft tissue modeling and object modification loop runs as fast as possible, up to 1000Hz, and queries the haptic display routine for the user's current position. If the user is currently interacting with the modeled tissue, then the type of interaction is determined, and if the tissue is being cut, the intersection occurs. After the topology of the model is modified, if necessary, the forces on the nodes are calculated and nodal positions updated, and the current model for displaying forces to the user is communicated to the haptics server. The scene is also, independently, graphically rendered at 30 frames per second. The separate haptic display routine receives updates from the modeling routines after every soft tissue update cycle, and interpolates between time steps to provide a smooth, stable interaction for the user, nominally updating at 1000Hz. These different routines and methods are described in the following chapters.

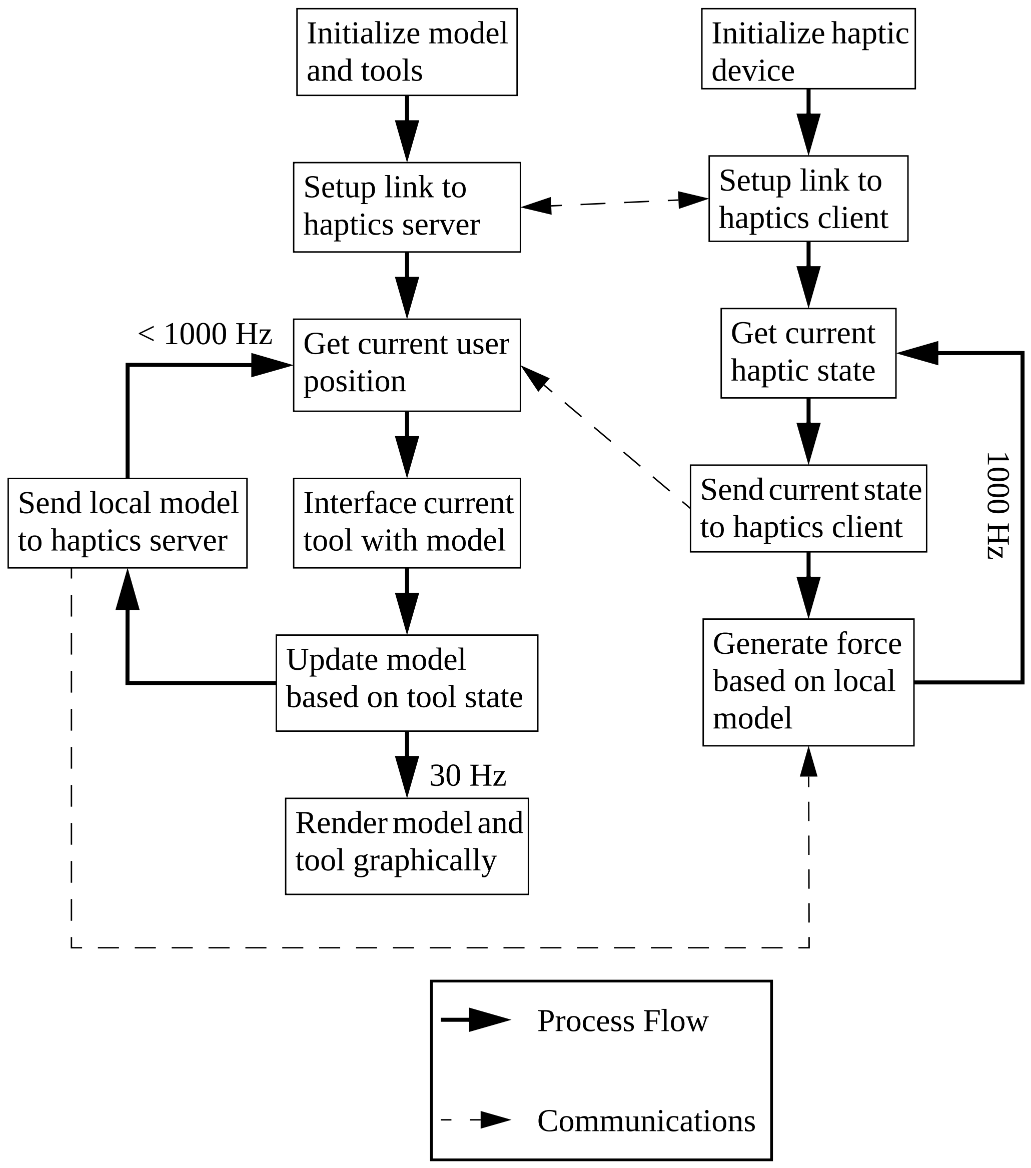
Initialize model and tools
Setup link to haptics server
< 1000 Hz
Get current user position
Send local model to haptics server
Interface current tool with model
Update model based on tool state
30 Hz
Render model and tool graphically
Initialize haptic device
Setup link to haptics client
Get current haptic state
Send current state to haptics client
1000 Hz
Generate force based on local model
Process Flow
Communications

FIGURE 1. System Diagram.

Chapter 5

Cutting

Cutting of soft tissue models generates new model topologies within the surgical simulator. The new topology of these models has to reflect competing goals. From the user's viewpoint, the cut should exactly follow the path that she traces out. From the simulator's viewpoint, the cut should impact its computational throughput as little as possible. These two competing goals, accuracy of the cut and a minimal increase of computational load, lead to the two main thrusts of this work on cutting: accurate and stable progressive cutting that follows the user's path, and minimal new element creation for modified elements.

After describing the general cutting process, we explain the motivation and method for generating minimal sets of new elements when cuts occur. Next, progressive cutting is described, both within elements and between elements. Lastly, to combat possible model instability, snapping of intersection points to maintain model stability is described.

5.1 General Cutting Procedure

The general procedure for cutting through elements with our tetrahedral based surgical simulator consists of the following steps. First, the initial intersection between the cutting tool and the model is detected. To do this, we test surface triangles and edges against the motion of the cutting tool to determine if the cutting tool moved across any of the boundaries of the surface, creating either face intersections, caused by the motion of the tip of the cutting tool, or edge intersections, caused by the motion of the cutting edge itself. Once an intersection is detected, we encode where on that intersected element the initial intersection occurred. We then test all the other faces and edges of that element against the motion of the cutting tool. For all element faces and edges that are intersected, we also propagate the intersection to the original element's neighbors, thereby quickly moving the cut surface through the model. Lastly, for each element that has been intersected, we subdivide the element once the cut has completed within its interior. This basic process is shown in Figure 2.

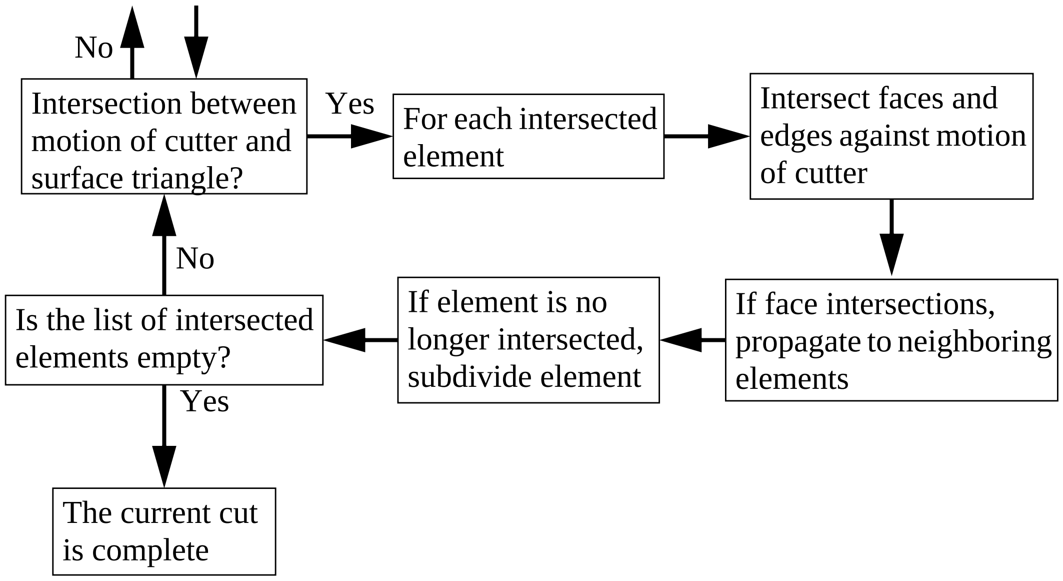
No
Intersection between motion of cutter and surface triangle?
Yes
For each intersected element
Intersect faces and edges against motion of cutter
No
Is the list of intersected elements empty?
If element is no longer intersected, subdivide element
If face intersections, propagate to neighboring elements
Yes
The current cut is complete

FIGURE 2. General cutting process.

5.2 Minimal New Element Creation

When an individual element is intersected by a cutting tool, there are three possibilities for splitting that element: removing the individual element completely; finding an element boundary and splitting along that; and generating the cut surface using the exact intersection points between the path of the cutting tool and the model. The first method does not preserve the volume and mass of the model, while the second method can generate cut surfaces that are very irregular and do not appear to actually follow the path of the cutting tool. We have implemented the last method, where we take the exact intersection points between the cutting tool and the model and generate the cutting surface between those points. First, we will describe the different possible topological types of intersections, then how we generated the minimal sets to replace the cut elements. Lastly, we describe how intersections are detected, stored, and propagated through the model, an important part of any cutting technique.

5.2.1 Element Subdivision

Tetrahedral elements cut by planar, or near-planar, surfaces will fall into one of five different topological cases, based on the number of cut edges and intersected faces.

There are two different cases where the tetrahedron is completely cut through into two pieces, and three cases where the element is cut, but not completely through. Figure 3 shows the different cases. The first example is when 3 edges are cut and a tip of the tetrahedron is separated from the rest of the element. The second example shows 4 edges cut, and the element evenly split into two. The first of the partially cut elements, when there are 2 face intersections, shows 1 edge intersection. The last two examples demonstrate, once again, 2 face intersections, and respectively, 2 and 3 edge intersections.

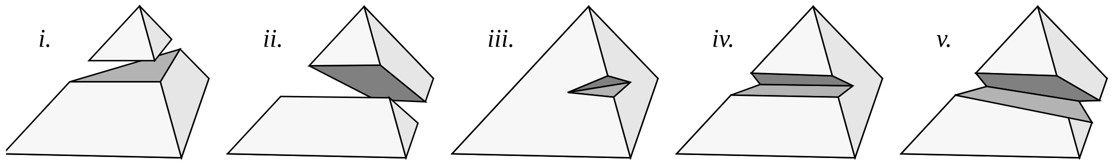

FIGURE 3. The five cases of tetrahedron subdivision after a completed cut.

5.2.2 Generation of the Minimal Set

To reduce the amount of computation required, we generate a minimal set of new elements to replace elements that have been cut. Individual procedures were implemented for each type of intersection, so that no excess elements would be created. Only four to nine new elements are created for each cut element, depending on the type of intersection. This minimal subdivision uses only the original vertices of the element and vertices created due to the cutting action: one vertex at the location of each face intersection, and two vertices at the location of each edge intersection. For example, Figure 4 demonstrates how each half of the cut element from case *ii* in Figure 3 is minimally subdivided. In case *ii*, six elements are created to replace the original one. The five different intersection cases are shown in exploded view, with the same numbering as in Figure 3, in Figure 5. These subdivisions contain, respectively, 4, 6, 6, 8, and 9 new elements to replace every intersected element.

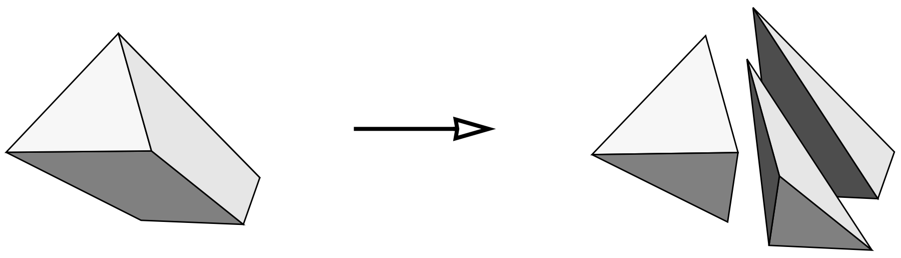

FIGURE 4. Minimal element subdivision.

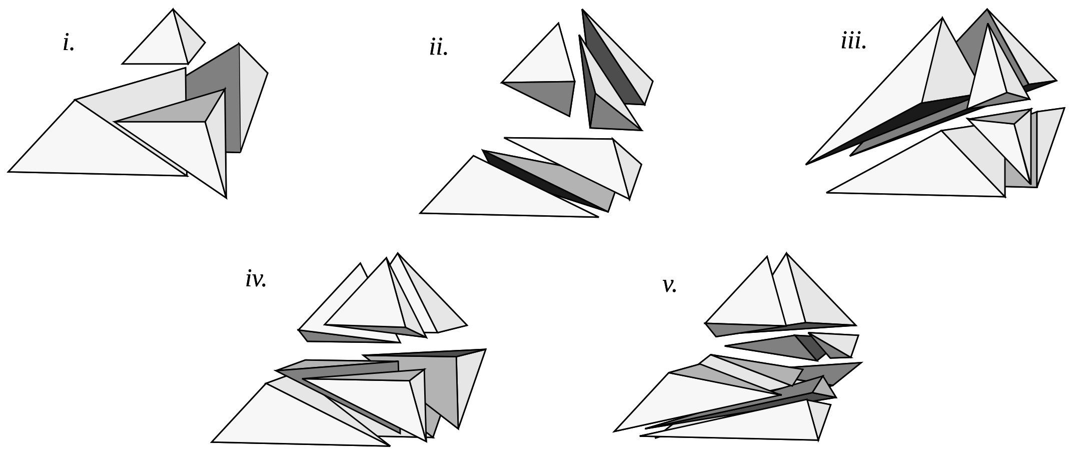

FIGURE 5. Minimal element subdivision, exploded view.

To determine these minimal subdivisions, each case was examined by hand to find the minimal set to replace the original element. The resultant elements were then encoded based on which edge and face intersections were present in the cut element. For element subdivisions that could be easily broken down, like the top and bottom portions of case *ii*, a separate subroutine was written to subdivide the six vertices present into three new elements. In doing so, the routine also checks to see if, on the faces of the segment with four vertices, any of the diagonal edges already exist within the model, to ensure that the model is internally consistent across element boundaries, without any crossing diagonal edges.

Within the framework of the tetrahedral mesh numbering, there are multiple orientations of the cut element based on the ordering of the vertices and cut edges. When the element is completely cut into two, there are four different permutations when three edges are cut, as in case *i*, and three different permutations in case *ii*, when four edges are cut. The four different permutations of case *i* correspond to each of the four vertices being cut away from the remaining three. When only one edge is cut, as in case *iii*, there are six permutations, and there are twelve permutations for both cases *iv* and *v*.

Each procedure uses a lookup table to determine how to mirror or rotate the vertices to fit the default orientation. The lookup table basically reorders the numbering of the vertices of the element. After the ordering is determined, any new edges that are needed are created. Then the new tetrahedra are created and the original tetrahedron is removed.

5.2.3 Comparison of Minimal Sets to General Subdivision

A complete, general, subdivision (1 vertex on each face, and 1 vertex in the middle of each edge) yields 17 news elements for every intersected element. By a careful examination of the possible intersection scenarios, we generate an intelligent subdivision which only creates between five and nine new elements per intersection.

The general subdivision is shown in Figure 6, where the circles are the locations where new vertices are inserted. Two vertices are inserted at edge intersections, and one vertex is inserted at all other locations.

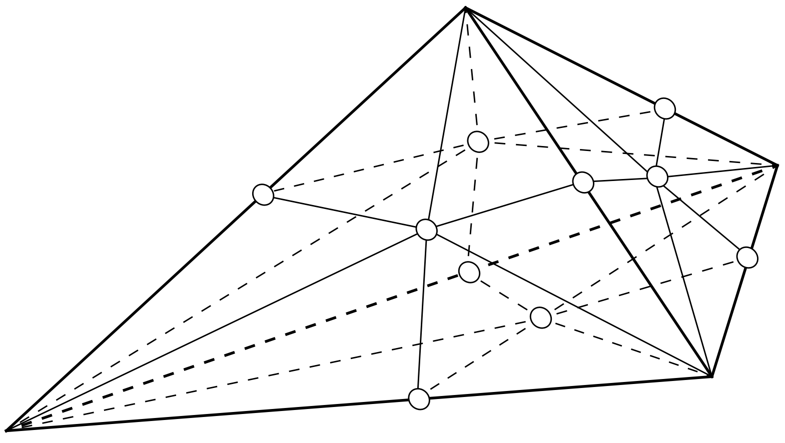

FIGURE 6. General tetrahedron subdivision.

5.2.4 Intersection Detection and Propagation

Cutting with a scalpel can be viewed as the motion of a finite length cutting edge passing through an object. If the body of the blade is ignored, and the edge is taken to be infinitely sharp, then the problem is reduced to tracking the passage of a line segment corresponding to the cutting edge moving in time and space through a tetrahedral mesh. Just checking to see if the cutting tool is in the interior of an element is not sufficient, since the cutting edge could pass through an element between time steps. Figure 7 shows the path of a cutting edge from time t_i to t_{i+1}, as it creates two face intersections and one edge intersection.

The swept surface created by the path of the cutting edge must be tested at every time step for intersections with the model. Two tests are required: the intersection between the path of the tip of the cutting tool and the faces of the tetrahedron, and the intersection of the swept surface and the edges of the tetrahedron. The path of the tip of the cutting tool is a line segment whose endpoints are the positions of the tip of the cutting tool at time t_i and t_{i+1}. The swept surface is a quad whose vertices are the

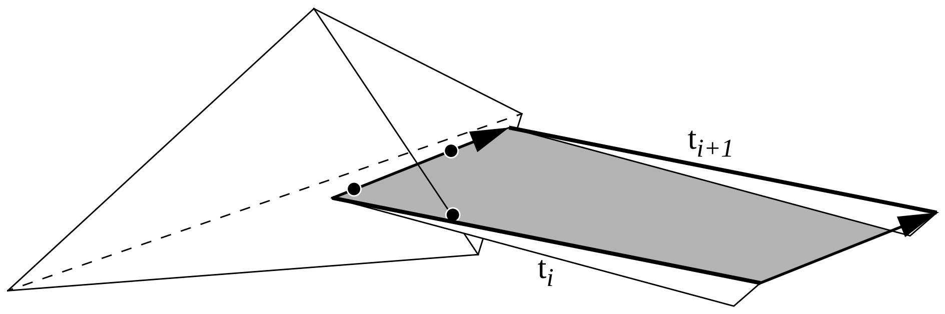

FIGURE 7. Cutting edge intersection with a tetrahedron.

endpoints of the cutting edge, both the tip and the base, at time t_i and t_{i+1}. These two tests generate, respectively, face intersections, which mark the base of the cut, and edge intersections, which occur where the model is split in two by the cutting edge.

The procedure for updating the intersection state of the model starts with a global search to determine if the path of the blade has intersected the model. If any tetrahedron has been intersected, then all 6 of the tetrahedron's edges and all 4 of its faces are tested against the swept surface and cutting tip path. After all the tetrahedra are tested, if any were intersected, the model is marked as having an intersection present.

Next, if the model has been intersected, all of the intersected elements are checked to see if the cutting instrument has either passed through any non-intersected faces or edges, or has left the tetrahedron. If a new intersection occurs, then the intersection information for that tetrahedron is updated. Neighboring elements that also contain the newly intersected edge or face are updated and tested against the motion of the cutting tool, thereby using spatial coherency to propagate the cutting motion through the model. If the cutting edge no longer passes through an intersected element, then the user has completed the cut, and the element will be subdivided.

Cuts are assumed to pass through an element, such that only one intersection exists per face or edge, and elements are subdivided after a cut is completed. Cuts where a tool enters and leaves the element through the same face are not modeled.

5.2.5 Intersection Testing

The technique used for the actual intersection tests is based on the ray-triangle intersection routine described in [30]. The implemented method is a fast intersection routine, which returns, when an intersection occurs, the parametric distance along the ray to the triangle, and the coordinates of the intersection point within the triangle. The original routine in [30] was modified to generate the intersection between a triangle and a finite length edge instead of an unbounded ray.

To accelerate the initial intersection detection between the cutting tool and the triangles, a bounding sphere test between each surface triangle of the model and the swept surface traced out by the scalpel between time steps is performed. If the spheres do not overlap, that triangle is not tested further.

The first intersection test for any triangle that passes the bounding sphere test is done between the current position of the triangle and the path of the tip of the cutting edge traced between time steps. If this test fails, then each edge of the triangle is intersected against the quad traced out by the motion of the cutting edge between time steps. The edge-quad test is performed as two edge-triangle tests, with the quad split into two triangles.

5.2.6 Intersection Coordinates

Once a collision between the cutting tool and the model is detected, the local coordinates of that intersection are used. For an edge intersection, the local coordinate is the value between 0 and 1 that encodes the relative distance along the edge for the intersection point. For a face intersection, the coordinates, *u* and *v*, are the distances along two of the edges to the intersection point within the triangle. Using the local coordinates, both single value coordinates for edge detection and two value coordinates for intersections on element faces, allows us to a use a simple encoding for the actual intersection point. The coordinates, which translate cartesian positions into a local reference frame within the edge or triangle, are also used to propagate not only the current position of the intersection to any new vertices created, but also the rest position and current velocity of the new vertices. Given the local coordinates, any value at the vertices, not just position, can be transformed.

The equation to transform local coordinates back into cartesian space, or to transform any value at the endpoints to the interior is, for edge intersections:

$$V_u = (1-u)V_o + uV_1 \qquad \text{(EQ 1)}$$

where V_u is the transformed value at the point represented by *u*, and V_0 and V_1 are the values at the two endpoints of the edge.

For face intersections, the equation is:

$$V_{uv} = u(V_1 - V_o) + v(V_2 - V_o) + V_o \qquad \text{(EQ 2)}$$

where V_{uv} is the transformed value at the point represented by u and v, and V_0 and V_1 and V_2 are the values at the vertices of the triangle. Note, both vectors and singular values can be easily transformed with these equations. Also, for the interior point to be within the triangle, the sum of u and v has to be between zero and one, inclusive.

5.3 Progressive Cutting

Cutting through soft tissue is a procedure where the user expects immediate visual feedback as to the progress of the cut she is generating. Therefore, updating the model while the user is cutting it is required. There are two types of progressive cutting: the first method is to wait until the user completes cuts through individual elements, and then subdivide each element; the second method is to generate temporary subdivisions within elements as the user moves the cutting tool through the object. The first method minimizes computation load during cutting, but generates a small amount of lag on the order of the typical edge length within the model. The second method removes the lag, but takes more processing time and may generate very small elements.

5.3.1 Progressive Cutting Between Elements

Progressive cutting between elements generates the subdivision of elements after the cut through the individual element is completed. As the user moves through an element, the intersections between the cutting tool and the element are detected and stored. Once the cutting tool leaves an individual element, the element is permanently subdivided. This complete process is shown in the following pseudocode, which has been simplified by leaving speed up techniques and cut propagation out:

```
sweptQuad = Motion(CuttingEdge, PrevCuttingEdge)
sweptLine = Motion(CuttingTip,  PrevCuttingTip)
if ModelNotIntersectedLastTimeThrough
     foreach (SurfaceTriangle in Model)
          if (Intersect(SurfaceTriangle, CuttingEdge) or
              Intersect(SurfaceTriangle, sweptLine)   or
              Intersect(SurfaceTriangle, sweptQuad))
               AddToIntersected(SurfaceTriangle->Element)
               ModelIntersected = TRUE
          endif
     end
endif
if ModelIntersected or ModelIntersectedLastTimeThrough
     foreach IntersectedElement
          IntersectElement(sweptLine, sweptQuad)
          if CutComplete(IntersectedElement)
               SplitElement()
               RemoveOriginalElement()
          endif
     end
endif
```

For example, in Figure 8, from left to right, first the cutting tool enters the left element. As the cutting tool moves within the element, any new edge or face intersections are detected and stored. Lastly, when the cutting tool leaves the element, it is subdivided. The process then continues for that original element's neighbors.

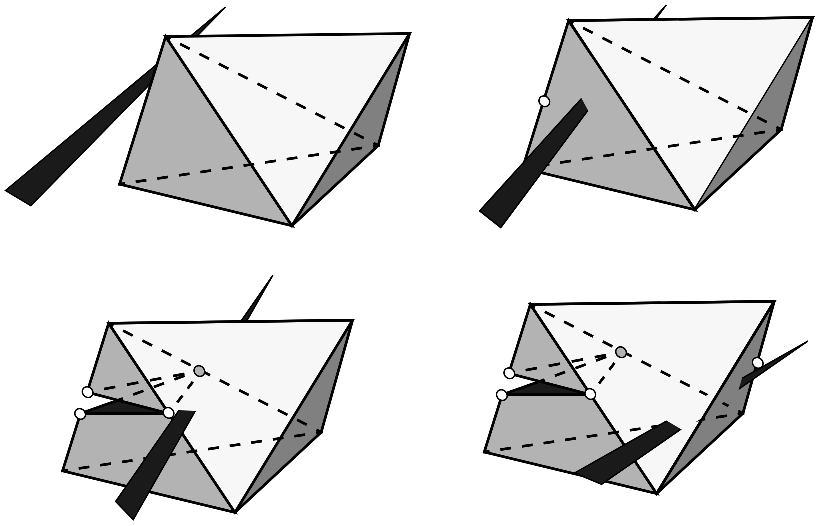
FIGURE 8. Progressive cutting between elements example.

One significant problem presents itself when implementing progressive cutting between elements. This occurs when two neighboring elements share an edge that is cut in two. When the first element is subdivided, two new vertices are inserted along each cut edge, and are used in the new elements that replace the first element. But, since the second element has not been subdivided yet, the new vertices are not connected to the second element. This is shown on the left side of Figure 9. Since the new vertices are not connected to that neighboring element, the new elements are free to rotate about the original vertices of the cut edge. This is shown on the right side of Figure 9, where the two neighboring elements were originally connected by the cut edges. Now, the new elements are attached to the neighboring element only by the original vertices of the cut edges, and the two new sections are shown rotating away from each other due to a combination of internal and external forces and an insufficient number of attachment points to the rest of the model.

This problem is alleviated by effectively attaching any new vertices generated by an edge intersection to their parent edge, as long as that edge exists. Since cut edges are removed once all the elements containing that edge are removed due to intersections, the edge will only exist as long as an original element that contains it has not been subdivided. This, then, is the indicator of whether an edge has been completely cut through or not.

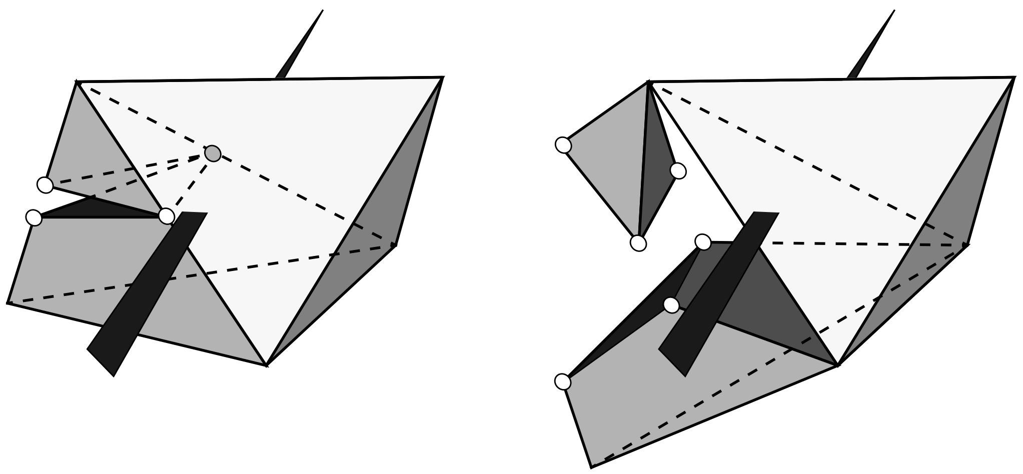

FIGURE 9. New vertices not connected to unintersected element.

The first step in this process is, after determining all the interior forces generated by the soft tissue and all the external forces acting on the nodes, to check all the edges for the case where the edge still exists but also has children vertices caused by a cut. For those edges that fit this criteria, first the intersection coordinate for the original intersection is ascertained. Then, the total force acting on the children vertices is summed, and then divided between the two vertices at the ends of the original edge. This step transfers the force acting within the new elements along the original edge to the original elements that still exist using that edge. Then, after the overall state of the object is updated, the same cut edges before are cycled through, and the child nodes of the cut edges are then moved to the correct position along the deformed original edge and their velocities are updated, using the initial intersection coordinate and Equation 1. This process is shown in the following pseudocode:

```
UpdateVertexForces()
foreach Edge that is Intersected
     u       = Intersection Coordinate
     f_total = Edge->Child_Vertex[0]->total_force +
               Edge->Child_Vertex[1]->total_force
     Edge->Vertex[0]->total_force += (1 - u) * f_total
     Edge->Vertex[1]->total_force +=  u      * f_total
end
UpdateModelState()
foreach Edge that is Intersected
     u       = Intersection Coordinate
     Edge->Child_Vertex[0,1]->current_pos =
          Transform_Vector(u,
                           Edge->Vertex[0]->current_pos,
                           Edge->Vertex[1]->current_pos)
     Edge->Child_Vertex[0,1]->current_vel =
          Transform_Vector(u,
                           Edge->Vertex[0]->current_vel,
                           Edge->Vertex[1]->current_vel)
end
```

5.3.2 Progressive Cutting with Temporary Subdivisions

Previous methods of modifying objects, and the basic technique described above, do not split an element until the cut has been completed. When the element size is large, this can introduce a noticeable lag into the cutting process. We have implemented a method of progressive cutting that generates a minimal subdivision of a partially cut tetrahedron. The subdivision is always based on the geometry of the original element, not of the previous temporary subdivision, thereby minimizing a potential source of error.

The general procedure for progressive cutting utilizes a temporary subdivision of each partially cut element, which is added to the general process described previously in Section 5.3.1. An example cut is shown in Figure 10. First, any temporary face intersections caused by the cutting edge are updated for each partially cut tetrahedron. A temporary face intersection occurs when the cutting edge, not the tip of the cutting tool, currently intersects a face. This type of intersection does not occur for the permanent intersections described previously. Then, the modified topology of the partially cut element is checked for any changes. A change occurs when a new intersection is created: for example, when the element is first cut into, or when the cutting edge or tip passes through another edge or face. If the topology has changed, a new minimal set of temporary tetrahedra are created and all the old temporary tetrahedra are removed. If the modified topology has not been changed, then the temporary elements are updated using the new positions of any temporary face intersections. Once the cutting edge leaves an element and the cut is completed, the temporary elements are removed, and a final subdivision is created.

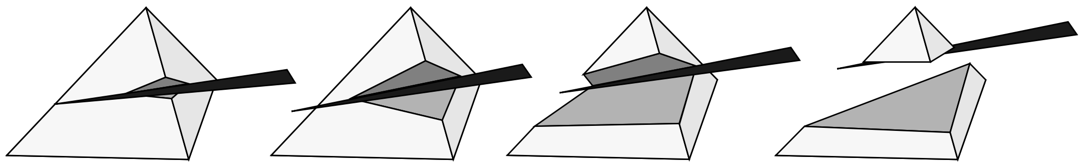
FIGURE 10. Progressive cutting with temporary intersections example.

As the progressive cutting moves through the model, the cutting routine modifies the underlying soft tissue model at each time step, which is much more frequent than the changes to the model described in Section 5.3.1. When the initial intersection occurs, all contributions to the stiffness and mass of the model based on the original element are removed. Then, the contributions based on the new, temporary elements are added. As the cutting instrument moves with respect to the model, the initial contributions due to the temporary subdivision are removed, in turn, and the new contributions are added based on the current geometry of the temporary subdivision. The changes in the pseudocode between the two types of progressive cutting are shown below, replacing the second section of the first pseudocode in Section 5.3.1 with:

```
if ModelIntersected or ModelIntersectedLastTimeThrough
    foreach IntersectedElement
        IntersectElement(sweptLine, sweptQuad)
        if CutComplete(IntersectedElement)
            RemoveTemporaryElements()
            SplitElement()
            RemoveOriginalElement()
        else
            RemoveContributionFromOriginalElement()
            RemoveTemporaryElements()
            TemporarySplitElement()
        endif
    end
endif
```

5.3.3 Different Possible Cases for Temporary Progressive Cuts

We have enumerated eleven different combinations of intersected edges, faces, and temporary face intersections, which are enumerated in Table 1. The different types of intersections are illustrated in Figure 11. Momentarily marking temporary intersections as permanent, many of these cases can directly use the procedures described in the previous section on minimal cutting. The cases which are not topologically similar to those described in Section 5.2.2 were implemented in a similar fashion, with a minimum number of new elements generated for each cut element, where each case was examined by hand to determine the correct subdivision.

Case:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**[a]	**10**	**11**
Edge Ints.	0	0	1	1	1	2	2	2	2	3	3
Face Ints.	1	2	0	1	2	0	1	2	2	0	1
Temp. Face Ints.	1	2	2	1	2	2	1	2	2	2	1
Interior Ints.	1	0	0	1	0	0	1	0	0	0	1

TABLE 1. Enumeration of different cases for temporary intersections.

a. Same number of intersections as case 8, but case 9 has a different subset of edges intersected.

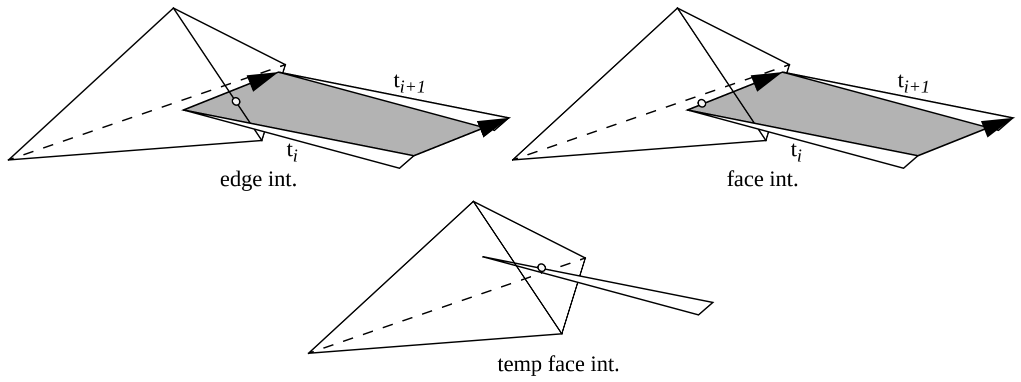

FIGURE 11. Different types of intersections for progressive cutting.

Figure 12 shows the topology of the different cases. Note the difference between the relationship of the edges intersected in cases 8 and 9. As described previously, a lookup table is used on the intersected element to rotate and mirror its state to fit the orientation of the default topology.

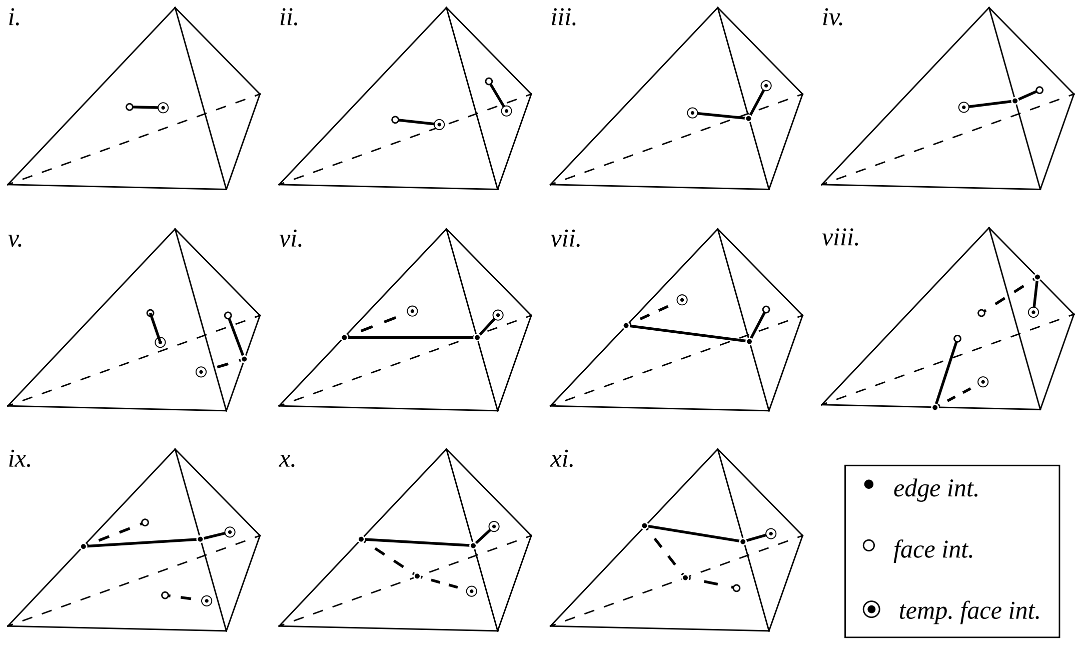

FIGURE 12. Eleven different progressive cutting cases.

5.3.4 Progressive Cutting: Cutting Tip Within Model

When the tip of the cutting instrument is within the interior of an element, ideally we would want the model to be able to open up along the cut of the blade, so that the user could see all the way up to the base of the cut, where the tip is located. But given the nature of the subdivision for a generic cut, this would not be possible. An example of this is shown in Figure 13, case *i*, which corresponds to Case 1 from Table 1. In this case, the tip of the blade is inserted fully through one face, with the tip of the cutting edge within the interior of the tetrahedron. I_i is the tip of the cutting edge, I_f is a permanent face intersection, and I_{tf} is a temporary face intersection. There are now two intersections on one face, one permanent and one temporary. Ideally, as described above, the object would be able to open up along the edges between the two face intersections, as shown in Figure 14. To be able to see the tip of the cutting tool, though, we would have to insert two intermediate vertices along the line between those two intersection points, and generate at least 8 more edges and 8 more temporary elements. This would be very computationally expensive to perform because of the large increase in the number of elements that would be generated.

If no intermediate nodes are inserted between the two face intersections, a straight line will always connect them, and the model will not be able to open up. Additionally, the fact that the model can not open up along these edges allows us to ignore the location of the tip within the model, and, in fact, to generate an arbitrary topology within the interior of the original element, since that topology will never be seen.

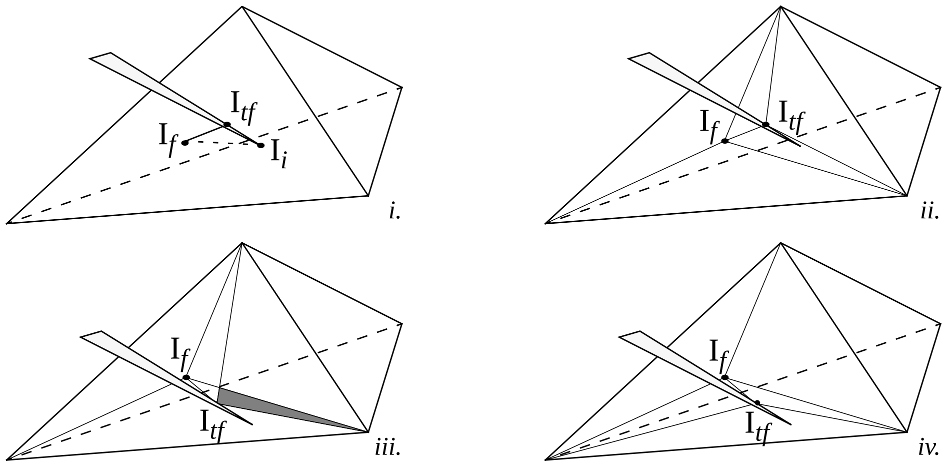

FIGURE 13. Temporary subdivision, with two intersections on one face.

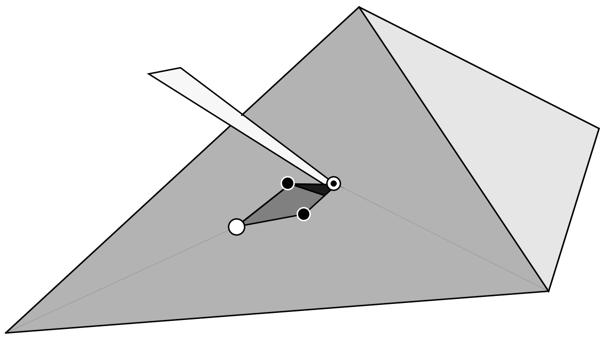

FIGURE 14. Temporary cut opening up between face intersections.

5.3.5 Progressive Cutting: Topology Change

Even though we may be able to ignore the position of the tip of the cutting tool within the element, we still have to make sure that none of the triangles generated on that original face overlap. This would occur, as shown in Figure 13, as the blade travels from case *ii* to case *iii*, when a temporary face intersection moves across an edge belonging to the other face intersection. The shaded area shows the overlapping area of the two triangles. If this occurs, then the modified topology of the partially cut tetrahedron has changed, and a new set of tetrahedra will be created, as is shown in case *iv*.

5.4 Stable Cutting With Snapping

Progressive cutting with the cut surface following exactly the path that the user traced out is the ideal. This motion, though, will create small elements when the cutting tool passes close to one of the original vertices in the model. Small elements are also created due to the temporary subdivisions, when the cutting edge is close to an edge of the original element. These small elements have either short edges or short height. They can become unstable due to the nature of the simulation technique that generates a set of stiff ordinary differential equations updated explicitly.

To counteract this problem, we utilize a snapping method with the progressive cutting between elements technique to assure that these small elements will not be created. In the following sections we will describe the general concept of the method, the test for determining probable stability, how we find the collection of perturbations to the original set of intersections that will generate a stable subdivision, and an enumeration of all the possible topological cases of intersections with snapping.

5.4.1 General Concept

Once a cut has been completed through an element, normally, we would just subdivide the original element. But, with model instability an issue, we first need to verify whether or not the desired subdivision is stable or not.

If the initial subdivision would not be stable, then we set up a list of possible permutations of the initial intersection state. The permutations are all the possible combinations of moving one or a combination of the intersections to their closest feature. The list is ordered based on a metric of how close the perturbed intersection state is to the original state, and is described in detail in Section 5.4.4.

After the list is set up, we generate, in turn, a test subdivision for each permutation in the list. If the currently tested permutation is stable, then we do a final subdivision based on that permutation, and return to the cutting routine. If the currently tested permutation is not stable, then we go to the next one in the list.

If none of the permutations generate a stable subdivision, then we utilize the best permutation from the list of possible cases that was generated, utilizing the stability values returned from the geometry test. This process is demonstrated in Figure 15.

To more clearly demonstrate the process, take, for example, Figure 16. In this element, there are 3 edge intersections, effectively cutting off the top of the tetrahedron. If testing of the initial subdivision, using the actual intersection points, determines that the subdivision would be unstable, then we will attempt snapping the intersection points. For each intersection point, we can try snapping to the closest vertex or

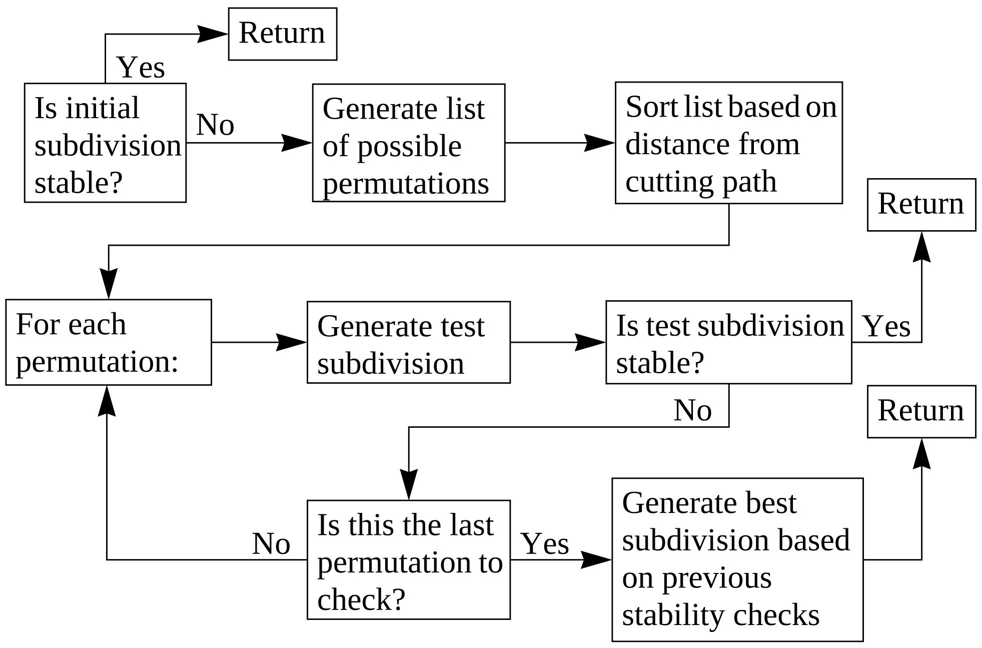
Return
Yes
Is initial subdivision stable?
No
Generate list of possible permutations
Sort list based on distance from cutting path
Return
For each permutation:
Generate test subdivision
Is test subdivision stable?
Yes
No
Return
No
Is this the last permutation to check?
Yes
Generate best subdivision based on previous stability checks

FIGURE 15. Flow of stable subdivision routine.

snapping to the center of the edge. We can try snapping individually, in pairs, or all 3 at once. The method of ordering the list is, as previous described, based on the distance from the perturbed intersection state to the initial cutting path. In this example, though, the first attempt moves intersection 0 (I_0) to vertex 0 (V_0). In this example, that permutation is also not stable. The second attempt moves intersection 1 (I_1) to V_0. In this example, that permutation, again, is not stable. The third attempt then tries to move both I_0 and I_1 to V_0. In this case, the resultant permutation is stable, and the subroutine returns with that final subdivision.

5.4.2 Geometry Test for Element Stability

Model stability will be described in detail in Section 6.2.5. We have found that overall model stability is dependent on individual element stability, such that if an individual element becomes unstable, it forces the whole model to become unstable. Therefore, we test individual elements to determine if they meet stability criteria.

To determine whether or not an individual element will likely become unstable depends on the geometry of the element. We use a simple test where the rest length of all the edges and the height of the vertices above their opposing faces are compared to a minimum value. Figure 17 shows both edge length and vertex height within a typical element. If all the values are greater than the minimum value for this element, then the individual element, and therefore the model, will remain stable. If any of the values are below that threshold, then the model may remain stable, but probably will not be

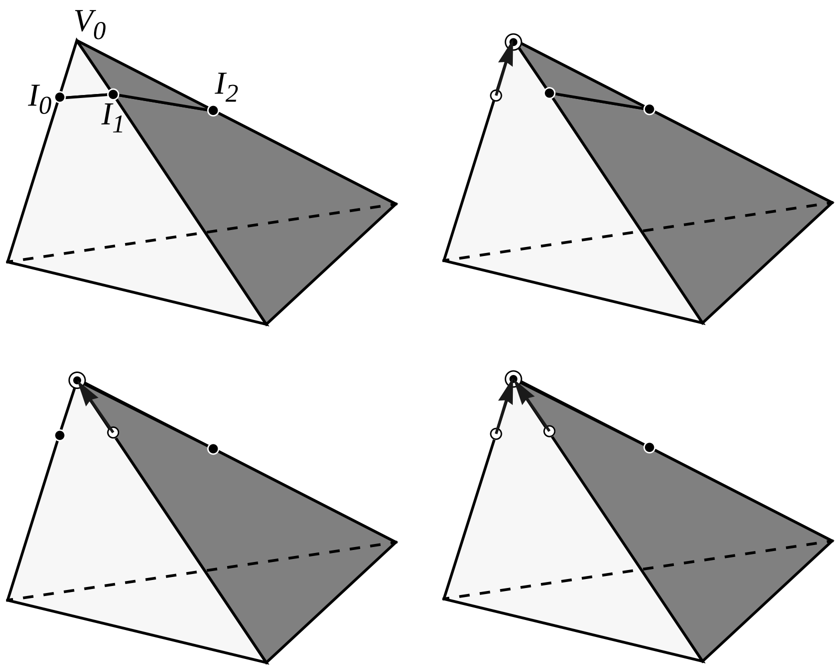

FIGURE 16. Example of stable snapping.

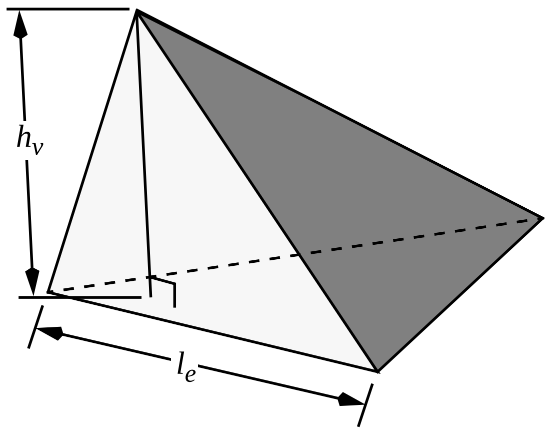

FIGURE 17. Element edge length and vertex height.

stable. So, the threshold is set at a safe value, not right on the cusp of causing the model to become unstable.

5.4.3 Where Intersection Points Snap

Figure 18 shows the two types of intersections and the features they can be snapped to. The edge intersection can be snapped either to its closest endpoint or to the center of the edge. The face intersection is more complicated. The shaded circle around the intersection represents the thresholded distance for snapping to the second-closest edge, if that edge has already been intersected. Before we determine where we will snap the face intersection, we check to see if the closest edge to the face intersection has been intersected and already used in a subdivision of a previously cut element. If the closest edge has not, then we check to see if any of the other edges on the intersected face have been intersected and used in a previous subdivision. If one has, and the edge is close enough to the face intersection, that edge is used as the closest edge, since we will then be snapping to a previously determined and used intersection point.

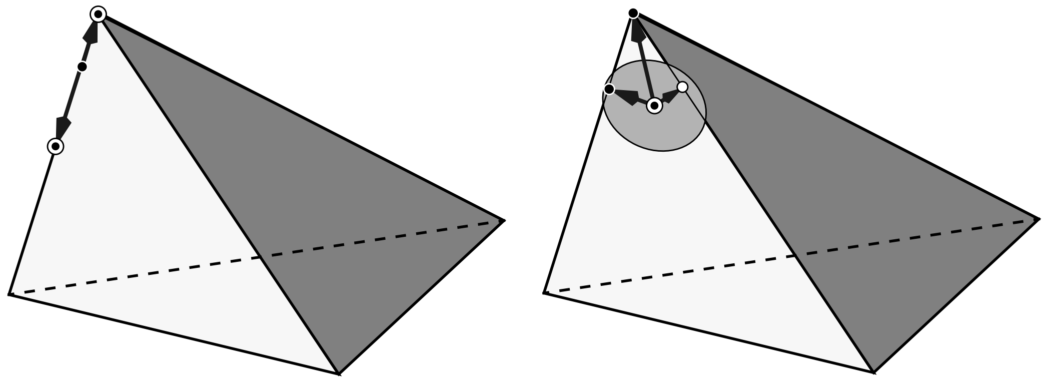
FIGURE 18. Where edge and face interaction points snap to.

After we have determined which edge on the intersected face is the "closest" edge, we project the face intersection to that edge. If we are forcing the snapping to go to the center of the edge and the projected point is in the middle 50% of the edge, then the projected point is moved to the center of the edge. Next, we test to see if the projected point is too close to one of the endpoints. If it is, then the intersection point is automatically forced to the closest vertex. If not, then if there is already an intersection on that edge, we snap the face intersection to that point. Lastly, if the projected point is far enough away from the endpoints of the edge, and there are no intersections already present on this edge, then the face intersection is snapped to the projected point on that edge.

Additionally, in the case that we have both a face intersection and an edge intersection of the same face, if the edge intersection is snapped to one its endpoints, then we make sure that if we try to snap the face intersection to that edge, the face intersection is forced to one of the endpoints, and not allowed to snap to the interior of that edge. It would not make any sense to move an intersection off an edge, and then put another intersection right on it. This is shown in Figure 19, where in the first step, an edge intersection is snapped to a vertex. Then the face intersection, which was initially further from its closest feature than the edge intersection was from the vertex, is then snapped. First we attempt to move it to the edge, and then realizing that the edge had already been snapped, we move it to the vertex that the edge snapped to.

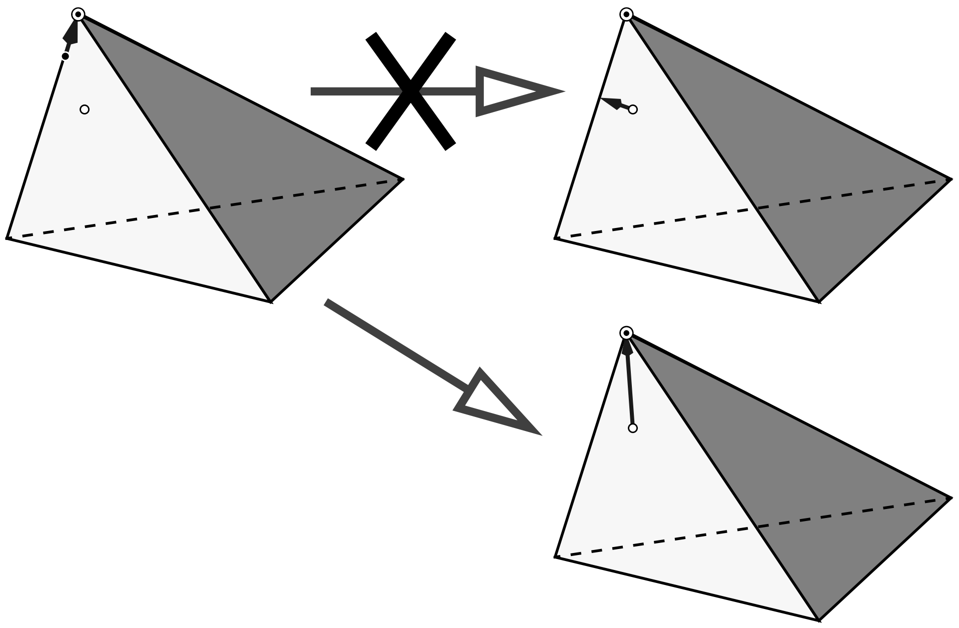

FIGURE 19. Face snapping to vacated edge.

5.4.4 Ordering of Possible Cases to Find Stable Subdivision

When the initial intersection state does not provide a stable subdivision, then snapping of one or more of the intersection points becomes necessary. The first thing that needs to be done, then, is to determine which collection of snapped intersection points is going to be most faithful to the original path traced by the user. While we are moving intersection points from their original positions, and therefore changing the location of the cut surface, we want to minimize the distance of the snapped cut surface from the initial cutting surface traced by the user.

For each intersection, the distance to the closest feature is recorded. For edge intersections, that closest feature would be the vertex at the end of the edge. For face intersections, there are more possibilities, as described in the previous section. Additionally, we can try moving edge intersections to the center of the edge, to maximize the distance from both endpoints while still maintaining the direction of the cut. Moving intersection points to coincide with original vertices of the element maximizes the likelihood of a stable intersection, since the whole original edge remains, and its edge length is no smaller than the original edge length.

We generate all the different combinations of moving initial intersection points, and sort the list based on the total distance that the vertices for each case have to move to reach their closest feature. For example, if there are 3 initial intersections, 2 face intersections and 1 edge intersection, then there are 7 different possible combinations of moving intersections: 3 combinations where we move single intersection points, 3 combinations of moving 2 intersection points, and 1 combination where we move all 3 intersection points. For each of these cases, we sum up the distance from the initial intersection points to where they will be moved, and then sort the list of possible combinations based on that distance. In this way, if the 2 face intersections were each 0.1mm from their closest edges while the single edge intersection was 1.0mm from its closest vertex, we would attempt moving both face intersections before moving the single edge intersection, given that the sum of the distances is 0.2mm vs. 1.0mm. Figure 20 and Table 2 enumerate the different permutations and the order of them for this example.

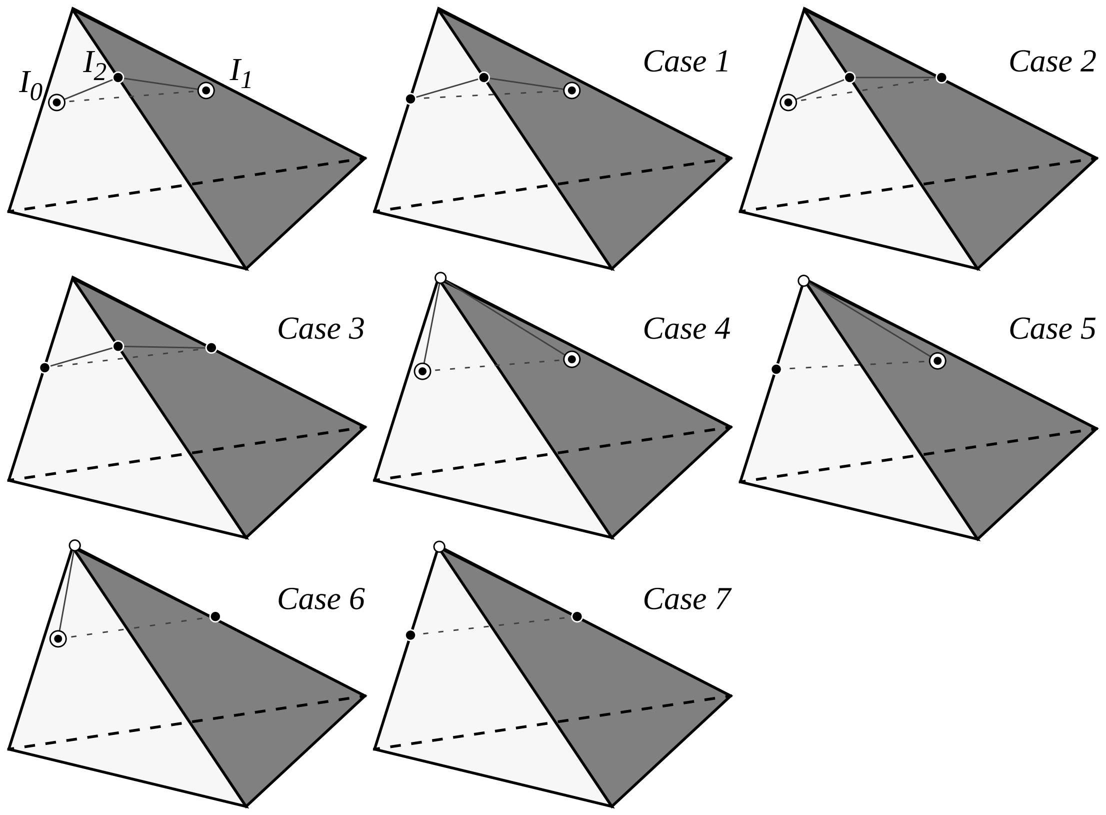

FIGURE 20. Sorting of possible permutations.

One restriction on the moving of intersection points is required. Just as we don’t move original element vertices, we can not move intersection points that have been used in previous element subdivisions. As the cut progresses through an object, we subdivide

Is intersection snapped?		Int. 0	Int. 1	Int. 2	Total Distance
Case:	1	Yes	No	No	0.1mm
	2	No	Yes	No	0.1mm
	3	Yes	Yes	No	0.2mm
	4	No	No	Yes	1.0mm
	5	Yes	No	Yes	1.1mm
	6	No	Yes	Yes	1.1mm
	7	Yes	Yes	Yes	1.2mm

TABLE 2. Sorting of possible permutations.

each intersected element in turn. After one element is subdivided, the new intersection points, which now have had vertices created at their positions, are fixed within the topology of the model. Since they are now part of permanent elements in the model, they can not be moved, since their change in position would affect the topology of not only their element, but due to stability issues, may affect one or more levels of modification back through the model. Therefore, as we move through the model, progressively cutting and subdividing elements, intersection points which are used in new elements are marked as unmovable. These unmovable vertices are not used in generating all the possible combinations of moving intersection points in subsequent elements.

Additionally, if the first pass through the possible combinations does not provide us with a stable subdivision, we go through the list again, forcing all edge intersections to snap to the middle of their edges, maximizing the possibility of a stable intersection while maintaining the general direction of the cutting path. If that pass does not provide a stable intersection, then we go through the list one last time, forcing all intersections to the closest original vertex of the element. If this pass also does not provide a stable intersection, then we use the combination that provided the most stable subdivision from the previous three passes through the combination list.

5.4.5 New Vertex Types

There are two basic types of new vertices created by cutting. Vertices created by the motion of the tip of the cutting tool are single vertices, where only one vertex is created at the intersection position. Vertices created by the motion of the cutting edge are created in pairs, so that the model can split apart where the cutting edge has parted it. This is simple to implement for the initial cutting, but a difficulty arises when snapping occurs.

For example, if there are neighboring elements that are cut, and one element is cut completely through at one time step. Then, at the next time step, the neighboring element is cut and the face intersections on it are snapped up to the edges it originally shared with its neighbor. So, now, the original paired vertices have to be replaced with a single vertex, since that area which was initially going to split open is now the base of the cut.

In the simulator, though, the replacement does not happen right away. There are many different data structures which include pointers to both vertices that were created when the first element was split. Because of this, it is simpler to symbolically link the two vertices so that their positions and forces are identical, and leave both vertices in the model. On the other hand, we don't want to increase the computational load unnecessarily due to the additional vertex and edges that link the two vertices. So, after the cut is completed and there are no more pointers in use for the two joined vertices, one of the vertices is removed and its edges linked to the other vertex of that pair.

5.4.6 Paired Vertex Above or Below the Plane

When cutting without snapping, vertices generated in pairs, caused by intersections along the cutting edge, are easily allocated as being above or below the cutting plane. As seen in Figure 21, when you insert two new vertices at an intersection along an edge, normally one is clearly connected to one endpoint, while the other is connected to the other endpoint. And, since the internal topology is clearly defined by the type of subdivision, connecting multiple paired sets of vertices is simple. An example where the edges from the paired vertices are drawn in bold is shown in Figure 21.

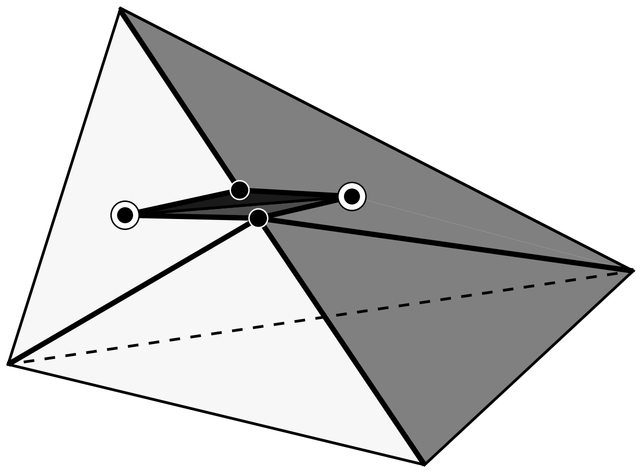

FIGURE 21. Paired vertices in an unsnapped intersection.

When a subdivision is caused by a snapped intersection, though, allocating and connecting vertices above or below the cutting plane is not as clear. If an edge intersection is snapped to a vertex, then the obvious connection of one new vertex to each endpoint of the edge is removed. For example, in the upper row of Figure 22 an edge intersection was snapped to V_0. Now, since the new vertices are not on an edge but at the endpoint, it needs to be determined which of the other vertices in the subdivision each new vertex will connect to. Looking at the geometry of this example, it is clear that the vertex above the plane, V_{0a}, should connect to V_1 and V_2, while the vertex below the plane, V_{0b}, should connect to V_3. Additionally, both new vertices will connect to the face intersections at the base of the cut. Note that labeling the two vertices as above or below the plane is arbitrary, as they exist at the same point in space, and each could easily take the place of the other.

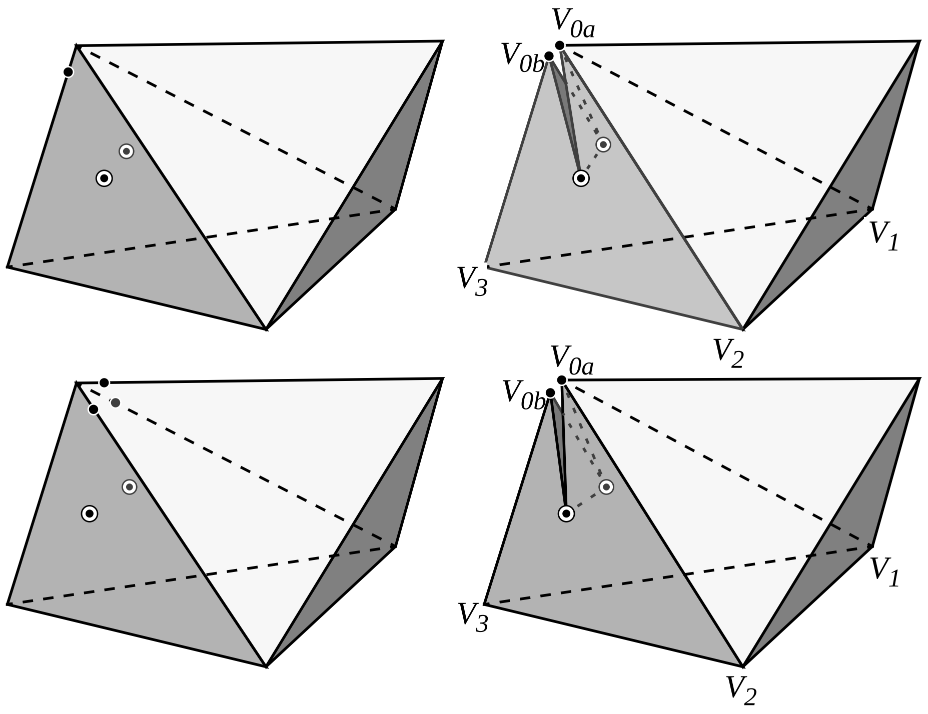

FIGURE 22. Example of paired vertices in snapped intersections.

Now, in Figure 22 in the bottom row, we have a different type of intersection. In this case, the upper element intersection was competed first, with all the edge intersections snapping to V_0. Since there is no internal subdivision to this element, only one of the paired vertices is used for the new element that replaced the original element, with the paired vertex kept as a placeholder. Then, when the cut through the lower element is completed, we have two face intersections and the paired vertices at the vertex that was previously snapped to, V_0. In this case, as in the previous example, it is clear that the vertex below the plane, V_{0b}, should connect to V_3, and the vertex above the plane, V_{0a}, should connect to V_1 and V_2.

To determine which of the paired vertices should be used in each new element in the subdivision of the original element, we look at the unintersected vertices present in the new element skeleton and their position with respect to the cutting plane. The skeleton consists of four vertex pointers, of which at least one of the vertices is an original vertex from the initial intersected element. Any intersected vertex or intersection point is considered to be on the cutting plane, while the unintersected vertices are checked to see if they are above or below the cutting plane. The total number of vertices above and below the plane are determined, and if there are both vertices above and below the plane, then the position of the new element with respect to the cutting plane is used. If the new element is predominately above the plane, then we use the paired vertex that is above the plane for this new element, otherwise we use the paired vertex below the plane. If only vertices above the plane, or vertices below the plane, are present, then we use the corresponding vertex of the paired set.

Figure 23 shows two examples of the cutting plane and element skeletons. In the element on the left, there are two sets of paired vertices, hidden from view, and 2 original vertices from the initial element. The original vertices are both above the cutting plane, which cause us to use the paired vertices that are labeled as being above the cutting plane. In the element on the right, though, one of the original vertices is above the cutting plane and the other is below. In this case, then, we look at the fraction of the new element volume that is above the cutting plane. Since in this case, the fraction above the plane is greater than 0.5, we consider the new element to be above the cutting plane, and use the paired vertices that are labeled as being above the cutting plane.

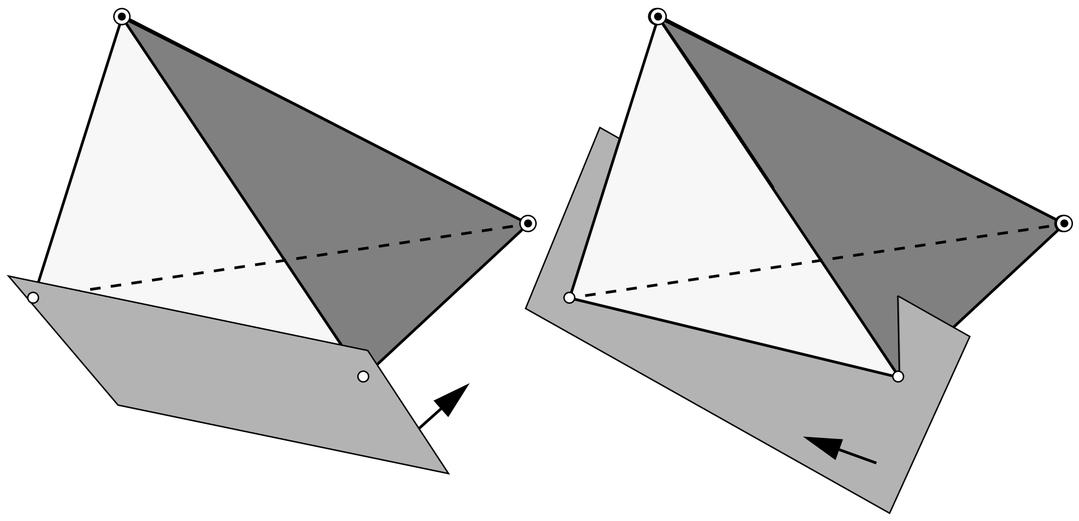

FIGURE 23. Example showing two different cases of paired vertices.

To calculate the fraction of the element above, or below, the plane, all that is required is the signed distance of each vertex from the plane. The non-trivial combinations of distances above, below, or on the plane are shown in Figure 24.

In the case where only one vertex is above or below the plane, or all vertices with non-zero distance are either above or below the plane, the fraction above the plane is

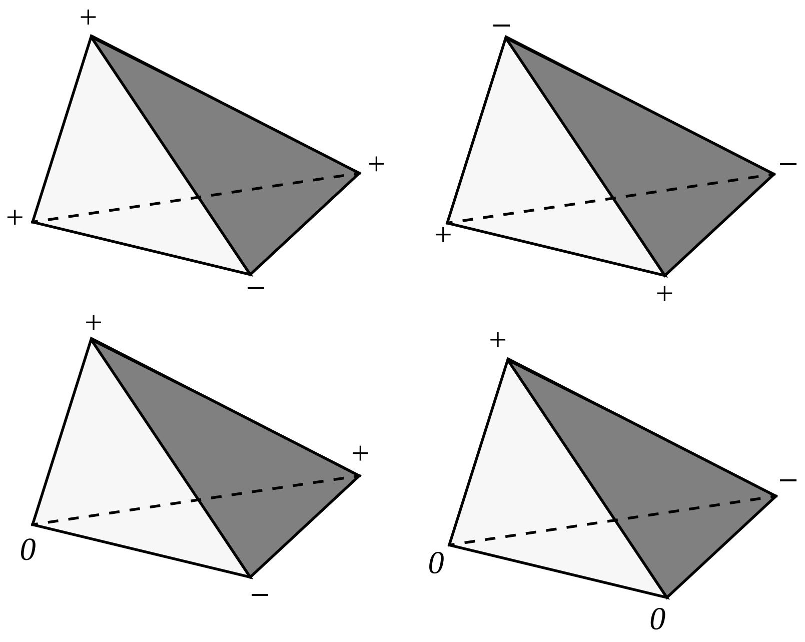

FIGURE 24. Non-trivial combinations of vertex distances above, below, or on the plane.

clearly exactly one or zero. If there are two vertices with non-zero distances, the equation for the fraction above the plane is a simple ratio of the distance from the vertices to the cutting plane:

$$F_a = \frac{d_a}{d_a + d_b} \qquad \text{(EQ 3)}$$

where F_a is the fraction of the volume above the plane, d_a is the distance of node *a* above the plane, and d_b is the unsigned distance of node *b* below the plane. Figure 25 shows this fraction graphically. The actual signed distance *along* the intersected edge could be used instead of the distance above the plane. This is because the distance *to* the plane is equal to the dot product of the cutting plane normal vector with the vector from the vertex to the intersection point. Since the normal vector is the same for both segments of the edge, the ratio of the actual distances is equal to the ratio of the distances from the cutting plane.

The rest of the cases derive from Equation 3. If there are more than 2 non-zero distances, then half of the element is split by one of the distance pairs, and then that new fraction is split again by one of the other distance pairs. In the case where there are 3 non-zero distances, this equates to:

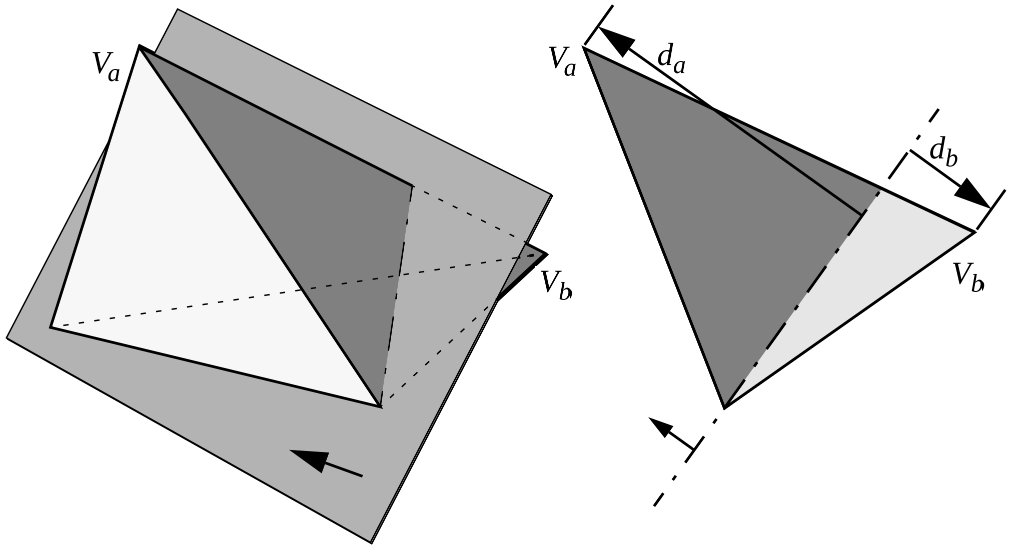

FIGURE 25. Volume fraction above the cutting plane.

$$F_a = \frac{d_a}{d_a + d_b}\frac{d_a}{d_a + d_c} = \frac{d_a^2}{(d_a + d_b)(d_a + d_c)} \qquad \text{(EQ 4)}$$

There are two different permutations when all four distances are non-zero. In the first case, there is one vertex above and 3 vertices below the plane:

$$F_a = \frac{d_a}{d_a + d_b}\frac{d_a}{d_a + d_c}\frac{d_a}{d_a + d_d} = \frac{d_a^3}{(d_a + d_b)(d_a + d_c)(d_a + d_d)} \qquad \text{(EQ 5)}$$

In the case where there are two vertices above, and two vertices below the plane, breaking down the resultant volumes goes through three levels, and the resultant fraction above the plane is:

$$F_a = \frac{d_a^2}{(d_a + d_b)(d_a + d_d)} + \frac{d_a d_b d_c}{(d_a + d_b)(d_a + d_d)(d_b + d_c)} + \frac{d_c^2 d_d}{(d_a + d_d)(d_b + d_c)(d_c + d_d)} \qquad \text{(EQ 6)}$$

5.4.7 Different Possible Cases For Snapped Cuts

Table 3 and Figure 26 demonstrate the different topological cases for snapped cuts. These 60 permutations shown in Figure 26, some of which have the same combination of intersected vertices, faces, and edges, were identified by first determining all possible permutations of intersection cases with an allowable number of intersections. Those possible cases were examined to weed out the cases which weren't truly possible due to geometric constraints. Lastly, during testing, additional cases were found to occur due to the snapping of intersections. Without snapping, a maximum of five intersections can occur on any one tetrahedron. But with snapping, we empirically found two cases with six intersections on an element. Also, cases which did not look geometrically possible without snapping did occur with snapping, and were similarly addressed. As in Section 5.2.2, each of these 60 cases was examined by hand to determine the minimal number of elements that needed to be generated to replace the original cut element.

Case:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**
Vertex Ints.	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	3	3
Edge Ints.	1	1	1	2	2	2	3	3	3	4	4	0	0	1	1	1	2	2	2	3	3	4	4	0	0	1	1	2	2	3	0	1
Face Ints.	0	1	2	0	1	2	0	1	2	0	1	1	2	0	1	2	0	1	2	0	2	0	1	1	2	1	2	0	1	0	1	1

TABLE 3. Enumeration of different cases for snapped intersections.

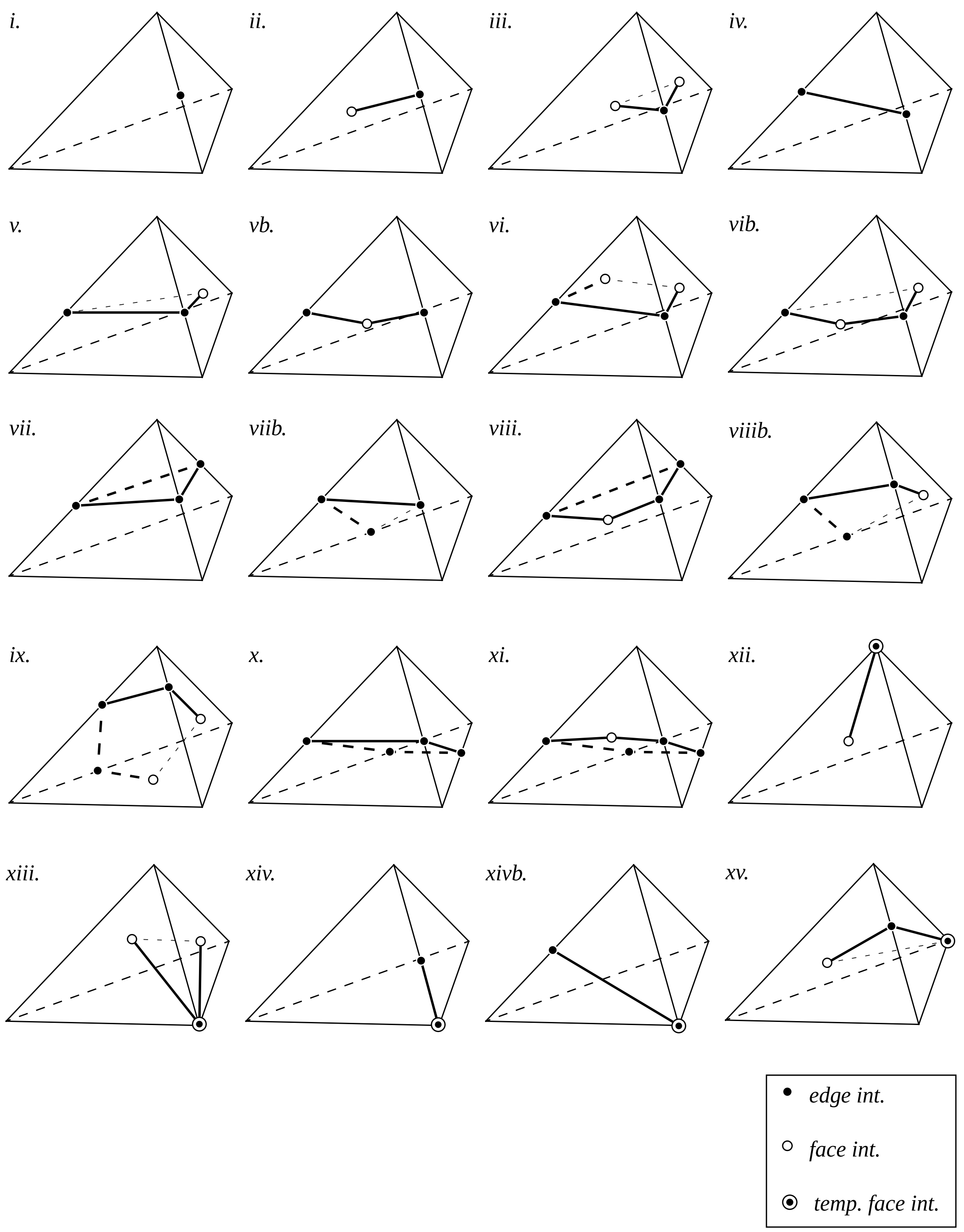

FIGURE 26. Snapped progressive cutting cases.

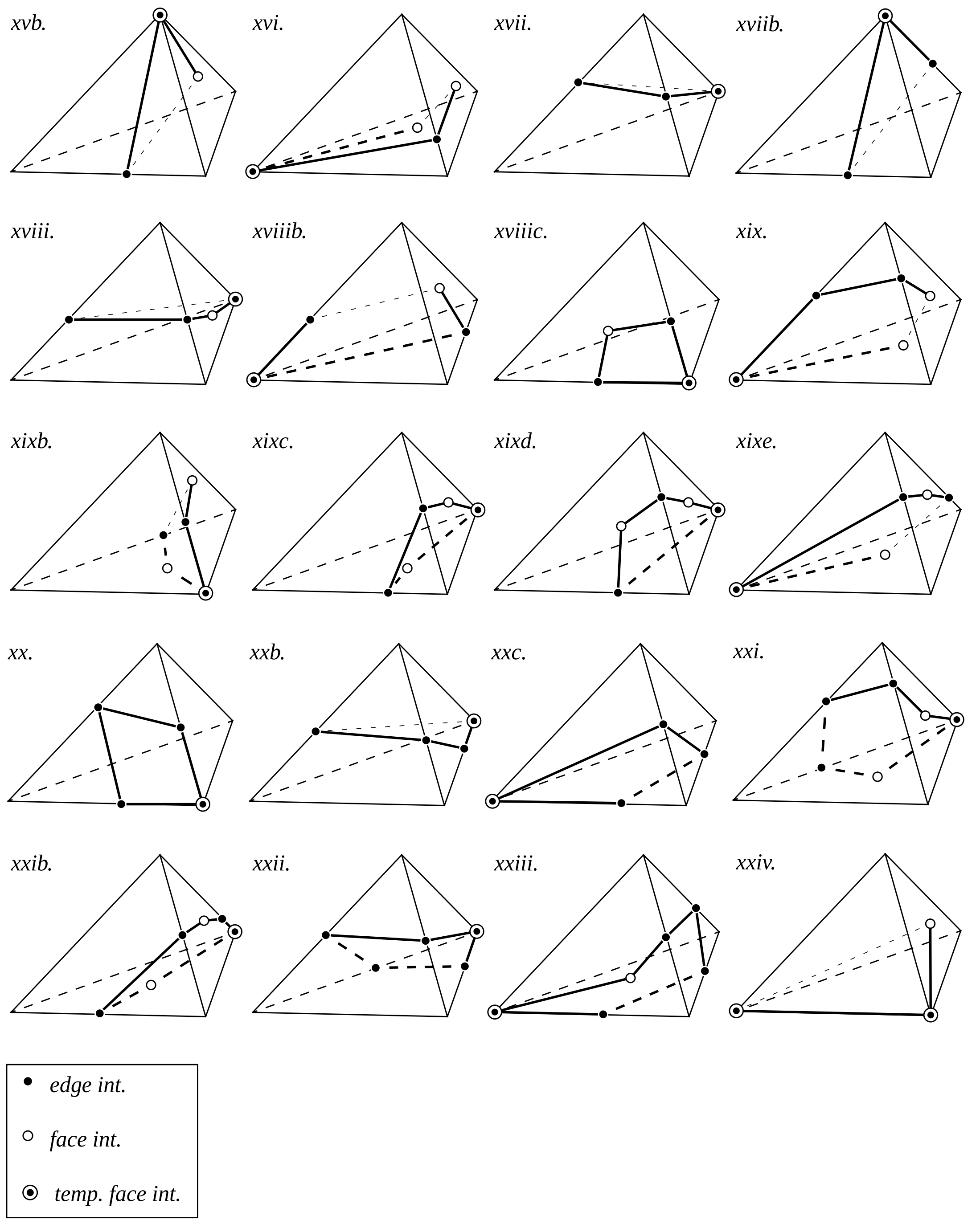

FIGURE 26. Snapped progressive cutting cases. (Continued)

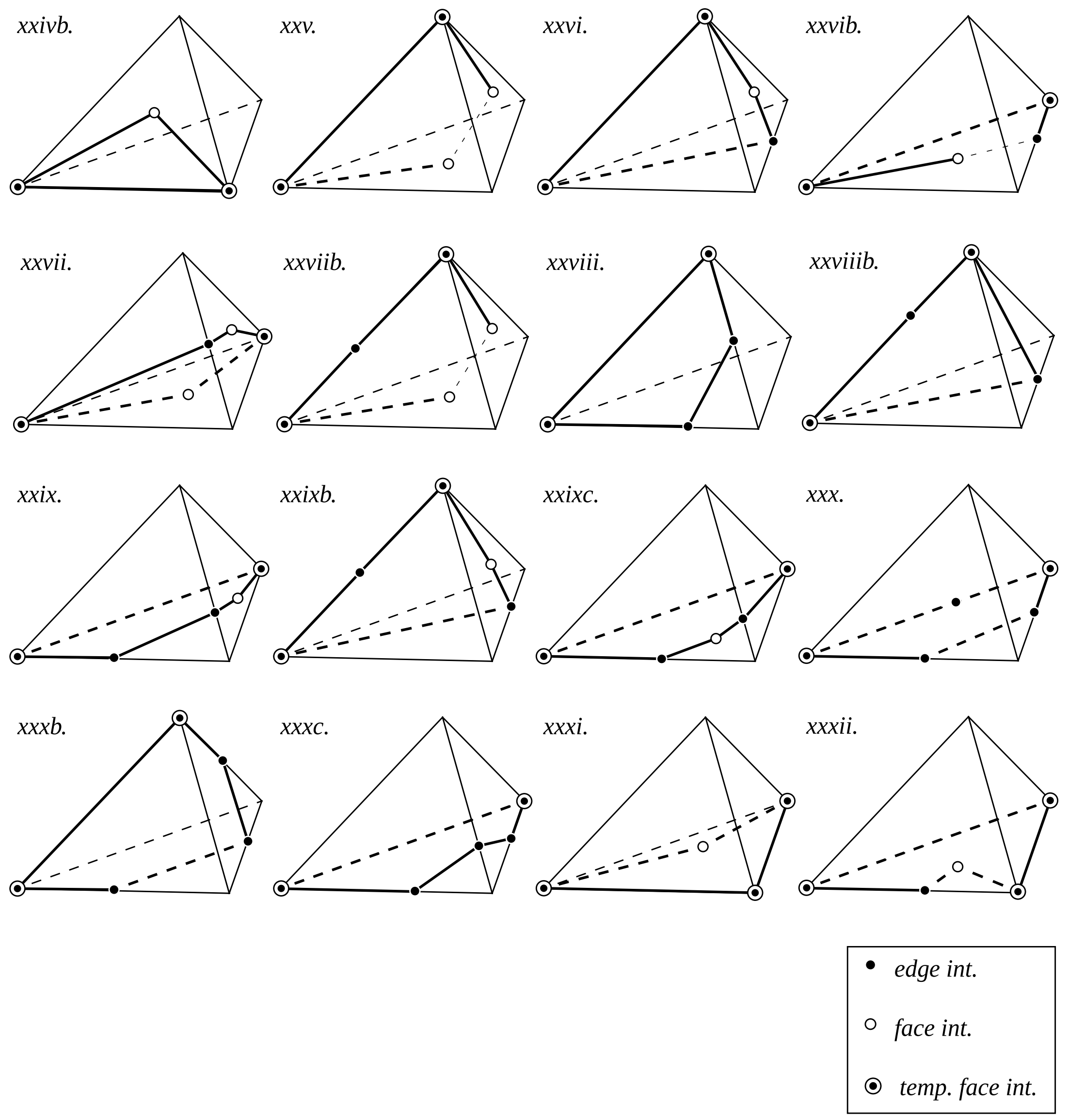

FIGURE 26. Snapped progressive cutting cases. (Continued)

Chapter 6

Soft Tissue Modeling

Physically based volumetric models provide the most accurate results for soft tissue simulation. The difficulty with these types of systems is the large amount of computation required. A finite element model with 500 elements will have a global stiffness matrix with a size on the order of 700 x 700, or larger. Inverting a matrix of that size or solving the constitutive equations is very time consuming. Mass spring systems can model objects volumetrically in a more efficient manner, but they are not physically based. The tensor mass model has similar computational properties as the mass spring model, but is physically based, and was the method chosen for the proposed system.

6.1 Tensor Mass System

The tensor mass system, as described by Cotin, et al. [13], breaks down the standard linear elastic finite element formulation into its component stiffness formulation, and can be viewed as a local formulation of the global finite element method. The advantage of this method is in its computational efficiency. To allow for easy modification of the system, the stiffness and damping terms are modeled locally, and not assembled into global matrices. The tensor mass system sums the contribution to each node and edge in the model from all the elements before the simulation commences. This allows topology changes to occur easily and quickly, without having to deal with large sparse matrices. Tetrahedral elements were chosen as the basic element type because of their simplicity, and relative ease of subdividing.

In addition to the standard element by element representation, the method relies on a model that represents the object as a list of nodes and edges. The standard 12 x 12 (4 nodes, each with 3 degrees of freedom) stiffness matrix for each element is calculated, and the 10 distinct 3 x 3 submatrices (6 edges and 4 nodes, each with 3 degrees of freedom) are distributed and linearly summed for each edge and node.

If the finite element system was to be solved on a per element basis, looking at the 12 x 12 stiffness matrix for each element, then the force contribution from each node and edge present would be calculated for each element. This would visit nodes and edges multiple times. For the tensor mass system, the simulation examines each node once and each edge twice, leading to a decrease in the number of calculations required per cycle. For a model of a cube consisting of 6 elements, the equivalent of 96 3 x 3 matrix multiplications are performed for the per element technique; the tensor mass system requires 46, which is equal to the sum of the number of nodes, 8, and twice the number of edges, 19, present.

6.1.1 Element Properties

The standard finite element method can be broken down into 5 steps [58]:

1. The continuous object is broken down by a set of lines or planes;
2. The elements are interconnected by the lines connecting a discrete number of nodal points on the boundary of the elements;
3. Shape functions are selected that transform nodal displacements into general displacements within the interior of the elements;
4. The shape functions uniquely identify the strain state within each element when given the nodal displacements;
5. A collection of forces at the nodal locations that balances out any boundary conditions and external forces is calculated, given the general relationship that:

$$\boldsymbol{f} = \boldsymbol{K}^e \boldsymbol{a}^e + \boldsymbol{f}^a \tag{EQ 7}$$

   where $\boldsymbol{f}$ is the nodal force, $\boldsymbol{K}^e$ is the element stiffness matrix, $\boldsymbol{a}^e$ is the displacement vector, and $\boldsymbol{f}^a$ is the applied external force vector.

The tetrahedral element shape and nodal numbering for this model are shown in Figure 27. The tetrahedral element consists of four vertices, numbered 0 through 3. They are ordered such that if you apply the right hand rule to the first three vertices, the resultant vector points toward the fourth vertex.

The general equation for the stiffness matrix, $\boldsymbol{K}^e$, is dependent on material properties (represented by the matrix $\boldsymbol{D}$) and the shape functions of the element (represented by matrix $\boldsymbol{B}$) given the following equation:

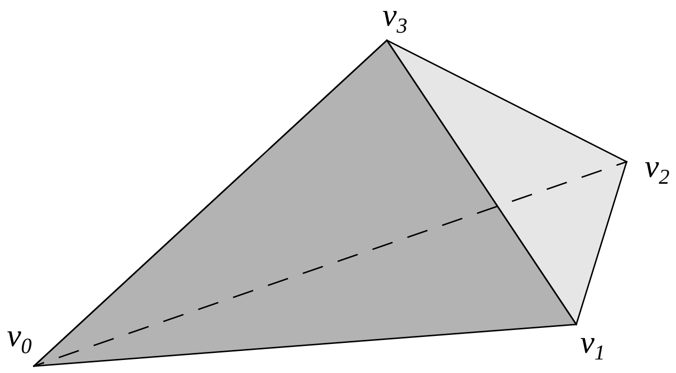

FIGURE 27. Tetrahedral element shape and nodal ordering.

$$\boldsymbol{K}^e = \int_{V^e} \boldsymbol{B}^T \boldsymbol{D}\boldsymbol{B}\, d(Vol) \qquad \text{(EQ 8)}$$

Given an isotropic solid, the equation for the stiffness matrix simplifies to:

$$\boldsymbol{K}^e = V^e \boldsymbol{B}^T \boldsymbol{D}\boldsymbol{B} \qquad \text{(EQ 9)}$$

Due to the nature of the linear elastic model, realistic deformations are limited to approximately 10% strain levels. If deformations grow past this 10% limit, the validity of the results decreases. This leads to limitations on what the linear elastic model can realistically simulate.

6.1.2 Nodal and Edge Properties

Cotin, et al breaks the element stiffness matrix for a tetrahedron down into its nodal and edge components using the following equations:

$$\boldsymbol{K}^e_{ij} = \frac{1}{36V^e}\left(\lambda \boldsymbol{M}_j \boldsymbol{M}_i^T + \mu \boldsymbol{M}_i \boldsymbol{M}_j^T + \mu(\boldsymbol{M}_i \boldsymbol{M}_j)\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\right) \qquad \text{(EQ 10)}$$

The stiffness matrix above depends on the Lamé material coefficients, λ and μ, and the geometry of the matrix, represented by the $\boldsymbol{M}$ vectors. The *i* and *j* indices represent the node or edge to which the stiffness matrix belongs. If the index is repeated, as in $\boldsymbol{K}^e_{ii}$, then the stiffness relates to the force felt by the node due to its own displacement from its home position. If the index is not repeated, as in $\boldsymbol{K}^e_{ij}$, then the stiffness relates to the force felt by the node *i* due to the displacement of node *j* from node *j*'s home position; this can be viewed as an edge effect since it occurs between two nodes. Also, due to symmetry, $\boldsymbol{K}^e_{ji}$ is the equal to the transpose of $\boldsymbol{K}^e_{ij}$.

The ***M*** vectors are defined as:

$$\boldsymbol{M}_j = (\boldsymbol{P}^0_{j+1} \wedge \boldsymbol{P}^0_{j+2} + \boldsymbol{P}^0_{j+2} \wedge \boldsymbol{P}^0_{j+3} + \boldsymbol{P}^0_{j+3} \wedge \boldsymbol{P}^0_{j+1})l$$
$$l = \begin{cases} 1 & j = 0, 2 \\ -1 & j = 1, 3 \end{cases} \qquad \text{(EQ 11)}$$

where $\boldsymbol{P}^o_i$ is the home position of node *i*. These vectors point away from the center of the tetrahedron, toward the exterior, and their magnitude is equal to twice the area of face *j*, which is the face opposite vertex *j*, with vertex numbering as shown in Figure 27.

To calculate the internal elastic force acting on a node *i*, the contributions from all the tetrahedra that node *i* belongs to are summed:

$$\boldsymbol{f}_i = \boldsymbol{K}_{ii}\overrightarrow{P^o_i P_i} + \sum_{j \in N(P_i)} \boldsymbol{K}_{ij}\overrightarrow{P^o_j P_j} \qquad \text{(EQ 12)}$$

where $\boldsymbol{f}_i$ is the nodal force, $\boldsymbol{K}_{ii}$ is the sum of the $\boldsymbol{K}^e_{ii}$ tensors associated with all the tetrahedra that node *i* belongs to; the tensor $\boldsymbol{K}_{ij}$ is the sum of the $\boldsymbol{K}^e_{ij}$ tensors associated with the edge from node *i* to node *j*, $\overrightarrow{P^o_i P_i}$ is the displacement vector of node *i*, and $N(P_i)$ is a list of all nodal neighbors of the node *i*.

Figure 28 shows a simple two element, tetrahedral mesh. For example, the stiffness matrix for node n_2, $\boldsymbol{K}_{22}$, is the sum of $\boldsymbol{K}^e_{22}$ from the element on the left and the element on the right, because node n_2 belongs to both elements. $\boldsymbol{K}_{11}$ is equal to $\boldsymbol{K}^e_{11}$ because node n_1 belongs to only one element. Similarly, $\boldsymbol{K}_{23}$ is the sum of $\boldsymbol{K}^e_{23}$ from both elements, while $\boldsymbol{K}_{12}$ is equal to $\boldsymbol{K}^e_{12}$ from the element on the left. Also, while the vertices are numbered from 1 to 5 in this model, within each element the nodal numbering is the same as in Figure 27. In this manner, the stiffness matrix for each edge and node encodes all the contributions from the elements that the edge or node is associated with.

6.2 Position Integration

Once the forces acting on the nodes of the model are determined, a method for calculating the current position of the nodes is needed. We looked at three different explicit solvers: first-order Euler integration, fixed-time step fourth-order Runge-Kutta, and three different formulations of the Verlet algorithms. We compared these solvers on the basis of computational load and maximum time step while still

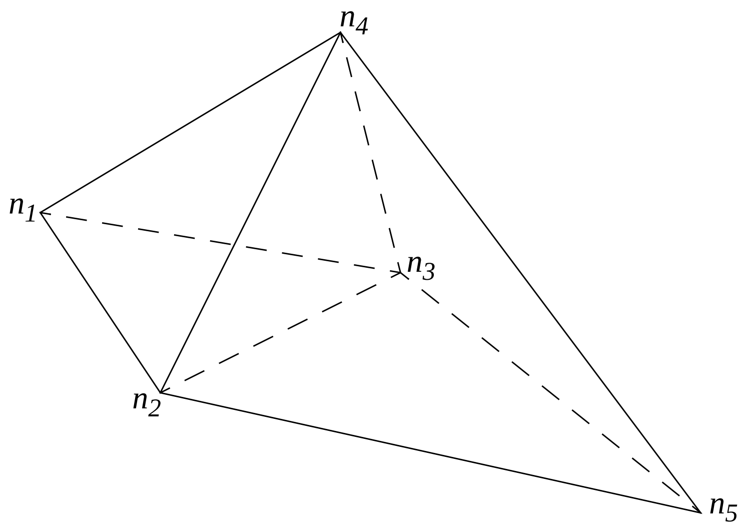

FIGURE 28. Summing of stiffness matrices.

maintaining stability. Due to the fact that the finite element model consists of a stiff set of ordinary differential equations, stability was the main concern with respect to choosing a solver. An explicit solver was selected to avoid having to solve a system of algebraic equations at every time step, which would have required setting up and inverting a large sparse matrix. Also, note that stability is the driving criteria, and not accuracy. Accuracy is important, but for the type of simulation described, and the interactivity of it, greater stability and computational efficiency is of more significance than greater accuracy. Computational efficiency affects synchronicity, the need for the time step to be equal the computational time per cycle, insuring that simulation time matches real time.

6.2.1 Nodal Dynamics

The position, velocity, and acceleration of the nodes is governed by standard Newtonian mechanics, using the following basic equation:

$$\boldsymbol{N}\frac{d^2}{dt^2}\vec{\boldsymbol{x}}(t) + \boldsymbol{C}\frac{d}{dt}\vec{\boldsymbol{x}}(t) + \boldsymbol{K}\vec{\boldsymbol{x}}(t) = \boldsymbol{f}^a \qquad \text{(EQ 13)}$$

where $\boldsymbol{N}$ is the mass matrix for the mesh, $\boldsymbol{C}$ is the damping matrix, $\boldsymbol{K}$ is the overall representation of the stiffness of the mesh, $\vec{\boldsymbol{x}}(t)$ is the vector of current positions of the nodes, and $\boldsymbol{f}^a$ is the vector of applied external forces, such as gravity and pushing forces from the user, acting on the nodes. These matrices are the global equivalents of the local matrices actually used in the tensor mass system. The calculation of the state of individual nodes is done on a per node basis, using scalar values for nodal mass, and Equation 12 to determine the force representative of the $\boldsymbol{f}^i = \boldsymbol{K}\vec{\boldsymbol{x}}(t)$ term. Raeligh damping is used to gernate ***C***, and α and β are chosen empirically so as to damp out oscillations in a reasonable period of time. The mass term for each node is proportional to the volume of the elements it belongs to. The mass for each node is determined with the equation:

$$m_i = \sum_{j \in E(i)} \frac{1}{4} \rho_j V_j^e \qquad \text{(EQ 14)}$$

where m_i is the mass of node *i*, *E(i)* is the set of elements that node *i* belongs to, ρ_j and V_j^e are the density and volume of element *j*, respectively.

This second order ordinary differential equation is solved using the explicit solvers described below. When rewritten on a per-node basis to show the acceleration acting on each node, Equation 13 looks like:

$$\frac{d^2}{dt^2}\vec{\boldsymbol{x}}_i(t) = \frac{1}{m_i}\boldsymbol{f}_i^a - \frac{1}{m_i}\boldsymbol{f}_i^i - \frac{1}{m_i}\boldsymbol{f}_i^d - \frac{c}{m_i}\frac{d}{dt}\vec{\boldsymbol{x}}_i(t) \qquad \text{(EQ 15)}$$

where $\vec{\boldsymbol{x}}_i(t)$ is the current position of node *i*, $\boldsymbol{f}_i^i$ and $\boldsymbol{f}_i^a$ are the internal and applied external forces acting on node *i*, $\boldsymbol{f}_i^d$ is the Raleigh damping force acting on node *i*, and *c* is the global scalar damping term, which models damping between the object and its environment.

Figure 29 demonstrates an example of a model of a simple cubic object, where each cube consists of 6 tetrahedra, randomly colored. The left image is of the model with no external forces acting and no displacements. The right image is the model deformed due to a gravitational force, with the model anchored at the top.

6.2.2 Euler Integration

First order Euler integration was implemented as a baseline numerical integration technique to verify the viability of the model, and as a comparison for the other two integration methods tested. The second order differential equation of the state of the nodes, Equation 15, was implemented as a system of two first order equations:

$$\begin{aligned} \boldsymbol{x}(t+h) &\approx x(t) + h\dot{\boldsymbol{x}}(t) \\ \dot{\boldsymbol{x}}(t+h) &\approx \dot{x}(t) + h\ddot{\boldsymbol{x}}(t) \end{aligned} \qquad \text{(EQ 16)}$$

While this equation did generate a solution for the current state of the model, it is limited both by its accuracy, which was not a large concern, and more importantly by the need for a very small time step to insure stability.

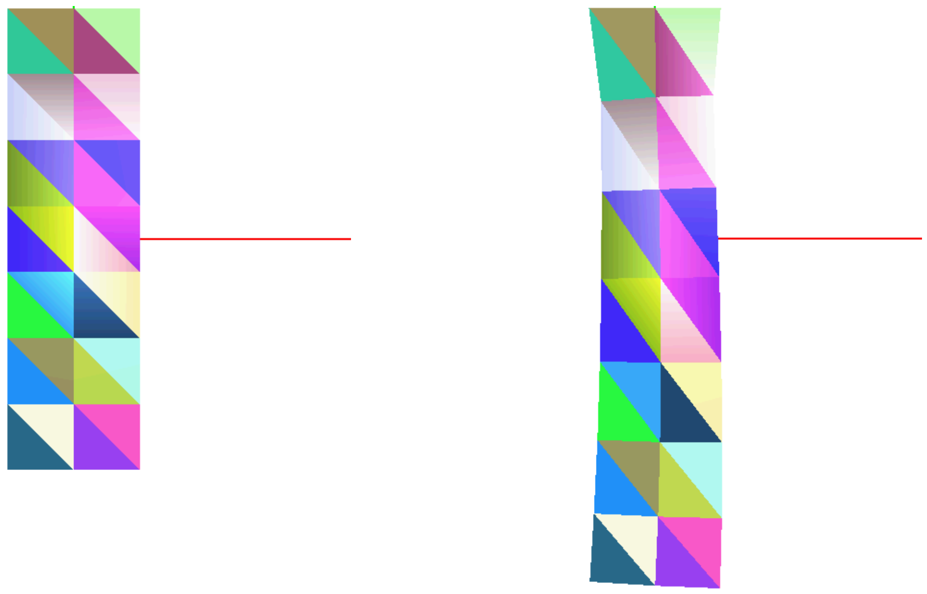

FIGURE 29. Tensor mass system deformation example.

6.2.3 Runge-Kutta Integration

Due to the small time steps required to insure stability with Euler integration, a more accurate and stable method was implemented. Fourth-order Runge-Kutta was chosen for its large increase in accuracy for a given time step, thereby allowing us to greatly increase the time step while maintaining stability and similar levels of accuracy.

The equations were set up in a similar fashion as before, as two sets of first order equations, and then solved with the fourth-order Runge-Kutta algorithm:

$$\begin{aligned} \boldsymbol{x}(t+h) &= \boldsymbol{x}(t) + h\boldsymbol{y}(t) & \boldsymbol{y}(t) &= \dot{\boldsymbol{x}}(t) \\ \boldsymbol{y}(t+h) &= \boldsymbol{y}(t) + hf(\boldsymbol{x}_t, \boldsymbol{y}_t) \end{aligned} \qquad \text{(EQ 17)}$$

where $f(\boldsymbol{x}_t, \boldsymbol{y}_t)$ is the function which generates the acceleration acting on the node when it has position $\boldsymbol{x}_t$ and velocity $\boldsymbol{y}_t$, as in Equation 15. This method is written as:

$$\boldsymbol{x}(t+h) = \boldsymbol{x}(t) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad \text{(EQ 18)}$$

where

$$\begin{aligned} k_1 &= f(t, \boldsymbol{x}_t) \\ k_2 &= f\left(t + \frac{1}{2}h, \boldsymbol{x}_t + \frac{1}{2}hk_1\right) \\ k_3 &= f\left(t + \frac{1}{2}h, \boldsymbol{x}_t + \frac{1}{2}hk_2\right) \\ k_4 &= f(t + h, \boldsymbol{x}_t + hk_3) \end{aligned} \quad \text{(EQ 19)}$$

The implementation of this method stacked the position and velocity vectors on top of each other, to generate one large vector that was passed to the Runge-Kutta routine written for Numerical Recipes in C [39]. That is, the two equations in Equation 17, for position and velocity, were solved simultaneously. While this method of integration did generate results that were much more stable than those produced by the Euler method, the simulation was still not quite fast enough.

6.2.4 Verlet Integration

The Verlet integration methods are popular in the molecular dynamics world, where molecules are modeled as point masses that behave strictly according to Newtonian mechanics [56]. Due to that similarity to the lumped mass model employed in the soft tissue simulation, the Verlet method was tried here. There are three popular forms of the Verlet algorithm: the basic Verlet, a leap-frog technique, and the Velocity Verlet algorithm. All three were tested and found to be quite comparable in terms of model stability, and more computationally efficient in this application than the other methods.

Basic Verlet Algorithm

The basic Verlet algorithm is simple and robust, and is the sum of two Taylor expansions around the current time step, which are summed, and then rearranged to provide the position at time *t+h*:

$$\begin{aligned} \boldsymbol{x}(t+h) &= \boldsymbol{x}(t) + \boldsymbol{v}(t)h + \frac{1}{2}\frac{\boldsymbol{f}^t(t)}{m}h^2 + O(h^3) \\ \boldsymbol{x}(t-h) &= \boldsymbol{x}(t) - \boldsymbol{v}(t)h + \frac{1}{2}\frac{\boldsymbol{f}^t(t)}{m}h^2 - O(h^3) \end{aligned} \quad \text{(EQ 20)}$$

$$\boldsymbol{x}(t+h) = 2\boldsymbol{x}(t) - \boldsymbol{x}(t-h) + \frac{\boldsymbol{f}^t(t)}{m}h^2 + O(h^4) \quad \text{(EQ 21)}$$

where $\boldsymbol{f}^t(t)$ is the sum of all the forces acting on a node at time *t*.

If velocity terms are needed, they can they be calculated with a difference term, although note that at time *t+h*, the velocity term to be calculated is that for time *t*:

$$\boldsymbol{v}(t) = \frac{\boldsymbol{x}(t+h) - \boldsymbol{x}(t-h)}{2h} + O(h^2) \qquad \text{(EQ 22)}$$

Since the velocity term is used in the calculation of forces for the nodes, calculating the velocity one time step behind is not ideal, so the Velocity Verlet algorithm was investigated.

Velocity Verlet Algorithm

The Velocity Verlet algorithm [50] calculates the velocities of the nodes at the new time step, and therefore improves upon the basic algorithm. The main drawback is that it is more expensive computationally. The basic Velocity Verlet algorithm requires four steps:

1. Calculate the midpoint velocity:

$$\boldsymbol{v}(t+h/2) = \boldsymbol{v}(t) + \frac{1}{2}\frac{\boldsymbol{f}^t(t)}{m}h + O(h^3) \qquad \text{(EQ 23)}$$

2. Calculate the new position based on the midpoint velocity:

$$\boldsymbol{x}(t+h) = \boldsymbol{x}(t) + \boldsymbol{v}(t+h/2)h + O(h^3) \qquad \text{(EQ 24)}$$

3. Calculate the new forces acting on the nodes, using the midpoint velocity and new positions:

$$\boldsymbol{f}^t(t+h) = \boldsymbol{f}_i^a(t+h) - \boldsymbol{f}_i^i(t+h) - c\boldsymbol{v}(t+h/2) + O(h^3) \qquad \text{(EQ 25)}$$

4. Calculate the final new velocity:

$$\boldsymbol{v}(t+h) = \boldsymbol{v}(t+h/2) + \frac{1}{2}\frac{\boldsymbol{f}^t(t+h)}{m}h + O(h^3) \qquad \text{(EQ 26)}$$

This form gives the most complete solution, but is slightly slower due to the fact that it has to cycle through the list of nodes twice, once before updating the total forces acting on the nodes, and once after. Depending on the memory architecture of the machine and the software, this can impact the computation time.

Verlet Leapfrog Algorithm

The Verlet Leapfrog algorithm explicitly calculates velocity, like the Velocity Verlet method, but the velocity it calculates is at the midpoint of the time step. The significant difference for us is that it only cycles through the list of nodes once, not twice, and therefore can run more quickly. While having the velocity calculated at the midpoint is not as accurate and desirable as determining it at the end of the time step, in practice, it did not prove to be a problem.

The algorithm first calculates the midpoint velocity:

$$\boldsymbol{v}(t + h/2) = \boldsymbol{v}(t - h/2) + \frac{\boldsymbol{f}^t(t)}{m}h + O(h^3) \qquad \text{(EQ 27)}$$

Then, the new position is calculated using the midpoint velocity:

$$\boldsymbol{x}(t + h) = \boldsymbol{x}(t) + \boldsymbol{v}(t + h/2)h + O(h^4) \qquad \text{(EQ 28)}$$

If the velocity term at time t is desired, it can be calculated with the difference equation:

$$\boldsymbol{v}(t) = \frac{\boldsymbol{v}(t + h/2) - \boldsymbol{v}(t - h/2)}{2} + O(h^3) \qquad \text{(EQ 29)}$$

This method proved to be the fastest of the three methods described in this section, while stability numbers were similar for all three.

6.2.5 Element Stability

Two of the main requirements for position integration within this type of surgical simulator, where the user is applying a highly variable amount of force, is efficiency and stability. Efficiency to reduce the computational load, and therefore increase either the number of elements that can be modeled or the complexity of the interaction routines between the model and the tools that the user wields. Stability is important so that the models behave appropriately, and do not oscillate and shoot off to infinity, thereby either generating large arbitrary forces or crashing the simulator.

The theoretical limits on time steps for the different routines do have interest, but only in an academic sense. Due to the complexity of the model, the variability of element size, and implementation issues, actual, empirical limits on the time steps proved to be of more use than the theoretical limits.

For instance, in implementing a cubical model, the theoretical limit on the time step for these explicit methods is proportional to the ratio of edge length to maximum wave velocity in the material:

$$h_{max} \leq \frac{l_{min}}{v_{max}} \qquad v_{max} = \sqrt{\frac{\lambda + 2\mu}{\rho}} \tag{EQ 30}$$

For one model, this results in a maximum time step of 0.005sec. Empirically, the maximum time step found for Euler integration was 0.00015sec, more than 30 times less. This may be due to the fact that each node is not moving independently, but is interconnected with many neighboring nodes in the model, thereby affecting the calculations of a representative stiffness matrix.

In addition to determining a minimum stable time step for a given integration method, we also need to determine the inverse, the minimum edge length, or threshold length, to maintain stability, given material properties and the actual time step being used, for use in our cutting routines.

This was also done on an empirical basis, given the fact that the minimum edge length is proportional to the maximum velocity that a wave can travel through a linear elastic finite element model:

$$l_t \geq c h v_{max} \tag{EQ 31}$$

where l_t is the minimum threshold length, h is the time step of the simulation, and v_{max} is determined using Equation 30. c is an empirical constant, determined through testing.

The testing method to determine c is straightforward. Given a particular model, the minimum edge length was found. Assuming that this edge length is l_{min}, and knowing v_{max}, we increased the time step, h, until the model became unstable. We then reduced this value to give a slight amount of cushion for stability, and set the value of c:

$$c = \frac{l_{min}}{h_{max}} \frac{1}{v_{max}} \tag{EQ 32}$$

Subsequently, during cutting, the element edge lengths and the height of the vertices are compared to l_t, from Equation 31, to determine whether the resultant element will be unstable or not.

6.2.6 Computational Efficiency

To determine which integration routine would work best overall, we calculated the trade-off between numerical stability and computational efficiency by examining the maximum time step that could be utilized without causing instability. We then divided this maximum time step by the actual time spent performing the computation to determine how close to, or how much better than, real-time the calculations could run.

A value of 1.0 would signify that the fastest this model could run would be real-time, while a value of 5.0 would be five times as fast as real-time, and 0.5 would be half as fast as real-time. These experiments were run on an SGI O2 with an 180MHz R5000 processor, with a model size of 144 elements, 63 vertices, and 262 edges.

Integration Type:	Max. Time Step	Calculation Time	Ratio to Real-Time
Euler	0.00015	0.0006	0.25
Runge-Kutta	0.0113	0.0027	4.19
Velocity Verlet	0.0074	0.0007	10.57
Verlet Leapfrog	0.0074	0.00067	11.04

TABLE 4. Computational efficiency vs. numerical stability for integration.

As can be seen in Table 4, the Verlet algorithms perform much better than the Euler and Runge-Kutta integration methods. Also, due to its better computational efficiency, the Leapfrog Verlet technique is a little more than 4% better than the Velocity Verlet method, and is the method used in our simulator.

Chapter 7

Object Interaction

While cutting tissue is the main method surgeons use to modify and effect changes in patients, it is not the only way in which they interact with the patient. Palpation is used quite often in determining possible pathologies in tissue and the location of particular features. Forceps are used for grasping tissue, either to hold it out of the way or to remove tissue fragments from the surgical field. Lastly, puncture of tissue is not exactly similar to palpation and grabbing, but needle sticks are used often, and suturing can be viewed as a series of tissue punctures, after which the punctured tissue is affixed to its neighbor.

7.1 Palpation of Model

Palpation can be viewed as a method of applying external forces to an object, either using the fingers directly or through some instrument. A surgeon would use her fingers to palpate tissue, for example, when feeling for a lump in underlying tissue, while she might use an instrument to hold tissue back or to palpate tissue while performing minimally invasive procedures.

To simplify the modeling of interaction with the soft tissue model, the fingertip can be represented by an implicit sphere, which roughly mimics the fingertip shape. Instruments are represented as finite length, implicit cylinders, which can push on objects both along their length and with their endcaps. Implicit shape models, where the model is described completely by a mathematical equation instead of a set of surface triangles, are used to simplify the intersection detection and calculations. With these two shapes, a simple and powerful paradigm for interaction was generated.

Two methods of interaction were implemented using these implicit shapes. With the sphere model, the first technique described generates interaction and intersections between the nodes themselves and the sphere, while the second technique intersects the surface triangles. For the cylinder only interactions between the cylinder and the triangles of the surface were modeled. Interacting directly with the surface, instead of with the individual nodes, proved to be a more reliable and realistic interaction modality.

7.1.1 Implicit Sphere - Node Interaction

The first method tested for interacting with the soft tissue model was to have the implicit sphere model intersecting the surface nodes. This method is fast and efficient, and generates smooth consistent forces.

This method generates external forces on the nodes using a penalty based method. In the penalty based method, the magnitude and direction of the force vector acting on a node, and acting back on the user, is dependent on how far into the sphere the node penetrates. The routine cycles through the list of nodes that are on the surface of the model, and checks to see which nodes are within the sphere centered at the user's current position. It stores pointers of all the nodes which fall within the diameter of the sphere, and then cycles through that list, and generates the external forces being applied to the intersected nodes. The external force vectors are also summed, so as to display the opposite force back to the user, so that she feels the effects of the deformation she is causing.

This method is demonstrated in Figure 30. As can be seen, two of the surface nodes intersect the implicit sphere, and external forces are then applied to the nodes. A force is applied to the sphere, which is equal and opposite of the sum of the forces on the nodes. This force is displayed back to the user through the haptic interface.

The forces applied to the nodes are:

$$\begin{aligned} |\boldsymbol{x}_i(t) - \boldsymbol{x}_s(t)| < r: &\qquad \boldsymbol{f}_i^e = k(r - |\boldsymbol{x}_i(t) - \boldsymbol{x}_s(t)|)\frac{\boldsymbol{x}_i(t) - \boldsymbol{x}_s(t)}{|\boldsymbol{x}_i(t) - \boldsymbol{x}_s(t)|} \\ |\boldsymbol{x}_i(t) - \boldsymbol{x}_s(t)| \geq r: &\qquad \boldsymbol{f}_i^e = 0 \end{aligned} \tag{EQ 33}$$

where $\boldsymbol{x}_i(t)$ is the current position of node *i*, $\boldsymbol{x}_s(t)$ is the current position of the implicit sphere, *r* is the radius of the implicit sphere, $\boldsymbol{f}_i^e$ is the external force being applied to node *i*, and *k* is the stiffness term for the implicit sphere, which determines how hard the sphere pushes on the nodes. Note that the internal structure of the model generates the internal forces that cause the nodes to push back on the implicit sphere.

While this technique can work well in most cases, there are clear cases where it fails. The most obvious case is when the diameter of the sphere is smaller than the average

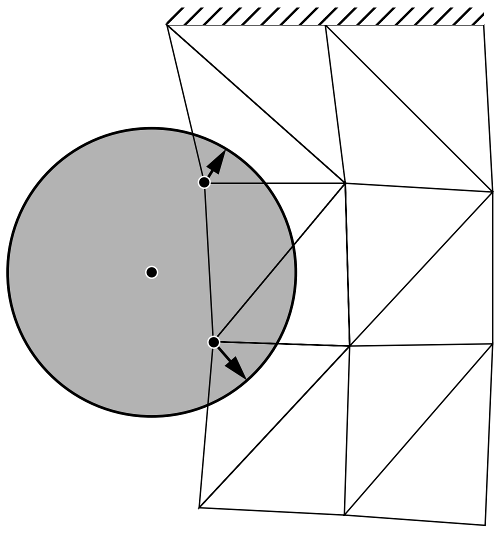

FIGURE 30. Cross-section of implicit sphere interacting with the nodes of a model.

distance between nodes. In this case, the sphere can just slip right between nodes and penetrate within the object without generating any forces. A no less significant, but more insidious difficulty arises when the sphere is on the order of the average distance between nodes. In this case, the sphere will generate forces at first, but as it pushes into the model, it effectively spreads the nodes apart and makes a hole in the surface for itself. The force needed to generate this hole is smaller than might be expected, because the directions of the forces being applied to the nodes tend to cancel each other out.

Say, as shown in Figure 31, that there are four nodes arranged in a square on the surface, or, in two dimensions, 2 nodes connected by an edge. The implicit sphere starts to interact with those nodes, and after a little bit of time, falls slightly into the crater at the center of the four nodes. In this case, all the force vectors are pointing to the center of the sphere, and the components of the force vectors parallel to the surface cancel themselves out. Therefore, the total force felt by the user can be much less that the sum of the magnitudes of the component forces. Then, as the user pushes further into the object, the force vectors acting on the nodes become more parallel to the surface of the model, and the total force magnitude becomes even smaller. This continues until the implicit sphere has penetrated to the interior of the model, with the total force being displayed back to the user becoming smaller and smaller as the penetration distance increases.

Because of this problem, it was decided that the interaction routines should deal with the surface of the model, instead of the nodes, to generate a more realistic interaction modality.

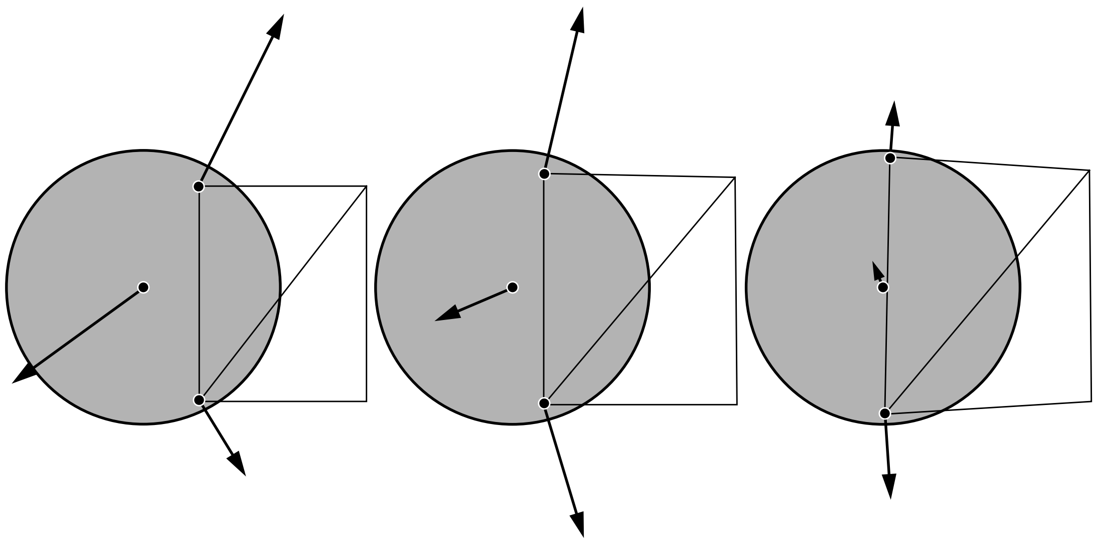

FIGURE 31. Example of hole generated by implicit sphere, in cross-section.

7.1.2 Implicit Sphere - Surface Interaction

Interacting with the surface triangles instead of the surface nodes is much more realistic and analogous to real palpations. In this manner, the user interacts with what she sees, and not some underlying, seemingly arbitrary, grid of points. The closest example to this method found in the literature is in Berkelman, et al., [4] where an impulse based method is used between two surfaces to enforce that no interpenetration occur between rigid models. But that method was not applied to deformable models.

This method uses a similar penalty based method as that described in the previous section, except that instead of using the distance from the node position to the surface of the implicit sphere, we calculate an approximation of the volume of intersection, and use that volume as the penalty term. This volume of intersection is shown in Figure 32. We are looking back from the front, and seeing the surface area that has been intersected by the implicit sphere. The volume of intersection is then projected back between the shaded area and the boundary of the sphere.

This routine cycles through the surface triangles of the object, in the same manner as the sphere-node routine, and first checks to see which triangles are intersecting the sphere. It first does a simple test to see if the sphere intersects the unbounded plane that the triangle defines. The test calculates the distance between the sphere center and the plane, and if the distance between them is greater than the sphere radius, then no intersection occurs. If the sphere does not intersect that unbounded plane, then it can not intersect the triangle itself, and no intersection occurs between the implicit sphere

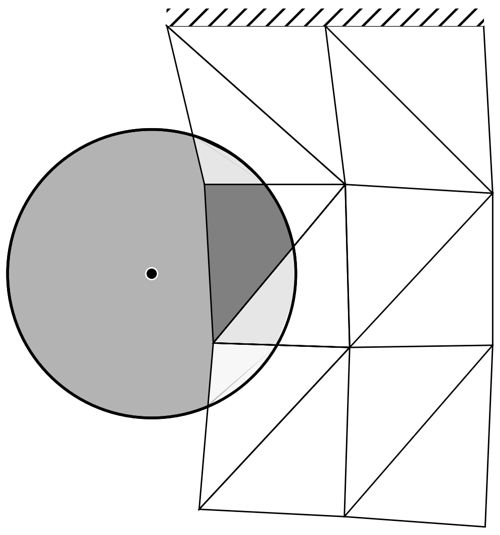

FIGURE 32. Cross-section of implicit sphere interacting with surface, showing the volume of intersection

and that triangle. If there is an intersection between the implicit sphere and the unbounded plane, we then check to see if there is an intersection with the triangle itself. The first part of this extended test is to see if any of the vertices are within the sphere itself. If so, then an intersection has occurred, the triangle is added to a list of intersections with the sphere, and the next triangle is tested.

If no vertices are within the interior of the sphere, then an exact intersection test is done. This test calculates the actual distance between the center of the sphere and the triangle. The closest point on the triangle could be within the triangle, or on any of its vertices or edges. We adapted the algorithm demonstrated by [16]. This algorithm looks at the gradient of the squared distance function between the point in question and the parametric form of the triangle. First, the parametric coordinates are calculated for the point where the gradient is zero. If these coordinates are within the interior triangle, then we use these coordinates. If they are outside the interior, then based on which region they are in, as shown in Figure 33, we find the parametric coordinates of the closest point, either on the closest vertex or edge. We then calculate the distance to the center of the sphere. If this distance is less than the radius of the sphere, then there is an intersection, and this triangle is added to the list of current intersections.

In testing to find the triangles which are currently intersected, the routine also uses the previous list of intersected triangles to speed up the calculations. The first time we test for intersections, we test all the surface triangles to insure that no intersections are missed. These intersected triangles are stored, and the next time through the routine, these current intersections are used as the basis for determining which new triangles are intersected. Instead of testing all the surface triangles, only the previously

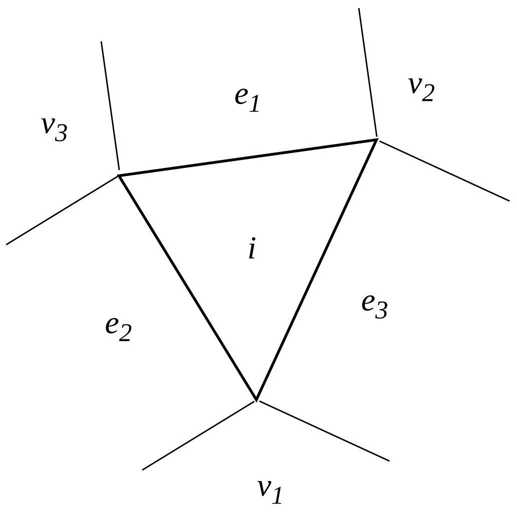

FIGURE 33. Closest point on a triangle.

intersected triangles and their neighbors are tested. This greatly reduces the number of intersection tests performed. This speedup relies on continuity of the model, and the fact that the models simulated are not convex in such a way that the user could contact two triangles that are not neighbors at the same time. In testing, this has been true and no problems attributable to this speedup have arisen.

Once we have a list of the triangles that are currently intersected, we then determine the volume of intersection and forces to apply for each one. The first step is to determine the fraction of the triangle that is within the volume of the sphere, and the centroid of that intersected area.

First, we calculate the center and radius of the intersection between the sphere and the plane that the triangle defines. In this manner, the intersection is now between a circle and a triangle in two dimensions, as shown in Figure 34. We then determine the topology of the intersected shape, composed of a set of segments of EDGES and ARCS. The different types of intersections are shown in Figure 35. The possible cases are: the projected circle is fully within the triangle; the triangle is fully within the projected circle; one edge of the triangle is intersected by the projected circle; and two or more edges are within the projected circle.

In the first case, where the circle of intersection is fully within the triangle, then the area and centroid of intersection are just those of the circle itself. In the second case, the area and centroid of intersection are just those of the triangle. If there are two segments in the intersection, as in the third case, then the area of intersection is part of a lopped off circle.

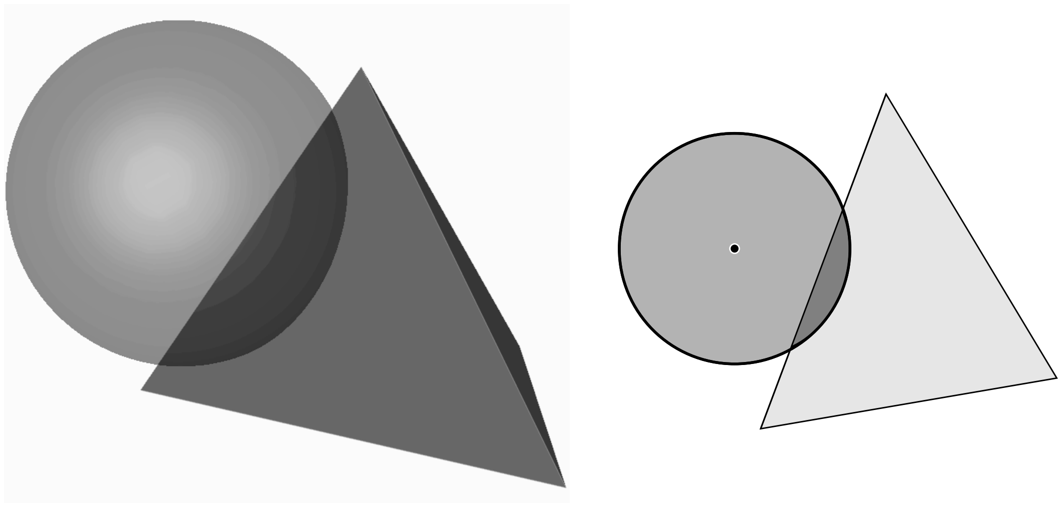

FIGURE 34. Sphere intersected with surface triangle, and projection of the sphere onto the plane of the triangle.

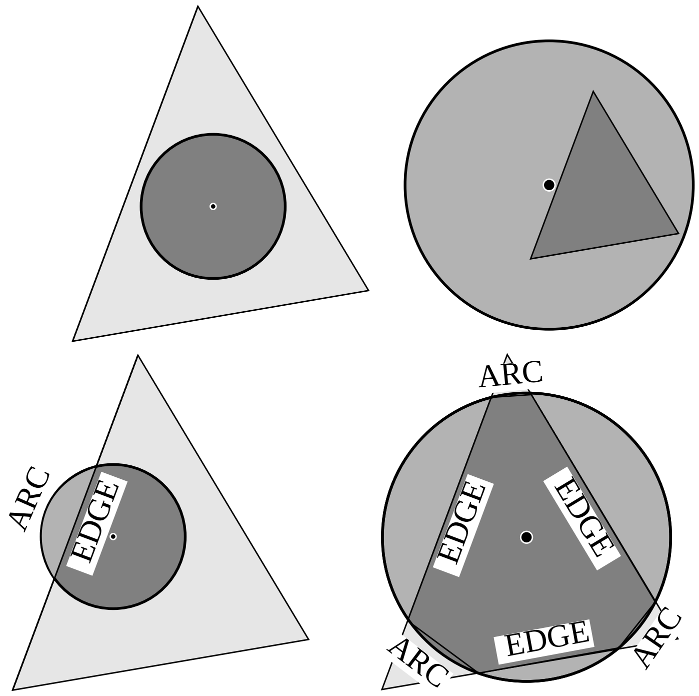

FIGURE 35. Different types of intersections between projected circle and triangle.

For this lopped off circle portion, we first determine whether or not we want the large portion of the lopped circle, or the small side, based on whether or not the center of the intersection circle is within the interior of the triangle or not. Next, we calculate the area of the lopped portion of the circle:

$$\begin{aligned} A_s &= r^2(\theta - \cos\theta\sin\theta) \\ A_l &= \pi r^2 - A_s \end{aligned} \qquad \text{(EQ 34)}$$

where A_s is the area of the small side of the lopped circle and A_l is the area of the large side, r is the radius of the circle, and θ is half of the angle of the pie shaped segment from the center of the circle to the two points where the lopping edge intersected the circle. The shaped of the lopped off portion, and the other side of the circle, is shown in Figure 36.

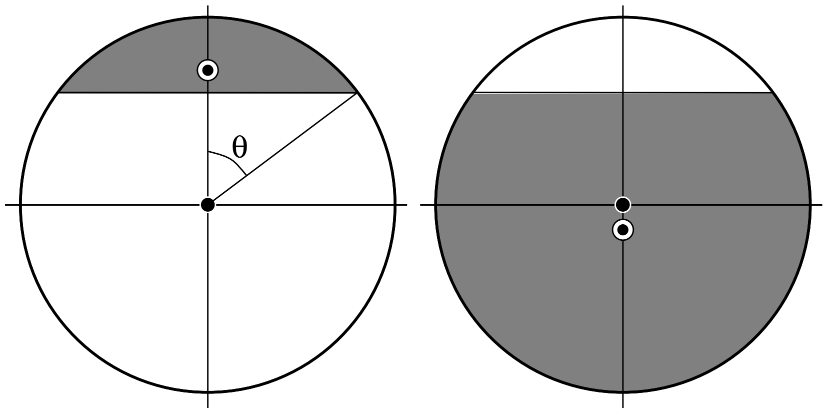

FIGURE 36. Area of lopped circle.

Lastly, we calculate the centroid of the lopped portion of the circle. The *x*-component is zero, while the equation for that *y*-component of the centroid is:

$$\begin{aligned} y_{cs} &= \frac{\frac{2}{3}(r\sin\theta)^3}{A_s} \\ y_{cl} &= \frac{-\frac{2}{3}(r\sin\theta)^3}{A_l} \end{aligned} \qquad \text{(EQ 35)}$$

where y_{cs} is the *y*-component of the centroid of the lopped section if we are looking at the smaller side of the circle, and y_{cl} is the *y*-component if we are dealing with the large side of the lopping segment.

If there are three or more segments, then there is an interior polygon area, and, possibly, an additional area created by ARC segments. In the case of the triangle fully within the projected circle, then there are no ARC segments. First, we calculate the area of the interior, polygonal section of the intersected triangle, using the method described in [22]. In the same function, we adapted the method in [3] to a three dimensional form to calculate the centroid of the polygon. After the centroid of the interior is calculated, the area and centroids of any ARC segments present are calculated using Equation 34 and Equation 35 and appropriately added to the total area and centroid of the intersected triangle section.

After the area of intersection and the centroid for the intersected triangle are determined, we calculate the distance between the centroid of the triangle and the sphere center. We also find the normal vector of the triangle. Then, the total force applied to the triangle is:

$$\boldsymbol{f}_i^t = kA_t(r - |\boldsymbol{x}_c(t) - \boldsymbol{x}_s(t)|)\hat{\boldsymbol{n}}_t \qquad \text{(EQ 36)}$$

where $\boldsymbol{f}_i^t$ is the total force acting on the triangle, *k* is the gain associated with the implicit sphere, A_t is the total area of the triangle intersected with the implicit sphere, $\boldsymbol{x}_c(t)$ is the centroid of the intersected area, $\boldsymbol{x}_s(t)$ is the position of the sphere center, $\hat{\boldsymbol{n}}_t$ is the unit normal of the intersected triangle, and *r* is the diameter of the implicit sphere. In this equation, the term:

$$A_t(r - |\boldsymbol{x}_c(t) - \boldsymbol{x}_s(t)|) \qquad \text{(EQ 37)}$$

approximates the volume of intersection, with the first term approximating the area of intersection, and the second term representing the depth of intersection. This is scaled by the gain, and the direction of force is perpendicular to the intersected triangle.

Ideally, we would then apply this force directly to the centroid of the intersected area. This is not possible with the lumped masses at the nodes, so we calculate scaling factors such that the total applied force to the triangle is equal to $\boldsymbol{f}_i^t$. The other main requirement is that the total moment acting on the triangle is equivalent to the moment generated by the force $\boldsymbol{f}_i^t$ acting at the centroid $\boldsymbol{x}_c(t)$. This can be represented by the set of equations:

$$\begin{aligned} \boldsymbol{F}_1 + \boldsymbol{F}_2 + \boldsymbol{F}_3 &= \boldsymbol{f}_i^t \\ d_{1x}\boldsymbol{F}_1 + d_{2x}\boldsymbol{F}_2 + d_{3x}\boldsymbol{F}_3 &= 0 \\ d_{1y}\boldsymbol{F}_1 + d_{2y}\boldsymbol{F}_2 + d_{3y}\boldsymbol{F}_3 &= 0 \end{aligned} \qquad \text{(EQ 38)}$$

which can be represented by the matrix equation:

$$\begin{bmatrix} 1 & 1 & 1 \\ d_{1x} & d_{2x} & d_{3x} \\ d_{1y} & d_{2y} & d_{3y} \end{bmatrix} \begin{bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \\ \boldsymbol{F}_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{f}_i^t \\ 0 \\ 0 \end{bmatrix} \qquad \text{(EQ 39)}$$

Inverting this and solving, we get:

$$\begin{bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \\ \boldsymbol{F}_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ d_{1x} & d_{2x} & d_{3x} \\ d_{1y} & d_{2y} & d_{3y} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{f}_i^t \\ 0 \\ 0 \end{bmatrix} \quad \text{(EQ 40)}$$

Lastly, since we only need the first column of the inverted matrix, this simplifies to:

$$\begin{bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \\ \boldsymbol{F}_3 \end{bmatrix} = \frac{\boldsymbol{f}_i^t}{\begin{vmatrix} 1 & 1 & 1 \\ d_{1x} & d_{2x} & d_{3x} \\ d_{1y} & d_{2y} & d_{3y} \end{vmatrix}} \begin{bmatrix} d_{2x}d_{3y} - d_{3x}d_{2y} \\ d_{3x}d_{1y} - d_{1x}d_{3y} \\ d_{1x}d_{2y} - d_{2x}d_{1y} \end{bmatrix} \quad \text{(EQ 41)}$$

where the variables are as in Figure 37, with $\boldsymbol{F}_i$ being the force on node i and $d_{ix,y}$ being the distance along the respective axes between the vertex and the centroid of the intersected area.

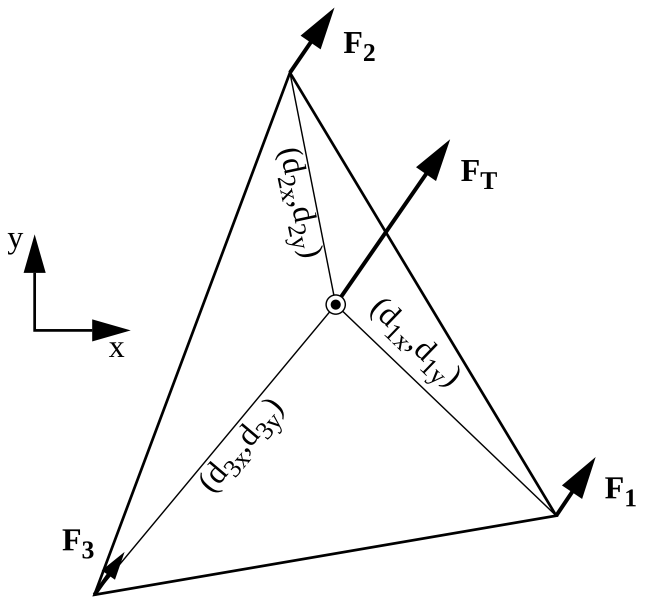

FIGURE 37. Equivalent forces at nodes to force at centroid.

Figure 38 shows a snapshot of a model being deformed due to palpation with an implicit sphere utilizing this method.

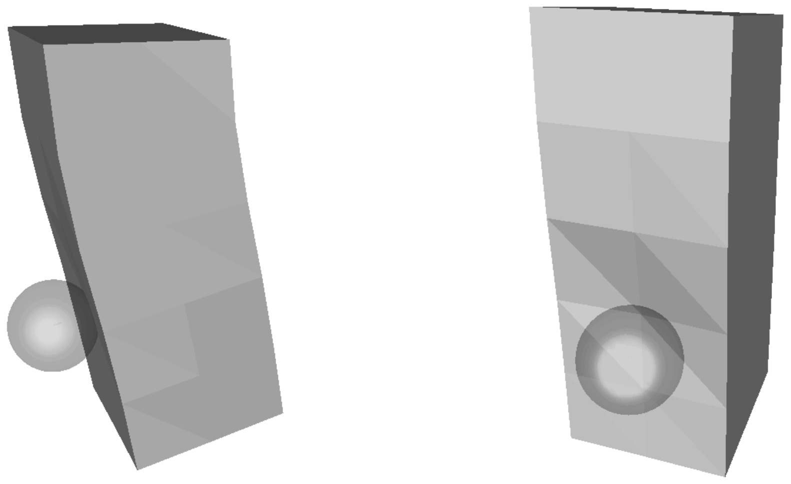

FIGURE 38. Model deformed by an implicit sphere touching the surface.

7.1.3 Implicit Cylinder - Surface Interaction

As mentioned before, an implicit cylinder can model many of the instruments used in surgery. Almost all laporascopic and endoscopic instruments can be modeled by cylinders or a collection of a small number of cylinders. If needed, a whole finger can be modeled as a set of small cylinders, possibly with an implicit sphere on the end. In this section, we will describe the geometric underpinnings of the implicit cylinder interaction.

The basic paradigm for modeling the interaction between the implicit cylinder and the model is the same as for the interaction between the implicit sphere and the model. First, intersections between individual surface triangles and the cylinder are detected. Then, for each triangle, the area and centroid of intersection within each triangle is calculated, a volume of intersection is determined, and then the total force due to that volume is distributed to the three vertices that make up the individual triangle.

The process of determining if the triangle and the cylinder intersect is more complicated than in the case of the sphere. Not only must the body of the cylinder be checked, but it must be truncated due to its length, and the endcaps at the two ends of the finite cylinder must also be checked for intersections.

The first test is a simple test to see if the central axis of the cylinder intersects the triangle. Next, we determine whether or not the vertices of the triangle are in the interior of the cylinder. Then, we see if either of the circles bounding the endcaps intersect the triangle. Lastly, if none of these simple tests determine that an intersection has occurred, we do a complete test to see if the individual edges of the triangle pass through the cylinder. In this test, we first determine whether the closest point between the edge and the axis of the cylinder is within the extent of the infinite cylinder or not. If it is, we then calculate the distance between that closest point and the surface of the cylinder, and the distance along the edge to the two intersection points. Next, we check to see if the edge would actually hit either of the two endcaps of the finite implicit cylinder, instead of either of the calculated intersections along the infinite extent of the cylinder. Lastly, we check to see if the intersection points are truly within the length of the finite edge. In this way, we calculate the true intersections with the cylinder, check to see if the edge would strike an endcap first, and then verify that the intersection points are within the actual length of the edge and the finite extent of the cylinder. This process is demonstrated in Figure 39.

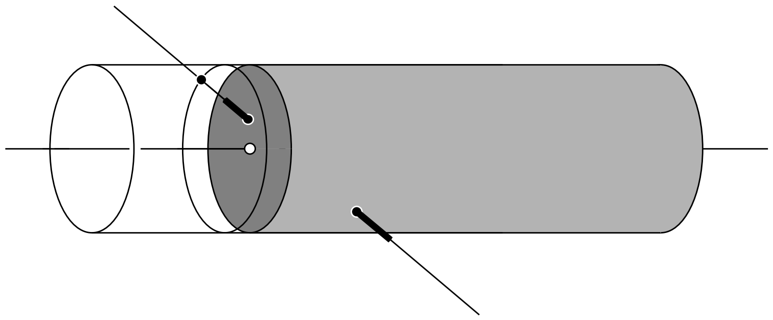

FIGURE 39. Intersection of a finite edge with a finite, implicit cylinder.

After it is determined that a triangle has been intersected by the implicit cylinder, we have to generate the boundary of the area of intersection that has been created. This is generated using the information on whether or not individual vertices are inside the implicit cylinder, whether the endcap boundaries intersected the triangle, and the exact intersection information generated by the tests between the triangle's edges and the cylinder. Each intersection point on the boundary is recorded, and the subsequent type of edge segment determined in the same manner as in Section 7.1.2.

We also determine the amount of projection needed, based on the angle between the implicit cylinder and the triangle normal. This is needed because if the cylinder axis and the triangle normal are not parallel, then the intersection boundary between the two is an ellipse and not a circle. To account for this, the intersection points are projected back to a circle on the cylinder. Then, exactly as in the implicit sphere routines, the centroid and area of intersection are calculated. The centroid is then projected back to the plane of the triangle, while the area of intersection is scaled by:

$$A_c = A_p \frac{1}{\hat{n}_t \cdot \hat{a}_c} \qquad \text{(EQ 42)}$$

where $A_{\mathbf{c}}$ is the area projected back onto the triangle, A_p is the area of the projected intersection, $\hat{\boldsymbol{n}}_t$ is the normal vector of the triangle, and $\hat{\boldsymbol{a}}_c$ is the direction of the axis of the implicit cylinder. Note that the projected area scales with the inverse of the dot product $\hat{\boldsymbol{n}}_t \cdot \hat{\boldsymbol{a}}_c$. As the cylinder axis moves away from being parallel to the normal of the triangle, the major axis of the ellipse of intersection increases, as does the area of intersection. At the limit, when the two axes are perpendicular and the cylinder lies parallel to the triangle's plane, the dot product $\hat{\boldsymbol{n}}_t \cdot \hat{\boldsymbol{a}}_c$ goes to zero and the projected area becomes infinite. In this case, we use the actual intersection area of the triangle.

Once we have the location of the centroid and the area of intersection, we use the same calculations as for the implicit sphere method to determine the total force and to divide it among the three vertices of the triangle. The total force applied to the triangle is:

$$\boldsymbol{f}_i^t = kA_t(r - |\boldsymbol{x}_c(t) - \boldsymbol{x}_s(t)|)\hat{\boldsymbol{n}}_t \qquad \text{(EQ 43)}$$

with the force split up according to:

$$\begin{bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \\ \boldsymbol{F}_3 \end{bmatrix} = \frac{\boldsymbol{f}_i^t}{\begin{vmatrix} 1 & 1 & 1 \\ d_{1x} & d_{2x} & d_{3x} \\ d_{1y} & d_{2y} & d_{3y} \end{vmatrix}} \begin{bmatrix} d_{2x}d_{3y} - d_{3x}d_{2y} \\ d_{3x}d_{1y} - d_{1x}d_{3y} \\ d_{1x}d_{2y} - d_{2x}d_{1y} \end{bmatrix} \qquad \text{(EQ 44)}$$

7.2 Grasping of the Soft Tissue Model

Grasping and pulling on tissue is often used in surgery for many different purposes. Clamping tissue and securing it temporarily to keep tissue out of the surgical field, pulling on a tissue to expose a cut surface, and removing cut tissue from the surgical field all rely on the ability to affix tissue with a tool. Grasping of the soft tissue model within the simulator is built upon the interaction routines described in Section 7.1. A simple tool modeled as an implicit sphere interacts with the object, and when closed, or activated, grasps the object. This tool model could be easily replaced with a more realistic model of forceps represented as two small cylindrical elements. Grasping of the model is demonstrated with three different methods, grasping individual nodes of the model, grasping individual triangles of the model, and grasping of an arbitrary point on the surface of the model. While the first method is an analog of the implicit sphere model interacting with individual nodes, the other two methods more directly correlate to actual grasping. The second method generates forces based not only the position of the grabber, but the orientation of it also. The last method does not generate forces based on the orientation of the grabber, but more accurately relates forces based on grasping the closest point on the model to the grasper when the user closes the tool, and is the method most commonly used within our simulator.

7.2.1 Grasping a Single Vertex

One method of grasping a model is to affix the closest node in the model to the grasping tool. A fixed offset between the current position of the closest node and the grasper is generated to insure that the grasper appears to be holding a point centered under the grasping tool. The node would be moved along with the grasper to generate deformations based on the motion of the tool. Figure 40 shows how this type of grasping would be set up.

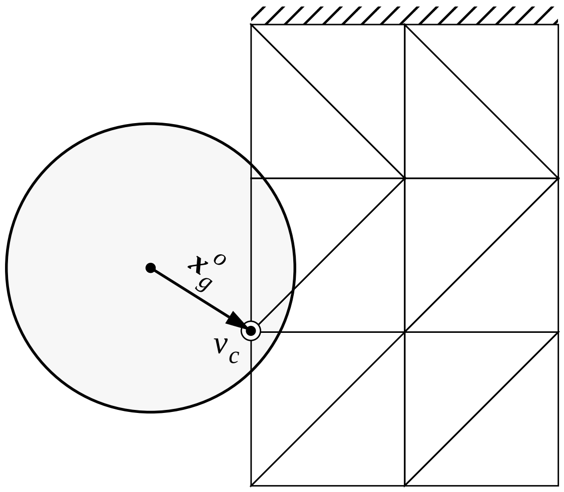

FIGURE 40. Grasping of a node.

The first step with modeling this type of grasper is to determine whether or not there are any nodes close enough to the grasper to be within its working range. If there are, then the closest node is selected as the grasped node, and the current offset between it and the grasper is determined:

$$\boldsymbol{x}_g^o = {}_0^g\boldsymbol{R}^T(\boldsymbol{x}_c(t) - \boldsymbol{x}_g(t)) \qquad \text{(EQ 45)}$$

where $\boldsymbol{x}_g^o$ is the current offset within the grasper reference frame, ${}_0^g\boldsymbol{R}^T$ is the transpose of the rotation matrix between the grasper's reference frame and the inertial frame, $\boldsymbol{x}_c(t)$ is the current position of the closest node, and $\boldsymbol{x}_g(t)$ is the current position of the grasper.

At each time step, then, the current position of the grasped node is set to:

$$\boldsymbol{x}_c(t) = \boldsymbol{x}_g(t) + {}_0^g\boldsymbol{R}\boldsymbol{x}_g^o \qquad \text{(EQ 46)}$$

The main problem with this method is similar to the difficulty with the palpation based on nodal positions, in that there are times when the grasping tool may be touching the surface but not in contact with any of the nodes. In this case, when the user closes the grasper, there will be nothing for it to grasp within its reach, which would be unexpected for the user. We therefore investigated grasping of the closest triangle to the grasper instead of the closest node.

7.2.2 Grasping of a Surface Triangle

Interacting with the surface triangles solves the difficulty associated with grasping of the nodes directly. The user will be grasping the triangle he is currently touching, and will be able to rotate it based on the orientation of the tool, as if the grasper was clamping the triangle in its jaw, instead of just fixing on a point. Figure 41 shows an example of the initial grasp and calculations of offsets for this method of grasping.

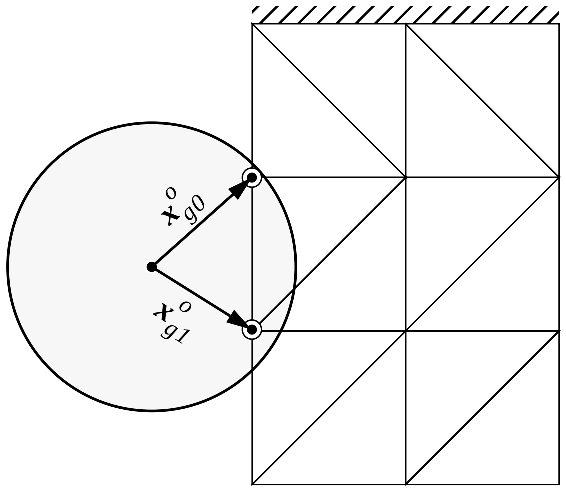

FIGURE 41. Grasping of a triangle.

This method is set up the same way as the method of grasping a node, except that instead of finding the closest node, the closest triangle within the grasping radius of the tool is located. Once the closest triangle is found, grasping offsets are calculated for the three vertices that make up the triangle:

$$\boldsymbol{x}^o_{gi} = {}^g_0\boldsymbol{R}^T(\boldsymbol{x}_i(t) - \boldsymbol{x}_g(t)) \qquad i = 1\ldots3 \tag{EQ 47}$$

where $\boldsymbol{x}^o_{gi}$ is the current offset within the grasper reference frame of the *i*th vertex of the closest triangle, ${}^g_0\boldsymbol{R}^T$ is the transpose of the rotation matrix between the grasper's reference frame and the inertial frame, $\boldsymbol{x}_i(t)$ is the current position of the *i*th vertex of the triangle, and $\boldsymbol{x}_g(t)$ is the current position of the grasper.

At each time step, then, the current position of each vertex of the grasped, closest, triangle is set to:

$$\boldsymbol{x}_i(t) = \boldsymbol{x}_g(t) + {}^g_0\boldsymbol{R}\boldsymbol{x}^o_{gi} \qquad \text{(EQ 48)}$$

In this manner, the triangles current state, both position and orientation, with respect to the grasper is fixed until the grasper is opened.

7.2.3 Grasping a Point Within a Triangle

One other method for grasping a model is to affix the grasper to the model with a stiff spring damper system. While there is no direct analog to this within the range of medical instruments, it can be useful as a method of directly applying forces instead of displacements to the modeled tissue.

Grasping can be viewed as shown in Figure 42. When grasping is initiated, the closest point on the model to the center of the grasper is found, and then that point is connected to the grasper with a spring and damper.

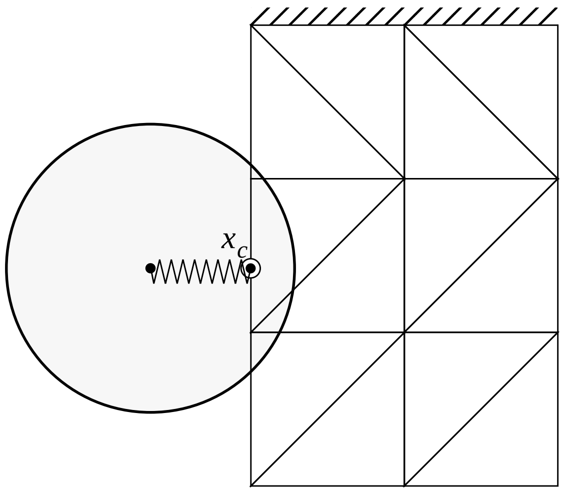

FIGURE 42. Grasping the closest point on the model to the grasping tool.

The force applied to the closest point, and back to the user through the grasping tool, is proportional to the displacement between the tool and the closest point and the relative velocity between the two:

$$\boldsymbol{f}_c^e = k(r_a - |\boldsymbol{x}_c(t) - \boldsymbol{x}_g(t)|)\frac{\boldsymbol{x}_c(t) - \boldsymbol{x}_g(t)}{|\boldsymbol{x}_c(t) - \boldsymbol{x}_g(t)|} + b\left((\dot{\boldsymbol{x}}_c(t) - \dot{\boldsymbol{x}}_g(t)) \cdot \frac{\boldsymbol{x}_c(t) - \boldsymbol{x}_g(t)}{|\boldsymbol{x}_c(t) - \boldsymbol{x}_g(t)|}\right)\frac{\boldsymbol{x}_c(t) - \boldsymbol{x}_g(t)}{|\boldsymbol{x}_c(t) - \boldsymbol{x}_g(t)|} \qquad \text{(EQ 49)}$$

$$\boldsymbol{f}_g^e = -\boldsymbol{f}_c^e$$

where $\boldsymbol{f}_c^e$ is the force applied to the object at the closest point to the grasper, k and b are the stiffness and damping terms for the grasper, $\boldsymbol{x}_c(t)$ is the current position of the closest point, $\boldsymbol{x}_s(t)$ is the current position of the grasper, and $\boldsymbol{f}_g^e$ is the force applied back to the grasping tool. Note that the relative velocity term is dotted with the unit vector between the closest point and the grasper. This insures that only the velocity along the direction of the applied force is included in the damping term. Lastly, the value of *r* that is used in Equation 49 is adjusted when grasping begins so that the total force applied by the user, and back to the user, is the same as the force right before grasping commenced, when the grasper was palpating the model.

$$r_a = \frac{1}{k}f_p + |\boldsymbol{x}_c(t) - \boldsymbol{x}_g(t)| \qquad \text{(EQ 50)}$$

where r_a is the adjusted radius that is used in Equation 49 and f_p is the magnitude of the force acting on the grabber at the time step before grasping commenced.

$\boldsymbol{f}_c^e$ is the total force being applied to the closest point on the model. This force needs to be divided between the three nodes in a manner that does not generate any moments about the closest point. To do this, we utilize Equation 41 to generate the relative fractions of the total force that each vertex receives.

Figure 43 demonstrates a rectangular model being grasped using this method, with the grasping tool being pulled away, to the left, from the model. This model generates the best feeling response of the three described, although the second method can be used if control of the orientation of the grasped tissue is desired.

7.3 Needle Puncture Modeling

Placing a needle and puncturing soft tissue is done countless times in surgery. It is done when giving injections, when placing sutures, and when performing biopsies. The simulation does not currently simulate the placement of sutures, but does generate the forces created by the motion of the needle through the soft tissue. We have implemented a simple sharpness model for the needle to initiate penetration, and then track the motion and path of the needle through the object to generate transverse forces.

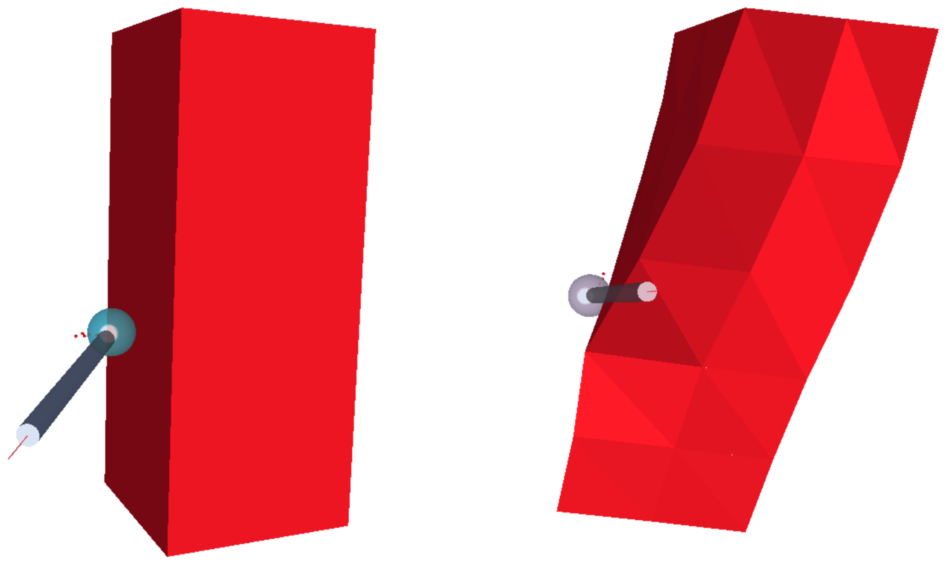

FIGURE 43. Example of soft tissue model being grasped and pulled upon.

7.3.1 Needle Sharpness Model

Needles are not infinitely sharp, so there is some deformation of the underlying tissue before puncture initially occurs. Looking at this as an elastic material, puncture will occur when the local stress passes a threshold for that tissue type. From an experimental point of view, though, we can simply look at the force being generated by the needle before puncture occurs. Using the now familiar penalty based method, a needle pushing on the tissue will generate the force:

$$\boldsymbol{f}_p^e = k(\boldsymbol{x}_p(t) - \boldsymbol{x}_n(t)) \tag{EQ 51}$$

where $\boldsymbol{f}_p^e$ is the force applied to the object by the needle, k is a stiffness term associated with the needle, $\boldsymbol{x}_p(t)$ is the current position of the closest point on the surface to the tip of the needle, and $\boldsymbol{x}_n(t)$ is the current position of the tip of the needle. Before puncture occurs, this force would be transferred to the vertices of the triangle using Equation 41.

If this force passes some threshold, based on tissue properties, sharpness of the needle, and the direction of the needle with respect to its velocity, then puncture will occur. This force threshold can be represented as:

$${}_n^T\boldsymbol{f}(t) = {}_t^T\boldsymbol{f}\frac{s}{|\hat{\boldsymbol{v}}_n(t) \cdot \hat{\boldsymbol{x}}_n(t)|} \tag{EQ 52}$$

where ${}_{n}^{T}\boldsymbol{f}(t)$ is the threshold force for the needle attempting to puncture the model at the current time step *t*, *s* is sharpness value for the needle, ${}_{t}^{T}\boldsymbol{f}$ is the force threshold value for this particular tissue type, $\hat{\boldsymbol{x}}_n(t)$ is the direction that the needle is pointing, and $\hat{\boldsymbol{v}}_n(t)$ is the direction that the needle is moving. The sharpness value is greater or equal to zero, where a value of zero would imply that the needle is infinitely sharp, and the threshold to initiate puncture is zero, and increasing values signify a needle that is more and more dull, and therefore will require more force to initiate puncture. The sharpness value can also be viewed as the radius of curvature of the tip of the needle. The last term, the dot product between the needle direction and the direction the needle is moving, encodes whether the needle is pushing, along its length, into the tissue or not. This would return a value ranging from one, if the needle is moving in the direction it is pointing, down to zero if the needle is just pushing from its side. An example of the initiation of a needle puncture is shown in Figure 44.

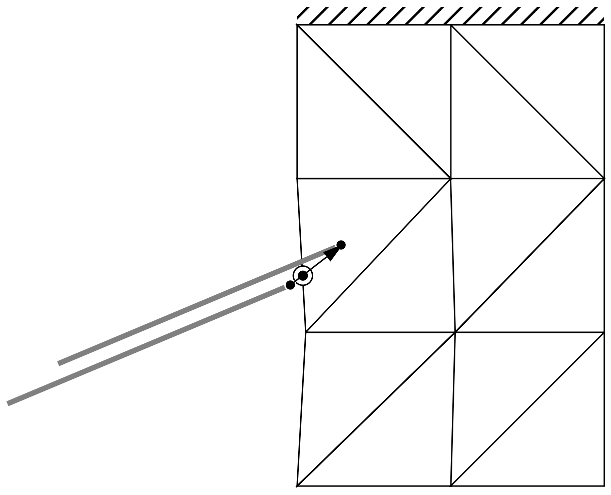

FIGURE 44. The beginning of puncture with a needle.

7.3.2 Propagation of the Needle Path

The first thing that is done after puncture has commenced, and at each time step during puncture, is to determine if the needle has been pulled back through the last triangle it passed through. As the needle is pushed into the soft tissue, we track all the triangles that it passes through, and record their local intersection coordinates in an ordered list, from first to last intersection point. So, to see if the needle is being pulled back out of the model, we check to see if the tip of the needle has passed back through the triangle of the last intersection point. If so, then we remove that intersection point from the list. If the needle tip did not pass back through the last intersection point, we then check the other triangles of the tetrahedron that the last intersection point is on, to see if the needle moved forward out of that element in the last time step. If so, we add the new punctured triangle to the list. These two possibilities are shown in Figure 45.

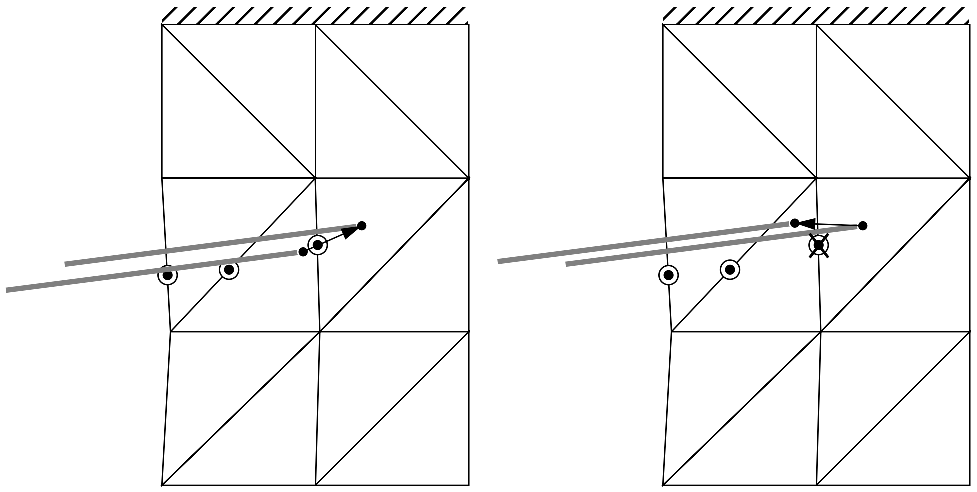

FIGURE 45. Checking the backward or forward motion of the needle.

Once the list has been updated with the motion of the triangle, we check to see if the list is empty. If the list is empty, then the needle pulled completely out of the model. If the list is not empty, then we generate the transverse forces to apply to the model based on the motion of the needle perpendicular to its recorded path.

7.3.3 Transverse Forces Generated by a Needle

After puncture is initiated, the needle will trace a path through the soft tissue. If the needle moves away from this path that it has traced, then forces should be generated to move both the tissue over towards the current position of the needle and the needle over towards the path that it had started tracing through the soft tissue.

For each intersection point that the needle passed through, we find the closest point on the needle, and generate a force to apply to the intersection point based on this displacement:

$$\boldsymbol{f}_I^e = k(\boldsymbol{x}_c(t) - \boldsymbol{x}_I(t)) \tag{EQ 53}$$

where $\boldsymbol{f}_I^e$ is the force to the applied at the intersection point, k is the stiffness measure of the needle, $\boldsymbol{x}_c(t)$ is the current position of the closest point on the needle to $\boldsymbol{x}_I(t)$, the current position of this intersection point. This force is then split up amongst the three vertices of the triangle using Equation 41. The force applied back to the needle is the opposite of the sum of the forces applied to the triangles:

$$\boldsymbol{f}_N^t = -\sum_{i \in I(N)} \boldsymbol{f}_{Ii}^e \tag{EQ 54}$$

where $\boldsymbol{f}^{t}_{N}$ is the total force acting on the needle, *I(N)* is the set of intersected triangles, and $\boldsymbol{f}^{e}_{Ii}$ is the force acting on the *i*th intersection point. Because the haptic interface we use can only display 3 degrees of freedom, we do not calculate the moment generated on the needle by the forces the user generates within the model.

A graphical example of how the forces on the individual intersection points are ascertained is shown in Figure 46. Note that in Figure 47 the object is deflected slightly due to the displacement of the needle from along its initial intersection points, represented by the dots in the wireframe image.

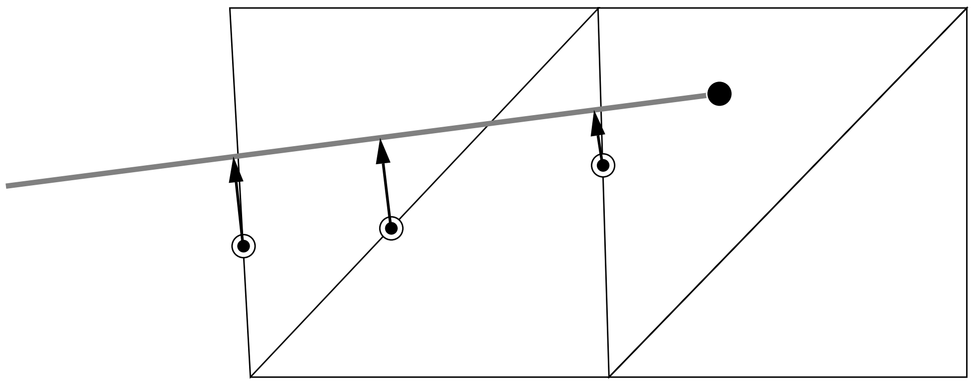

FIGURE 46. Force generation due to needle deflection.

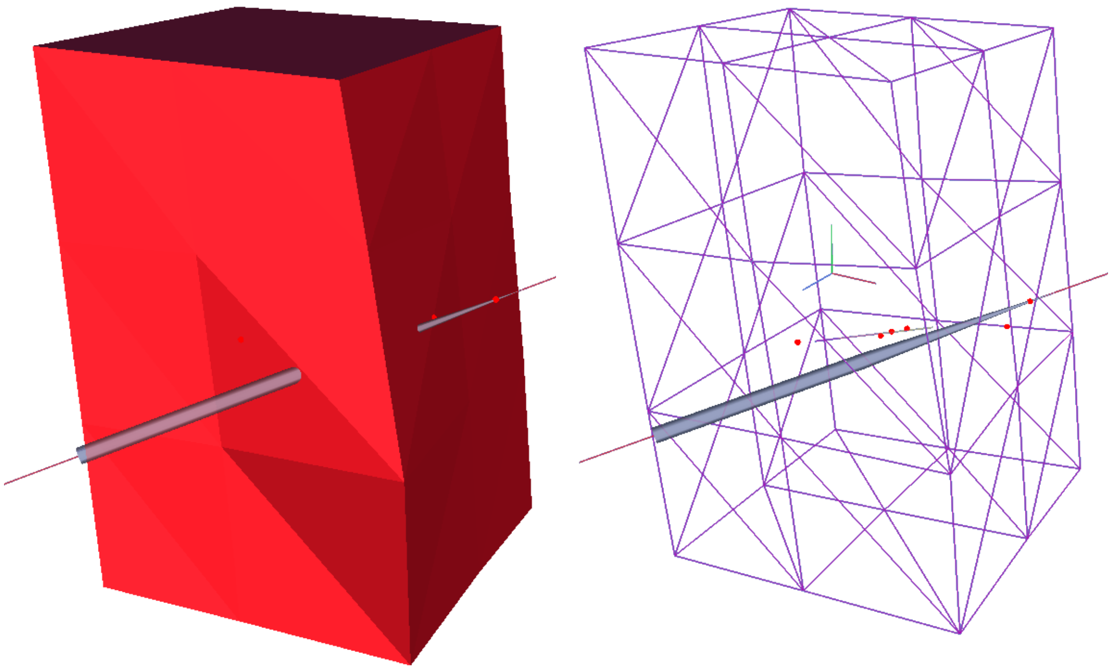

FIGURE 47. Example of object deformation caused by needle deflection.

Chapter 8

Haptics

Surgical simulation requires a method of interaction to be used for training and practice for surgery. While a simple graphical interface could be used, possibly with a 6 degree of freedom input device, haptic feedback provides the most powerful and useful modality for a complete surgical simulator. The basic concept behind haptic feedback is to allow the user to feel, through a physical device, the modeled object and the effect of any action that she initiates. The device can interface with the user through a simple manifestation that is familiar to the user. One available haptic device has an endoscopic gripper handle, while the device we use has a simple cylindrical handle.

8.1 Haptic Feedback in a Surgical Simulator

The basic purpose of haptic feedback is to display forces to the user based on the current position of the device and the current state of the simulator. A typical update cycle is shown in Figure 48. At the beginning of the update cycle, the current state of the device is determined. The current state of the device is then fed to the simulator, which updates its current internal state based on the position of the device and any internal parameters and models. Additionally, the simulation calculates the forces and moments to display to the user based on the state of the device and the simulation. Then, the forces and moments from the simulator are transformed and displayed back to the user.

For the device that is used in this simulation, a haptic update rate of 1000Hz is considered ideal to ensure device stability [26]. The difficulty arises in that the simulator does not always update at 1000Hz, and has an indeterminate rate that can vary based on computational load and can intermittently drop down to a rate around

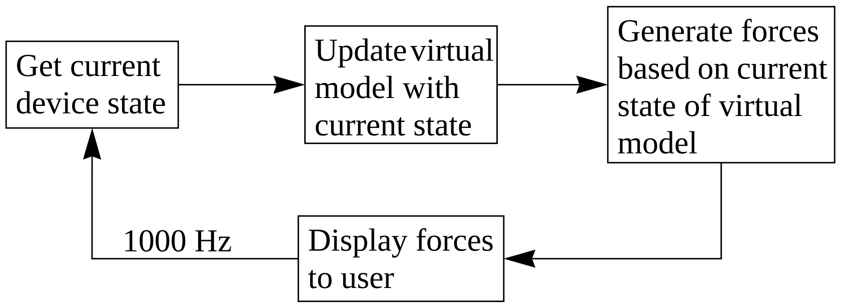
Get current device state
Update virtual model with current state
Generate forces based on current state of virtual model
Display forces to user
1000 Hz

FIGURE 48. Basic haptic feedback loop.

100Hz. Therefore, a method to generate intermediate forces based on slow and indeterminate updates is required.

8.2 Intermediate Representation

An intermediate representation was implemented, similar to the method described in [1]. The basic concept is to take a slow update rate simulation running as a haptic client, and run a simple, local model on the haptic server at a high update rate. This removes the effects of a zeroth order hold on the haptics system, where the haptic device might display a constant force, $\boldsymbol{F_1}$, for a time step of 5ms, and then a constant force, $\boldsymbol{F_2}$, for a time step of 3ms, and so on. With the intermediate representation, a local model that would generate forces similar to $\boldsymbol{F_1}$ would run for 5ms, and then after the next update, a local model that would generate forces similar to $\boldsymbol{F_2}$ would run until the next update. The possible difference between the two haptic modes is illustrated in Figure 49. The graph on the left demonstrates the force displayed without a local model, while the graph on the right demonstrates how the force can vary with a locally updated model. In this way, an indeterminate, slow, and erratically updated simulation would still give rise to a stable haptic experience.

Additionally, with a local model, the force displayed can be much more realistic. Without a local model, a certain constant force is displayed to the user, even if she moves in such a way as to break contact with the simulated object. Even though the force displayed to the user should be zeroed, there might be a noticeable lag before the force is zeroed due to the slow update rate. With a local model of the object updated at the full 1000Hz, if the user attempts to break contact with the simulated object, not only will the force be zeroed at the boundary of the local model, the force will correctly ramp down from the initial value to zero in a smooth fashion.

Three different types of local simulation were implemented and are described in the following sections. The first method, servoing to a setpoint, demonstrates a first, and naive, attempt for generating local models. The other two methods, local plane and

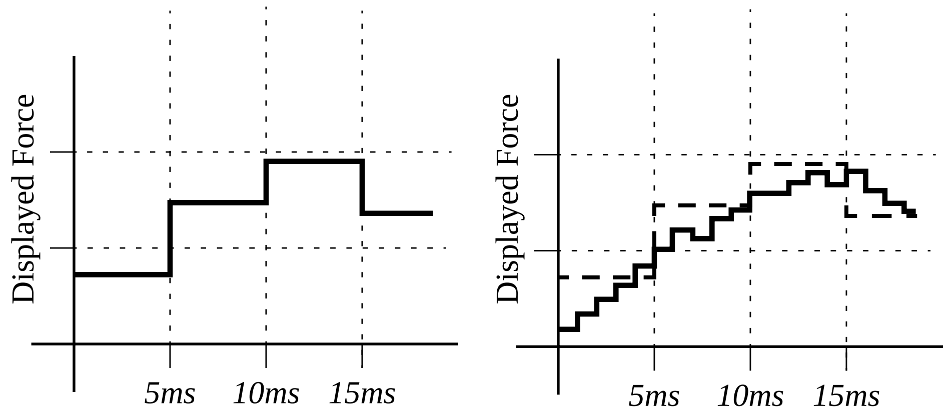

FIGURE 49. Typical force levels with and without intermediate representation with slow update rates.

line models, are the methods used in our simulator to generate appropriate forces that behave in a correct manner as the user's position changes with respect to the model.

8.2.1 Setpoint Local Model

The local setpoint model utilizes a setpoint calculated by the object simulation based on the current position of the user. For instance, utilizing a penalty based method for generating forces, where the force magnitude and direction are determined by the penetration of the user's position within the model, we can project the current position of the user back out to the surface of the model. Given this, forces can then be generated by servoing to the setpoint. For instance, in Figure 50 we see that the probe has penetrated slightly into the modeled object. That position is projected out to the surface, which is then communicated to the haptics server as the current setpoint.

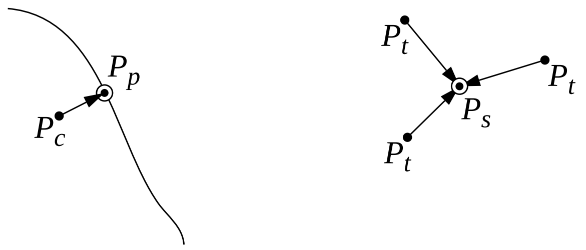

FIGURE 50. Example of servoing to a point.

The equation for the force generated by servoing to a point is:

$$F = -k(P_t - P_s) \tag{EQ 55}$$

where F is the force displayed back to the user, k is a gain term, P_t is the current position of the haptic device at time t, and P_s is the current position of the setpoint since the last update.

An extension of this method is to extrapolate and move the setpoint based on the recent motion of the setpoint. In this manner, we can account for the motion of the user. To do this, we move the setpoint along the line projected from the last two setpoints, and then servo to the moving setpoint. The equation for the motion of the setpoint is:

$$ {}^{t}P_s = \left(1 - \frac{t - t_u}{T}\right)(2P_{u-T} - P_{u-2T}) + \frac{t - t_u}{T}(2P_u - P_{u-T}) \qquad \text{(EQ 56)} $$

where ${}^{t}P_s$ is the position of the extrapolated setpoint at the current timestep t, t_u is the timestep when P_u was last received, P_u is the most recent received setpoint set at timestep t_u, P_{u-T} is the previous received setpoint set at timestep t_{u-T}, and T is the number of timesteps that passed between the previous two updates of the setpoint. Using this equation, the setpoint moves evenly along the path predicted by the previous two updates of the setpoint, as shown in Figure 51, and does not try to hold the user to a particular point in space.

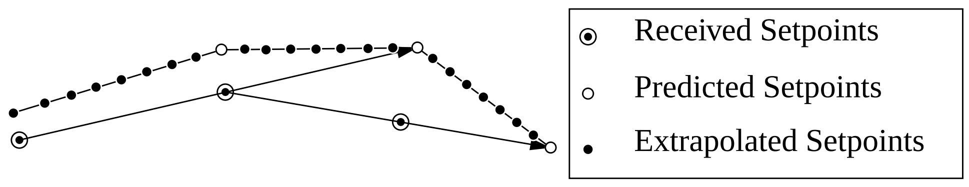

FIGURE 51. Extrapolated setpoint example.

There are clear problems with both of the setpoint methods. The first method generates a very sticky experience. If the user is interacting with a sphere, and is feeling the shape in a circular motion, then as she tries to move the device around the sphere, it is as if she is stuck to one point momentarily. Then, that point moves closer to her current location, at which point the device can move around the sphere a little more, given a similar force as generated at the previous setpoint. In this manner, a path can be traced around the surface, but it is punctuated by many hangups along the way. Additionally, this method does not deal appropriately with the user trying to pull away from the object. With a setpoint, if the user pulls back, away from the sphere, the local simulation still servos to the setpoint, thereby imparting a sticky feeling to the object.

To partially alleviate one of these problems, the extrapolated setpoint helps with the feeling of little hangups as the user traces a path. This works well as long as the user maintains an even pace. But, if the user tries to stop, for example, the setpoints will continue to extrapolate beyond the user's current position, and try to pull the user along the path of the setpoints. In this way, the simulation feels partly alive, since it adds energy back to the user. Also, this enhancement does not help alleviate the stickiness problem of the general setpoint method. Because of these problems, we investigated other local models for generating forces to display.

8.2.2 Constraint Plane Local Model

Interacting with a plane can generate a much more realistic experience than servoing to a point. The constraint plane local model method generates that experience by taking the position and direction of a plane and generating forces to keep the user on the positive side of that plane. In this manner, there is no attachment to a particular point, and no stickiness. In Figure 52, the plane shown is the local approximation of the surface, and forces are generated while the user is on the negative side of the plane.

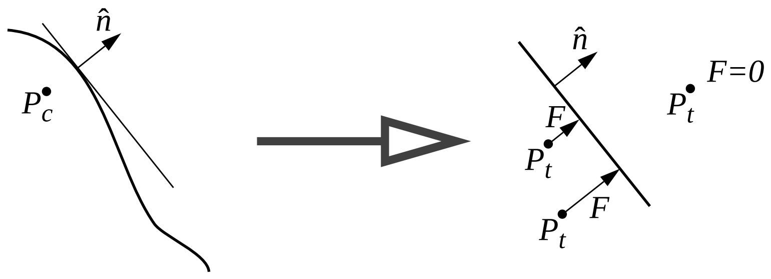

FIGURE 52. Constraint plane for local modeling.

As before, the simulation determines the closest point on the model to the user's current position. The surface normal at that point is then calculated, and the position and direction are communicated to the haptics server. On the haptic server side, given the position and direction of the constraint plane, the signed distance from the current position of the user to the constraint plane is calculated. If it is greater than or equal to zero, then no force is applied because the user has stopped interacting with the surface. Otherwise, the force is proportional to the depth of penetration, utilizing a penalty based method:

$$\begin{aligned} d \geq 0: \quad & \boldsymbol{F} = 0 \\ d < 0: \quad & \boldsymbol{F} = -kd\hat{\boldsymbol{n}} \end{aligned} \qquad \text{(EQ 57)}$$

where d is the depth of penetration, or the distance between the current position and the constraint plane, k is the gain or stiffness of the plane, and $\hat{n}$ is the normal of the plane.

The constraint plane method is used for all palpations of the deformable models. It also is used for grasping, where the force generated by the deformation of the model is used to create a local plane, pointing in the direction of the force vector generated by the grasping subroutine.

8.2.3 Line Constraint Local Model

The last local method implemented, to enable the simulation of needle puncture, is a line constraint mode. In this mode, forces are generated to move the user back toward a line in space, for instance, a line demarcating the current path of a needle. In this model, the user is free to move along the line, but will feel perpendicular forces if she tries to move away from the line in space, as shown in Figure 53.

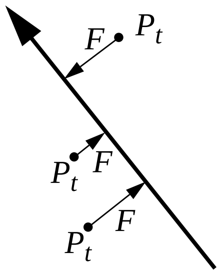

FIGURE 53. Line constraint for local modeling.

The data communicated to the local simulation running on the haptic server is similar to the plane model. The object simulation determines the direction of the constraint and position of the constraint line, which, for a needle stick, would be the location and direction of the path traced out by the needle up to its current location. Given that information, the haptic server then calculates the closest point on the constraint line to the current position of the device. The force generated and displayed to the user is proportional, then, to the vector between this closest point and the current position of the device:

$$\boldsymbol{F} = -k(P_t - P_c) \qquad \text{(EQ 58)}$$

where $\boldsymbol{F}$ is the force displayed back to the user, k is a gain term, P_t is the current position of the haptic device at time t, and P_c is the closest position on the constraint line. Note that at this point that the equation for $\boldsymbol{F}$ is very similar to that in the setpoint model. The significant difference is that the setpoint in the line constraint mode, as the closet point can be viewed, is free to move along the constraint line, and therefore does not generate the stickiness present in the setpoint model.

Chapter 9

Implementation Details

The general theory of the different components that make up this thesis were described in the previous chapters. While these descriptions are complete, as with any experimental system, there are many details of the implementation which directly affect the performance and quality of the work. In this chapter, we will describe the general layout of the experimental system, how the soft tissue model was created and updated, the way that the interaction and cutting routines fit within the scope of the simulator, and details on the haptics and graphics subsystems.

9.1 System Setup and Implementation

The simulator is composed of three main subsystems, the soft tissue simulation and interaction routines, the graphical subsystem, and the haptics subsystem. The soft tissue simulation sits at the core of the simulator, with the modification and interaction routines running concurrent with it. The graphics subsystem is another part of the main simulator, updating the graphical scene at 30Hz. Running on a separate machine is the haptics server. This server was implemented separately to insure the safety of the haptic device, so that it would behave gracefully if the simulator itself were to crash.

The system can be viewed in block-diagram form in Figure 54. The system starts up by either reading in from a data file or creating from scratch the soft tissue model. The tools that might be used are then created, and the graphics system started up. Next, the simulator tries to find a haptics server to generate user position updates. Once it connects to a haptic server, it starts updating the model's state and calling the appropriate interaction routines, based on position data it receives from the haptic server. At each time step, the simulator receives an updated user position from the haptics server. It then runs the interaction routine that is currently selected, and modifies or perturbs the model as necessary. The last step is to run the position integration routine. After that, the simulation returns, and then repeats. The simulation routines are run on a 1000Hz interrupt driven rate. If the process takes more than 1ms, then the process runs as fast as possible. The graphics routine runs at 30Hz. It runs as a separate process from the soft tissue simulation, and uses semaphores to insure that the data it is reading is not currently in use by the soft tissue routines.

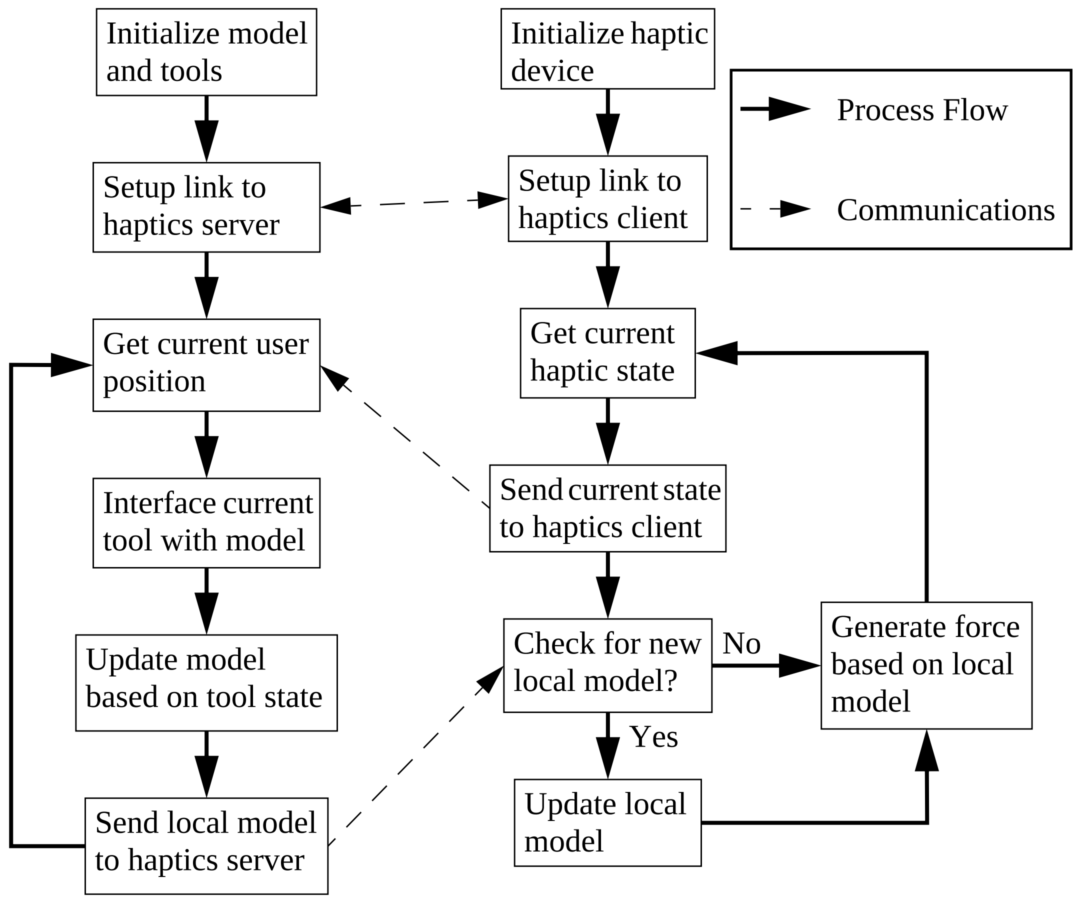
Initialize model and tools
Initialize haptic device
Process Flow
Communications
Setup link to haptics server
Setup link to haptics client
Get current user position
Get current haptic state
Interface current tool with model
Send current state to haptics client
Update model based on tool state
Check for new local model?
No
Yes
Generate force based on local model
Send local model to haptics server
Update local model

FIGURE 54. Block diagram of system flow.

The haptics server runs on a separate machine, and is started up separately. On startup, it initializes the haptic device, if necessary, and then opens a communications port and waits for a client to connect. Once a client connects, it receives commands from the client on changing the state of the device controller, such as rezeroing the device or enabling or disabling forces, and updates of the intermediate representation for generating forces. The haptics server also cycles on a 1000Hz interrupt, and at each cycle, it checks to see if it has received an update of the intermediate representation. If it has, it updates that representation, whereas if it doesn't, it just continues to run with the previous local model. It then generates forces to display to the user based on the current intermediate representation. The last part of the cycle is to broadcast the current position of the device back to the user. The server sends current position data back to the client at twice the rate it receives data, up to 1000Hz. This insures that the client does not receive a flood of updates at a rate much higher than it is running at. For example, if the server receives updates at 200Hz, which is the rate that the soft tissue simulation would be running at, then it only sends out updates at 400Hz. When the simulator next checks for updates, it might have two updates there, and it discards the older one.

The soft tissue system is currently implemented to run on a single or dual processor SGI. Results shown in this thesis were generated on a dual processor SGI Octane, with 250MHz R10000 processors. One processor handled the simulation of the soft tissue and the interaction routes, while the other processor handled the graphics rendering. The graphic board in this machine is an MXI. The haptics server runs on a single processor SGI Indigo-2 Extreme with a 250MHz R4400 processor. Communications between the two machines runs over 100Base-T ethernet.

9.2 Linear Elastic Soft Tissue Modeling

There are a great many different details on how our soft tissue model is implemented. In this section, we discuss the basis for our choice of tissue parameters and how the model is allocated and stored.

9.2.1 Tissue Parameters

While a linear elastic finite element model is not the best method for simulating soft tissue, due to its internal structure, we attempted to find tissue parameters that were roughly correct and appeared appropriate. The stiffness value used in the homogenous model of the liver was 2e6 N/m^2, which is similar to the value used in [38] and to the values determined for bovine livers in [11]. Tissue density of 1.05 g/cm^3 was obtained from [21], and is used for iteratively updating the model state. The Raleigh damping parameters were determined empirically to damp out motion of the model in an appropriate time frame. A value of 1e-5 1/sec was used for α, and 2e-4 m*sec was used for β.

9.2.2 Object Construction within the Simulator

Finite element models can be implemented in many different ways. A popular method for generating a fast model is to precompute a stiffness matrix for the model and invert it ahead of time. Then, updating the state of the model is quite simply a large matrix multiplication. This is not possible for this model due to the modifications and changes that can occur in the model. To help facilitate the modification of the model, we needed a memory structure that would be easily and quickly updated, for both removing and adding elements to the model.

In this vein, we implemented the model as a linked list of elements, nodes, and edges. We used a linked list instead of a fixed size array because we can not predict the number of elements that might end up in the model after cutting occurs. If we had a fixed size array, and generated more elements due to a cut than could fit in that array, it might take a disproportionate amount of time to resize and move the array. On the other hand, adding elements individually only requires the allocation of small chunks of memory. This requires a fair amount of memory overhead, but ensures that there should always be enough room to add more elements, assuming that the machine itself hasn't run out of memory. The basic data structures are shown in Figure 55.

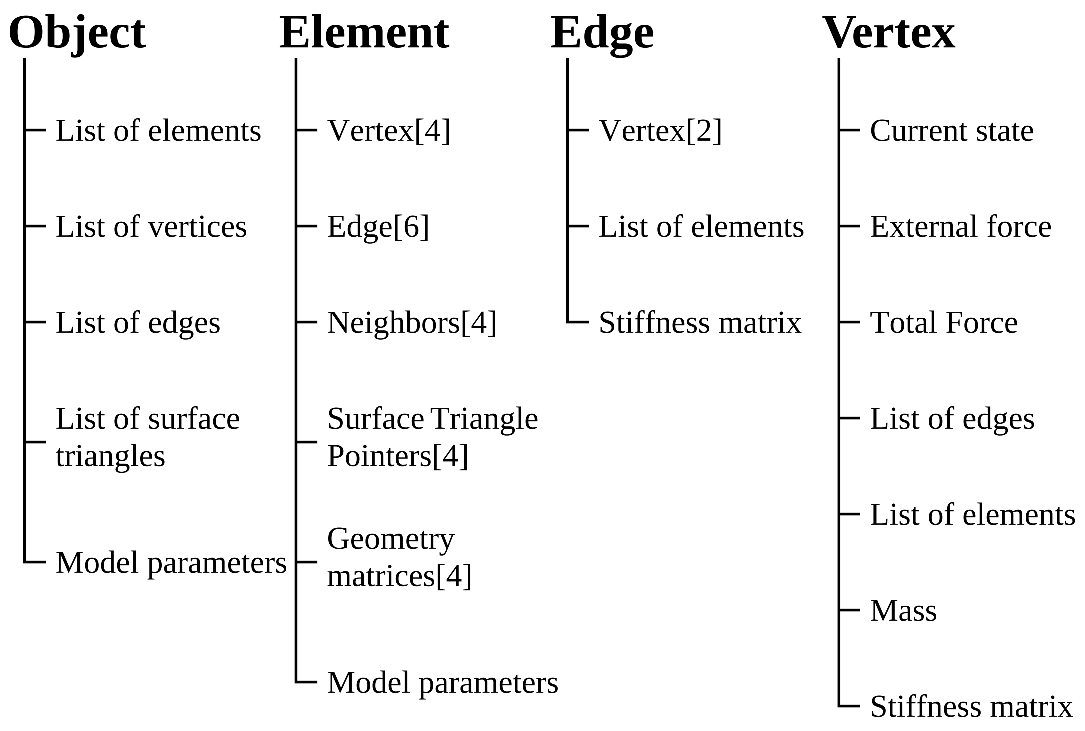

FIGURE 55. Basic data structures for the soft tissue model.

The object data type contains the basic pointers to the contents of the model. It contains linked lists of all the elements, nodes, edges, and surface triangles in the model. It also contains the tissue parameters and other values of import to the model. The object data type is passed to every function, to facilitate locating, and removing if necessary, any part of the model. The lists of vertices and edges are used in the model update routines to quickly compute, for every node, the forces applied to it.

Elements are the next largest data structures, and contain pointers to the four nodes and six vertices that make up the element. The element data structure also includes the calculated $\boldsymbol{M}$ vectors to speed up any modification of the element that may be necessary. Lastly, they also contain pointers to their neighboring elements to help with quickly propagating intersection detection throughout the model.

The edge data structure contains pointers to its two endpoints, its stiffness matrix, and a list of elements that it is a member of.

The vertex data structure contains its current state: position, velocity, and acceleration, and the external and total forces acting on it. It holds the vertex’s mass and stiffness matrix. It also contains the list of edges and elements that the vertex belong to.

Both the edge and vertex data structures include lists of the other data types that they belong to, in order to facilitate checking of certain conditions and to help remove other structures when the underlying data type is removed. This allows quick and easy modification of the model. For instance, if we had to remove a vertex, all the edges and elements that use that vertex will have to be removed to. It is much faster to store a list of those edges and vertices than to have to search through the object lists to find which edges and elements need to be removed.

9.3 Interaction Routines

The routines that implement the different types of interaction that the user can have with the model, cutting, palpation, grasping, and needle puncture, are implemented as routines that can plug into the soft tissue simulation. In this way, it is very easy to create and add a new type of interaction routine, and to cycle through the available routines.

All the different routines act in the same manner within the framework of the simulation system: the state of the interaction tool is updated by the position update from the haptics server, the tool is tested against the model, resultant forces or modifications are then applied to the model, and then the routines return to the process that called them. In this case, the soft tissue modeling routine, which then feeds into the position integration state.

All of the tools include the relevant data for their action. For instance, the cutting tool includes the length and size of the cutting blade, the position and orientation of the blade, and the length of the handle. It also contains any data structures it needs to facilitate and speed up the cutting process, like a list of the currently intersected elements. The same is true for the other routines.

9.4 Haptics System

The haptics system uses the PHANToM device from Sensable Technologies, Inc. The haptics environment is built on a simple package that encapsulates the basic i/o package that ships with the PHANToM. The PHANToM is a version 1.5 model, which provides 3 degrees of freedom of feedback, and 6 degrees of freedom of input. While this device can not generate torques to display to the user, it is adequate for an experimental system.

9.5 Graphics System

The graphics subsystem was written in OpenGL on the SGI, and uses the glut library to perform simple windowing functions. The system can display the model as either a wireframe or surface rendered model, and shows graphical models of the user's tools within the workspace. Zooming is implemented, as is arbitrary rotation utilizing the Arcball routines from [44]. The monitor is considered to be the inertial reference frame, and rotations in the graphical view are transferred to the object data structure as the gravity direction, so that gravity always points down on the monitor. In this way, with gravity enabled, the object can be rotated and simple deformations of the model can be tested and shown. Simple stereo rendering is also implemented.

Chapter 10

Examples and Performance

The results of the progressive cutting techniques described in the previous chapters can be quantified in two ways. The first numerical result is the decrease in computational load the minimal set creation of these techniques has when compared with generating full sets. The second numerical result is the deviation of the generated cut surface from the surface traced out by the user. The second half of this chapter demonstrate examples of the simulator running with two types of model, a simple rectangular model, and a liver model courtesy of Project Epidaure at INRIA.

10.1 Changes in Update Rate

When the cutting tool passes through the model, all intersected elements are modified. The update rate of the state of the model is dictated by the size of the model, so the number of elements directly impact how quickly the model runs. When the described cutting routines process an intersected element, they generate between five and nine new elements, compared to seventeen new elements for a general subdivision, or the complete removal of the intersected element. Given the passage of the cutting tool through the model shown in the upper left in Figure 56, we generated subdivisions to demonstrate the following techniques:

1. the general minimal element creation method
2. cutting with snapping
3. and the complete removal of any intersected elements.

The number of elements that would have been created with a general subdivision, along with a comparison of computation times and expected update rates are shown in Table 5.

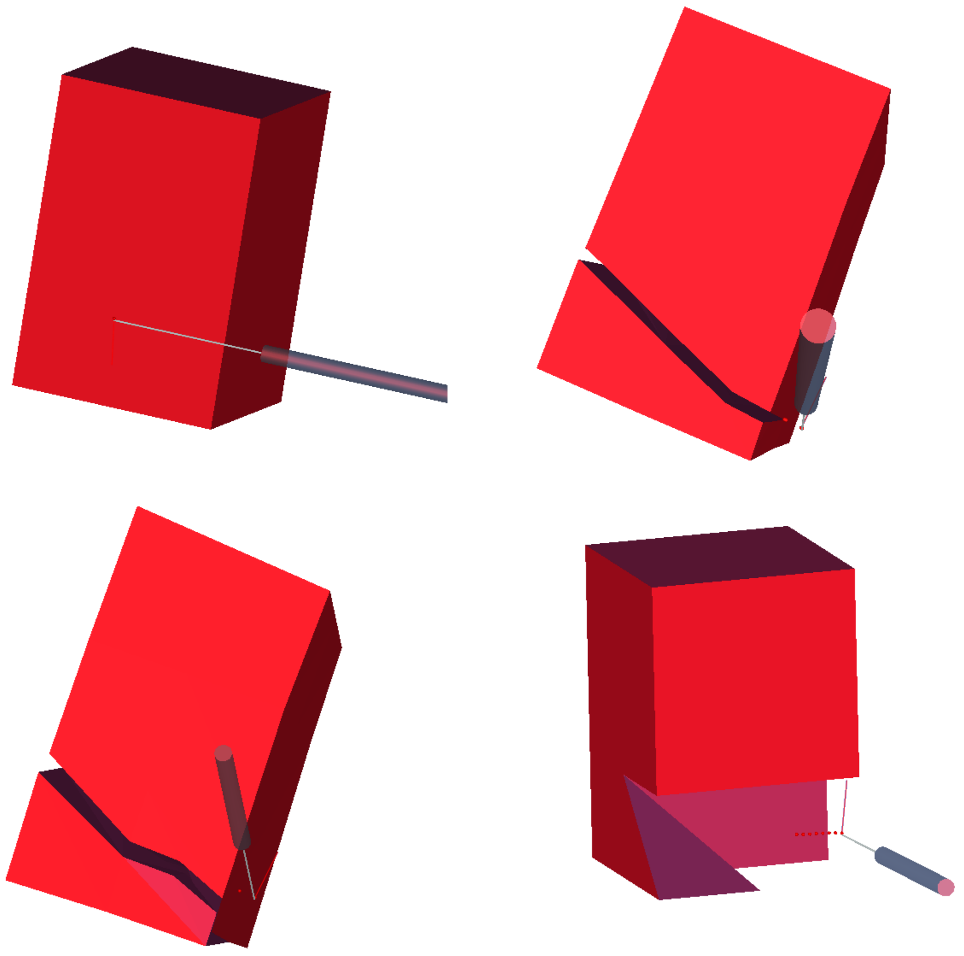

FIGURE 56. Completed cuts for showing changes in update rates.

	Original Number of Elements	Original Integration Time	Number of Elements After Cutting	Integration Time After Cutting	Decrease in Integration Time over General Subdivision
General Subdivision	72	0.00030	232	0.00109[a]	-
Element Removal	72	0.00030	62	0.00023	79%
Progressive Cutting	72	0.00030	128	0.00061	44%
Cutting w/Snapping	72	0.00030	106	0.00050	54%

TABLE 5. Changes in update rate and number of elements based on cutting method.

a. Expected integration time.

As we can see, using the general subdivision method the model more than tripled in size. Using the progressive cutting method, the model size increases by a much smaller number. The increase in the number of elements was more than two and a half times less, 6.6 new elements vs. 17 new elements for every element replaced. This translated, for this model, into 44% better integration time. Looking at the results for cutting with snapping, we achieved a similar improvement in the integration time and number of elements after cutting, with almost four times fewer new elements (4.4 new elements per cut element) and a 54% decrease in integration time after cutting. Completely clearing out intersected elements, predictably, resulted in fewer elements and a faster update time after cutting, but at the expense of cutting and model accuracy.

10.2 Distance of Cut Surface from User's Path

When the user moves a cutting tool through a model, intersections between the path of the cutting tool and the edges and faces of the model are generated. These intersections represent a discrete form of the path of the cutting tool. They also represent the ground truth of our knowledge of how the tool's path interacts with the model. This ground truth is generated by the progressive cutting method described in Section 5.3, since it only uses the intersection points created by the tool.

The second numerical comparison is based on the distance of the generated cut surface from this ground truth. Using the technique of completely removing any intersected surfaces, this metric would be the mean distance of the vertices that make up the removed elements from the ground truth path. For the cutting with snapping method, the metric is the mean distance of the vertices used on the generated cut surface from the ground truth path. The equation for this distance is:

$$\bar{d} = \frac{1}{n_v} \sum_{i \in V} \left| \boldsymbol{x}_i - {}^{t}\boldsymbol{x}_c \right| \qquad \text{(EQ 59)}$$

where $\bar{d}$ is the mean distance to the ground truth surface, n_v is the number of vertices on the cut surfaces, V is the set of those vertices, $\boldsymbol{x}_i$ is the position of the ith vertex, and ${}^{t}\boldsymbol{x}_c$ is the closest point on the ground truth surface to the ith vertex.

These calculations are performed on the undeformed models after the cutting occurs. The same path as in Figure 56 is used, and the actual cut surfaces are shown in Figure 57. First is the surface traced by the progressive cutting routines, which uses the precise intersection points. The second image is of the snapped progressive cutting example. Note how the surface is similar to the exact surface, but deflects away at points which were initially too close to edges and vertices of the original elements. The last image is of the surface generated by completely removing the intersected elements. Unlike the previous two images, where only one apparent surface is visible because the upper and lower cut surfaces are coincident, this surface has two distinct parts, since part of the original model was removed. Another example of cutting through a rectangular object is shown in Figure 58 and Figure 59. An example of the results of cutting a liver model with the different methods is shown in Figure 60 and Figure 61. Results for these examples are tabulated in Table 6.

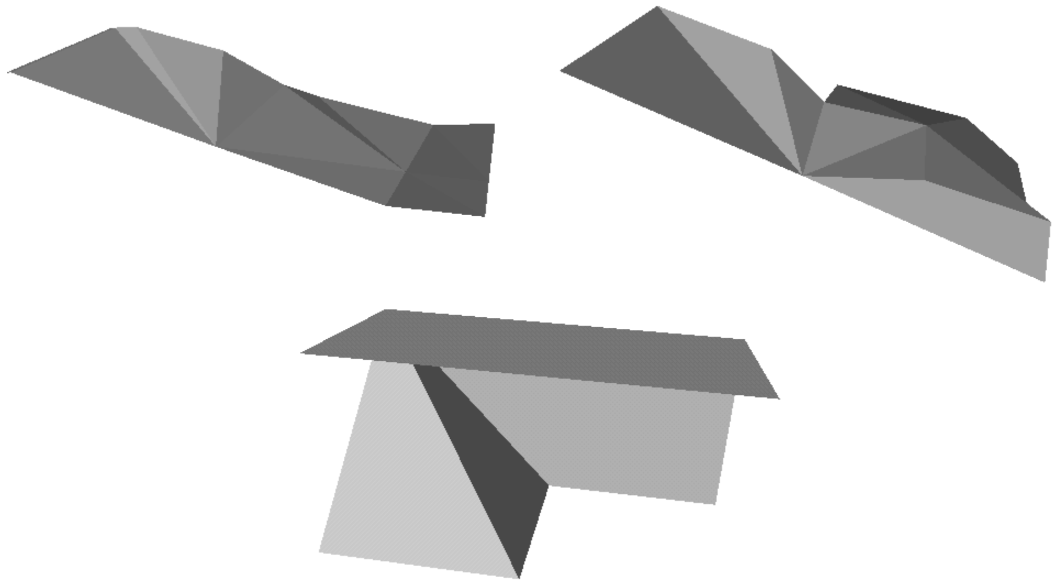

FIGURE 57. Cut surfaces generated by the same motion as in Figure 56.

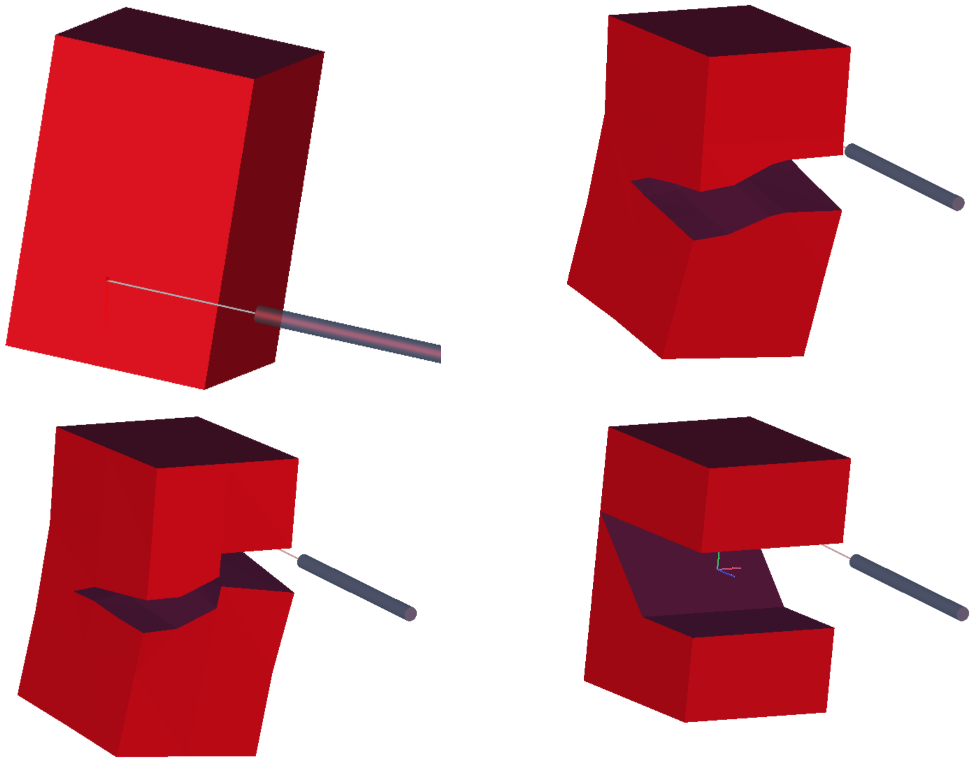

FIGURE 58. Second example of completed cuts demonstrating different cutting methods.

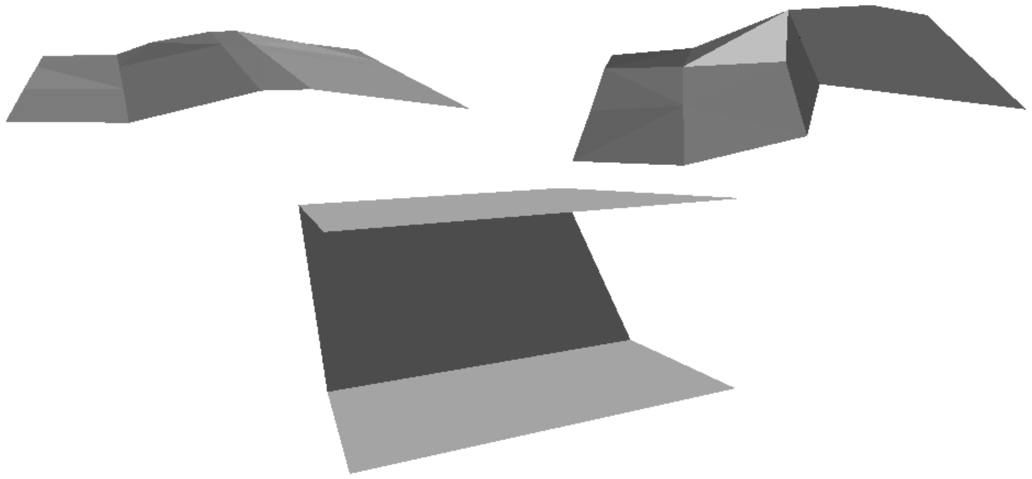

FIGURE 59. Cut surfaces generated by the same motion as in Figure 58.

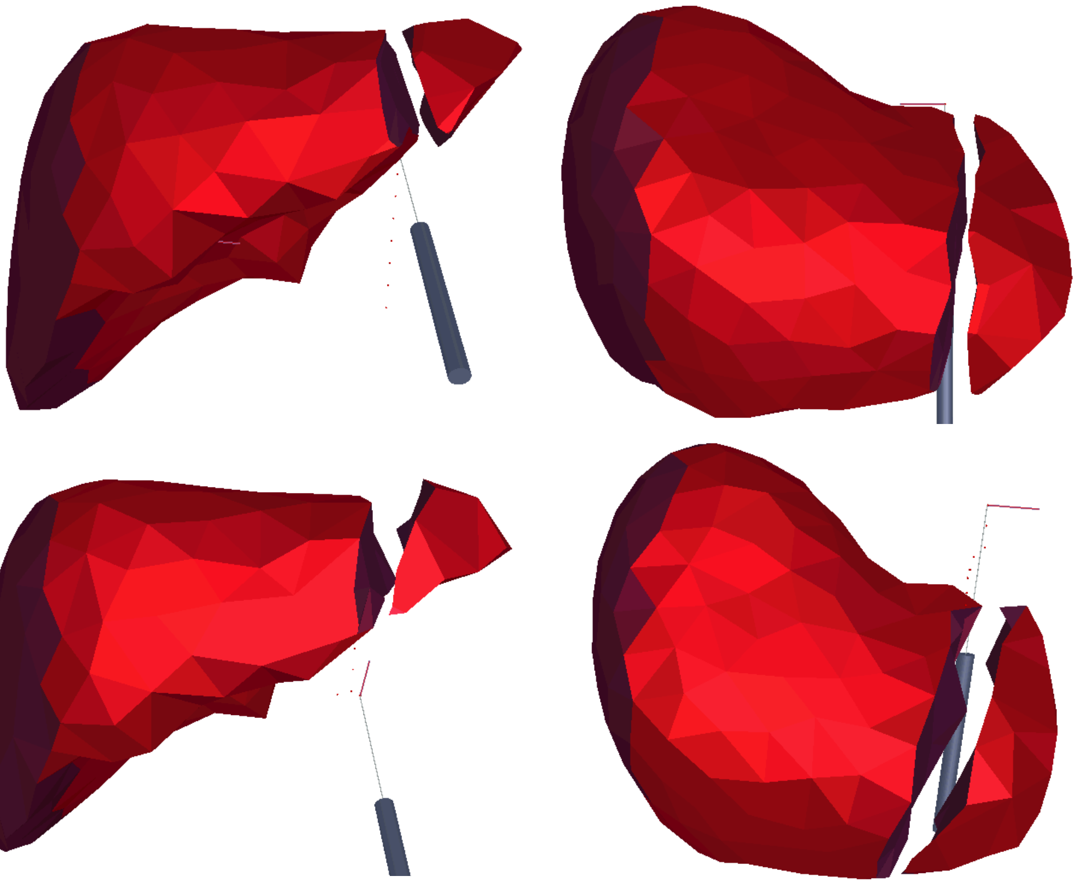

FIGURE 60. Liver model example (Exact, snapped, clearing cuts).

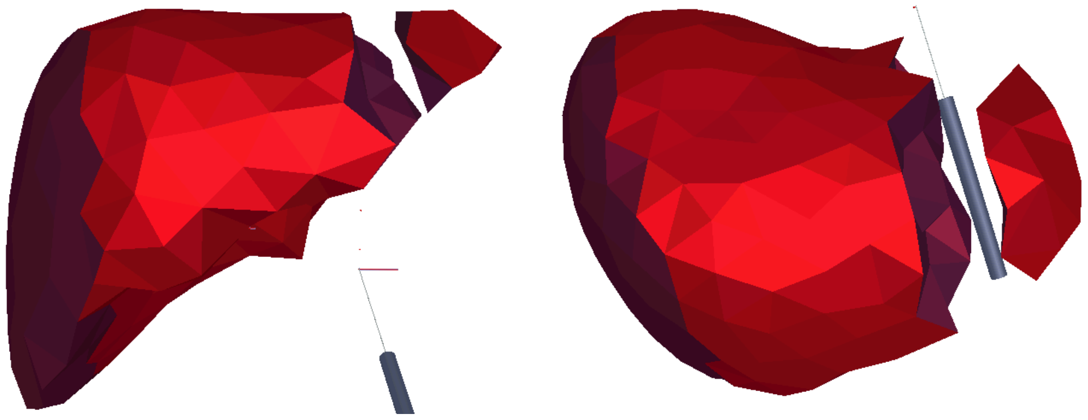

FIGURE 60. Liver model example (Exact, snapped, clearing cuts). (Continued)

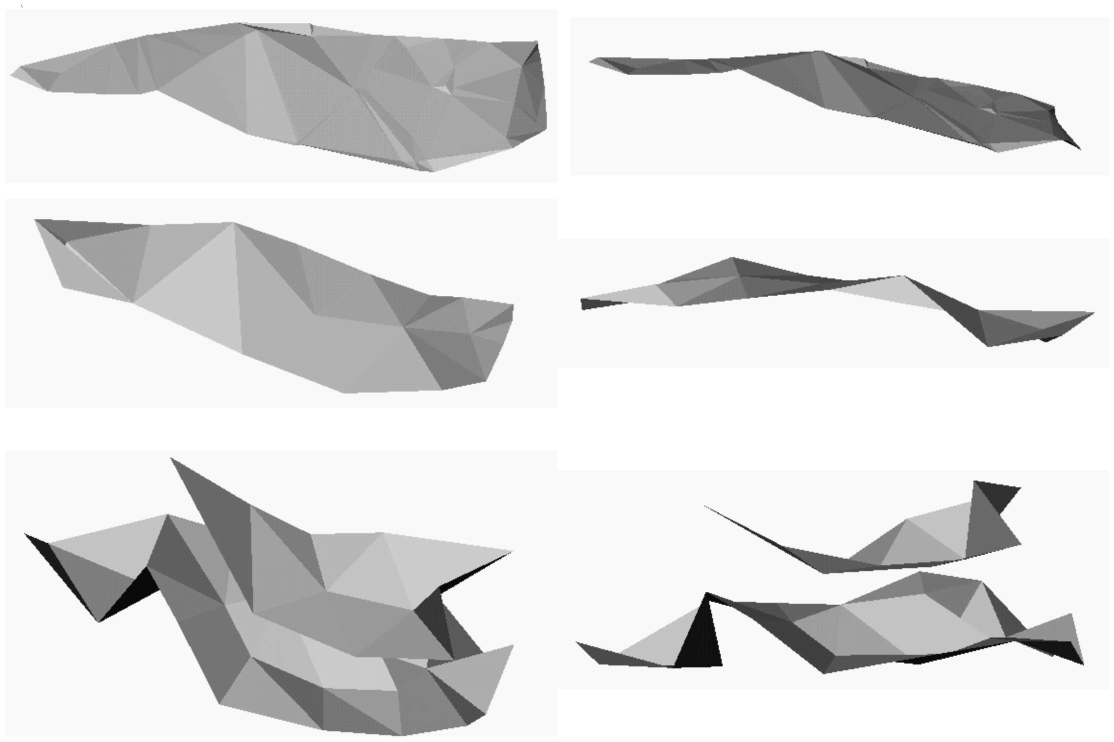

FIGURE 61. Cut surfaces of model shown in Figure 60. (Exact, snapped, clearing cut).

	Rect. Example 1	Rect. Example 2	Liver Example
Progressive Cutting	0.000 (mm)	0.000	0.000
Removal of Intersected Elements	14.050	16.008	13.833
Cutting with Snapping	3.318	2.368	2.162

TABLE 6. Mean distance from ground truth.

As can be seen from these results, while the progressive cutting does follow the path exactly, cutting with snapping is still significantly better than just removing the intersected elements, which generates the maximum deviation of the cut surface from the path traced out by the user. These distances are in relation to a typical edge length of 30mm for the rectangular model and an average edge length of 26.522mm for the liver model. Element removal, instead of subdivision, causes an average deflection from ground truth of half of the typical edge length. The deflection from the ground truth caused by snapping is on the order of 10% of the typical edge length. In the two rectangular examples, the improvement in actual distance from the ground truth surface due to cutting with snapping is 4.23 and 6.76 times. The improvement in the liver model example is 6.40 times better. Additionally, the snapping method does not remove mass and volume from the model. In the liver model, the clearing method, as shown by the distance between cut surfaces at rest in Figure 61, removes 3.5% of the object's volume. This reduces the mass of the liver from 3.470kg to 3.351kg, a change of 119 grams. The snapping method does not change the model's volume or mass at all.

10.3 Progressive Cutting with Temporary Subdivisions

Figure 62 shows the results of an example of cutting through a rectangular object that is under tension. There were 576 tetrahedra in the rectangular object before cutting, on a 4x6x4 cubical lattice, and 954 afterwards. 60 elements were cut, removed, and replaced by a new set of 312 tetrahedra, an average of 5.2 elements added for every element removed.

10.4 Interaction with a Rectangular Model

Testing of the simulator was done with a rectangular model to facilitate seeing deformations and performing predictable cuts and other interactions with the object. The basic shape of the object was shown in Figure 29, where the image on the left is of the undeformed object and the image on the right is the object under the influence of gravity. The nodes on the top of the model are anchored in space. This model is rendered in random colors to facilitate viewing of the deformation. The model used in the following figures is made up of 20 cubical blocks, 30mm on a side, where each block consists of six tetrahedra. The undeformed state of this model is shown in Figure 63. This model requires 0.188 milliseconds of computation per cycle to update the model state while using a time step of 1.0 milliseconds.

Figure 64 shows the effects of the user interacting with the model. The first image shows the user palpating the object with an implicit sphere model, while the next image shows the effects of the user grasping the model and pulling to the side.

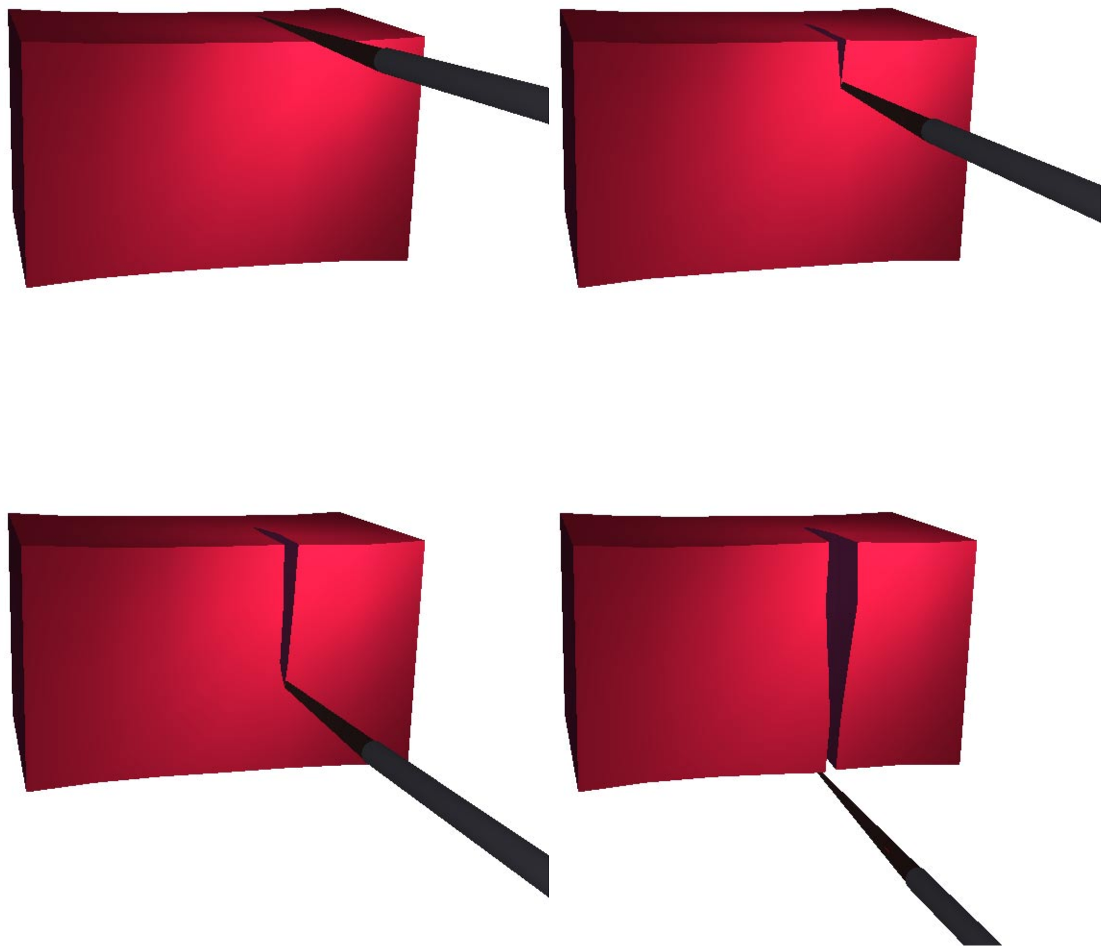

FIGURE 62. Progressive cutting, with temporary subdivisions, of a rectangular model.

The next two figures show the results of the user cutting the rectangular model with snapping. Figure 65 shows a partial cut through the object, while Figure 66 shows a complete cut through the object, with a simple displacement of the cut portion.

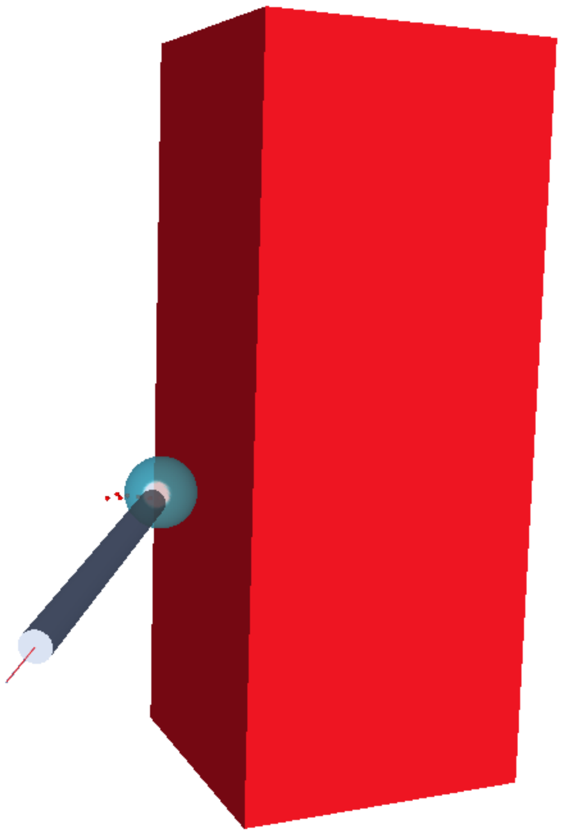

FIGURE 63. Undeformed image of basic rectangular model

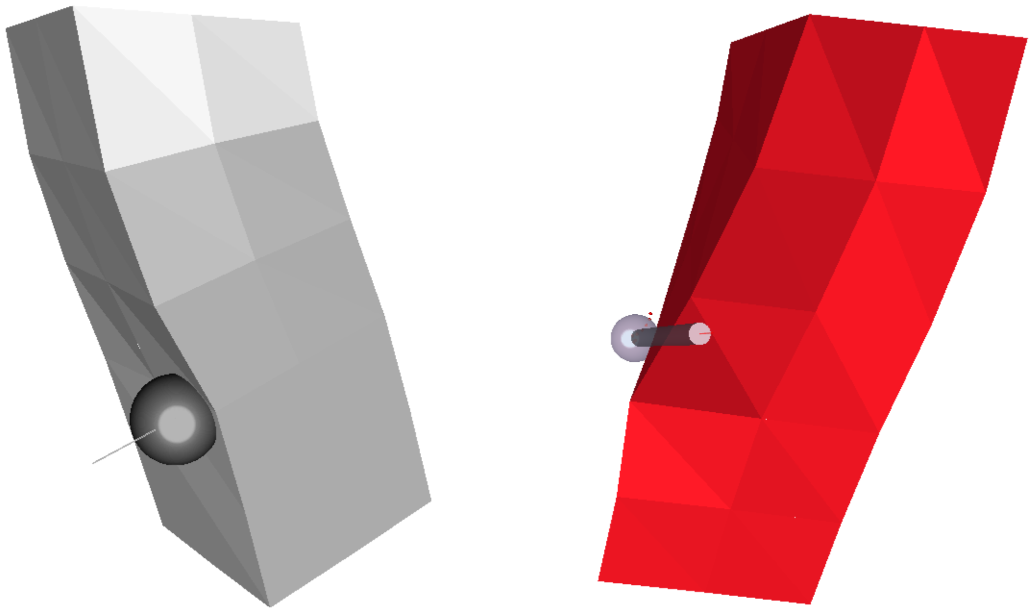

FIGURE 64. Palpating and grasping the rectangular model.

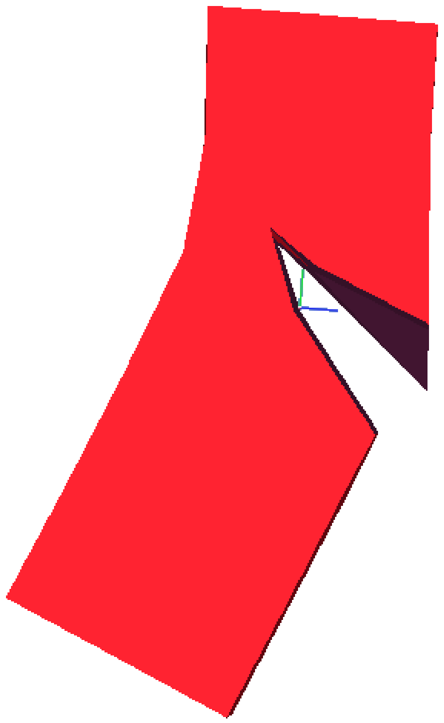

FIGURE 65. Partial cut of rectangular object.

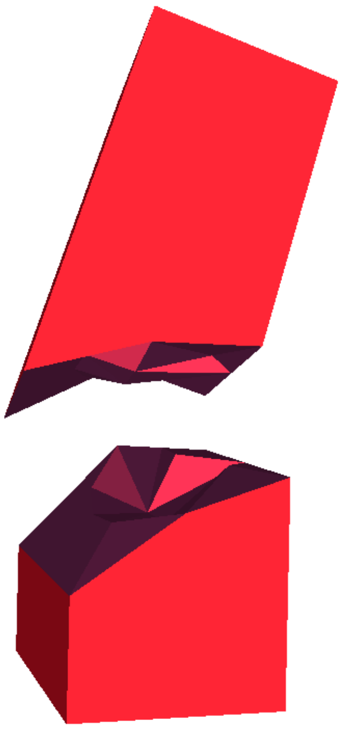

FIGURE 66. Complete cut of rectangular object.

10.5 Simulation of Liver Model

The underlying simulation of the liver model is identical to that of the rectangular model. The only difference is that the liver model is read in by the simulator from a data file, while the rectangular model is generated at runtime. The basic shape of the liver is shown in Figure 67. Note, that the surface nodes on the left side of the liver are anchored in space, to keep the liver fixed in space during interactions. A fixed surface to rest it on could have been used, but the collision detection required for that was deemed too expensive. The model consists of 467 nodes that make up 1967 elements. The simulation runs at approximately 300Hz, and has a time step of 0.001 seconds. As currently implemented, this model does not achieve synchronicity and is not real time.

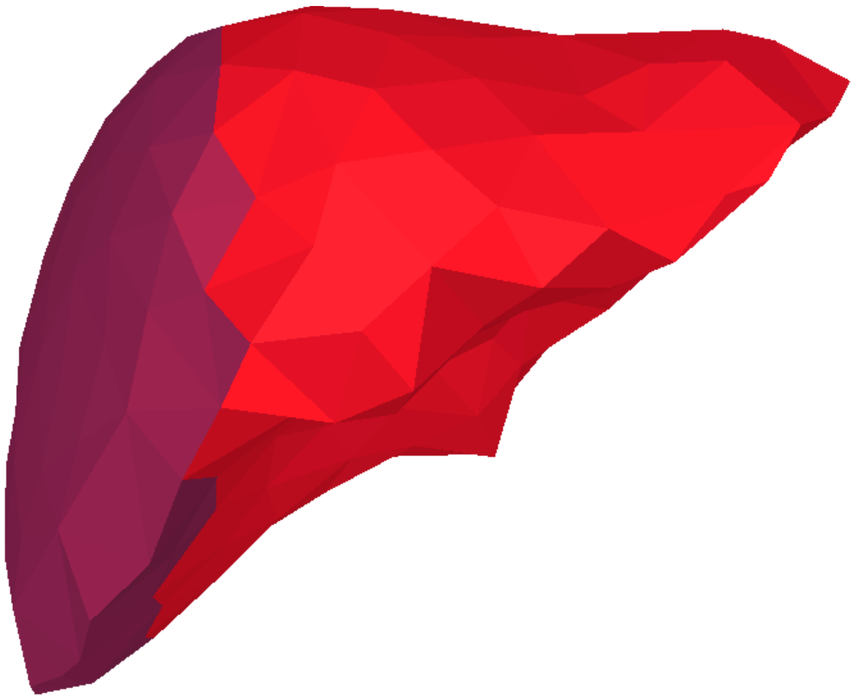

FIGURE 67. Undeformed model of the liver.

Figure 68 shows the liver model deformed under the effect of gravity. As was mentioned, the nodes on the left side of the model are fixed in space, so only the right side of the liver is affected by the gravitational force.

Figure 69 shows the liver being palpated by an implicit sphere, similar to pushing on it with a fingertip. Figure 70 shows the effect of the user palpating the liver with a cylindrical object, similar to a straight probe. This shows the effects of the implicit cylinder modeling.

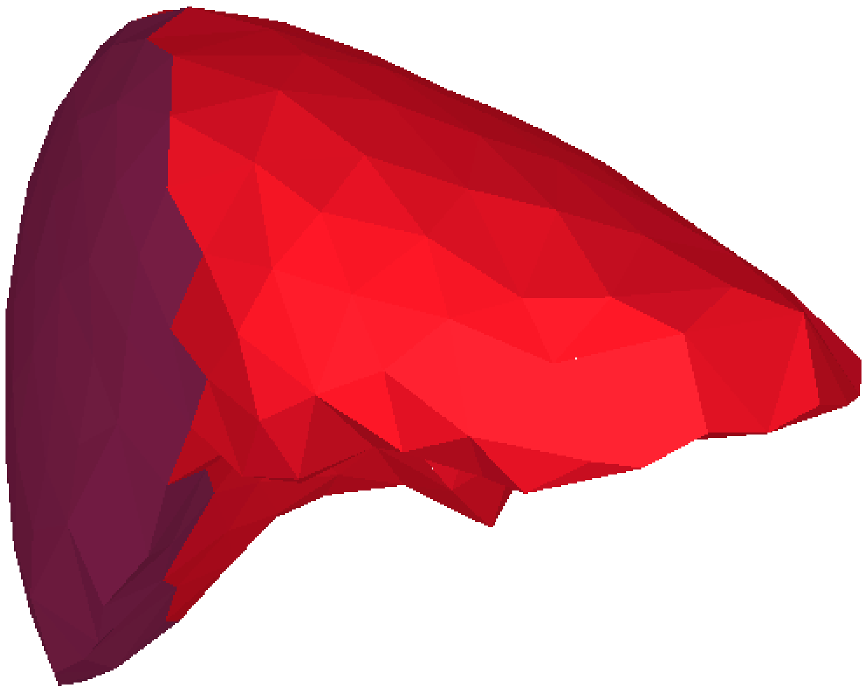

FIGURE 68. Model of the liver under the effect of gravity.

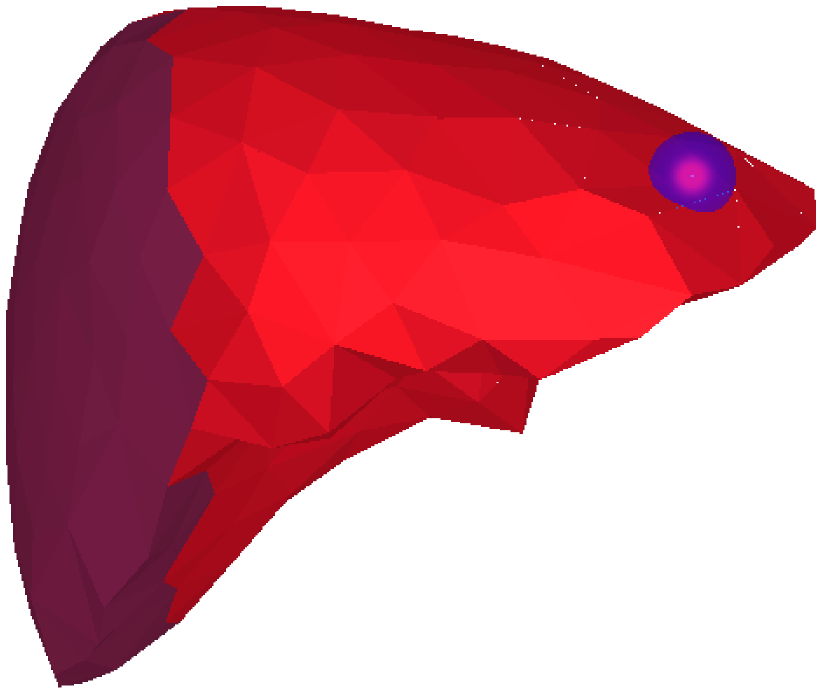

FIGURE 69. Liver model palpated by implicit sphere.

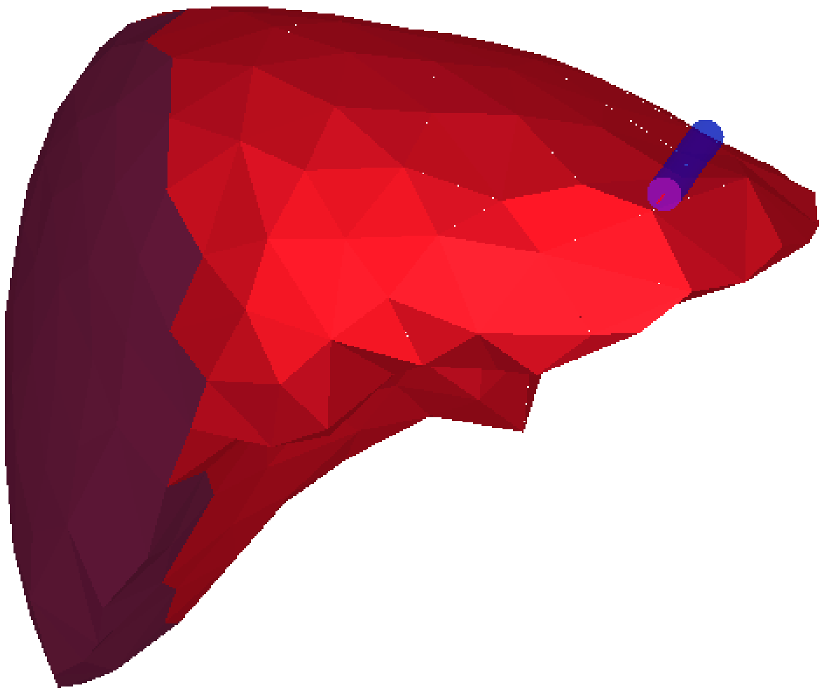

FIGURE 70. Palpating the liver model with an implicit cylinder.

The last four figures show the effects of cutting the liver model. Figure 71 and Figure 72 shows a partial cut that has been grasped by the user and pulled to open up the cut. Figure 73 and Figure 74 show the effect of a complete cut of the model of the liver.

Update Rate Considerations

Based on the results in [28], we foresee at least an increase by a factor of 2 the number of nodes than can be modeled in real-time by moving to a system based on an Intel Pentium, running at 1.0 GHz or above. Meseure and Chaillou [28] show that the computation times between a R10000 at 194 MHz and a Pentium II at 300 MHz are very similar. These results were generated on a R10000 at 250 MHz, so moving to a Pentium IV at 1.6 GHz could result in a computational increase by a factor of a 4.

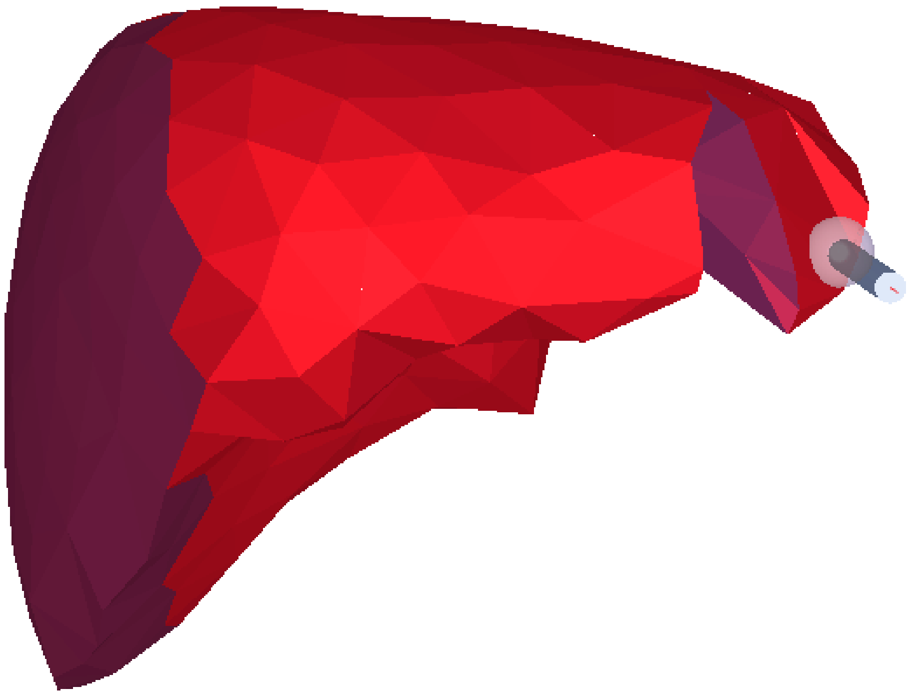

FIGURE 71. Partial cut of the liver, front view.

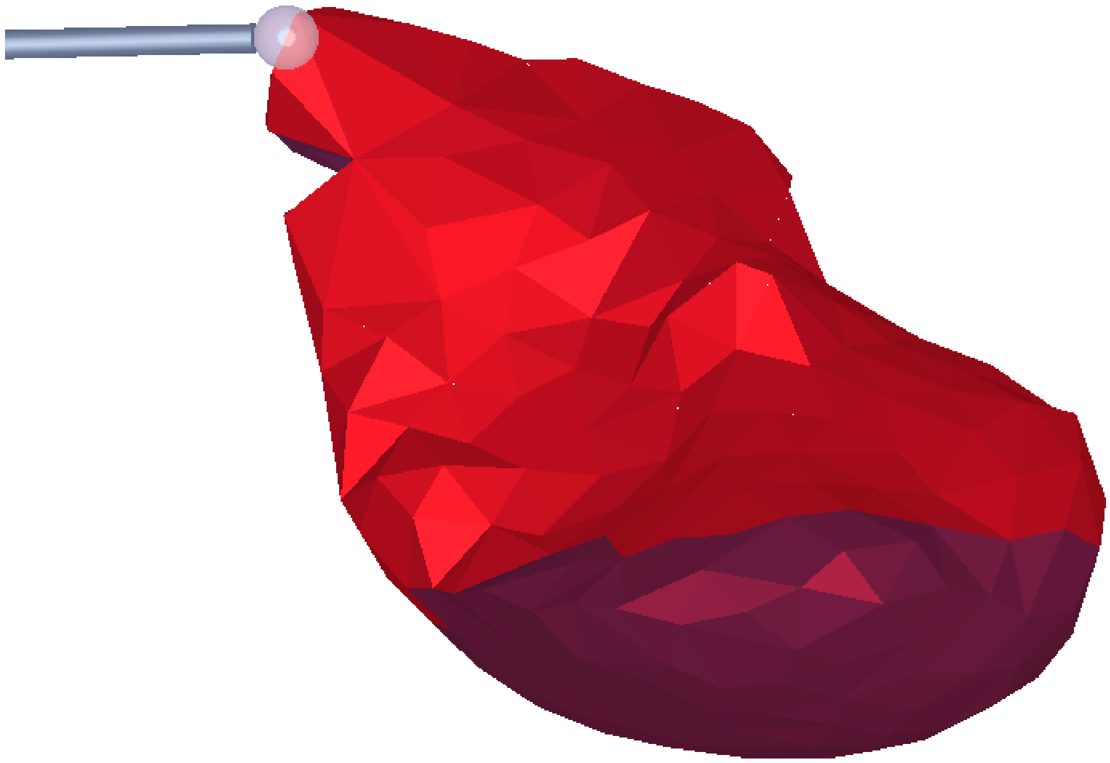

FIGURE 72. Partial cut of the liver, bottom view.

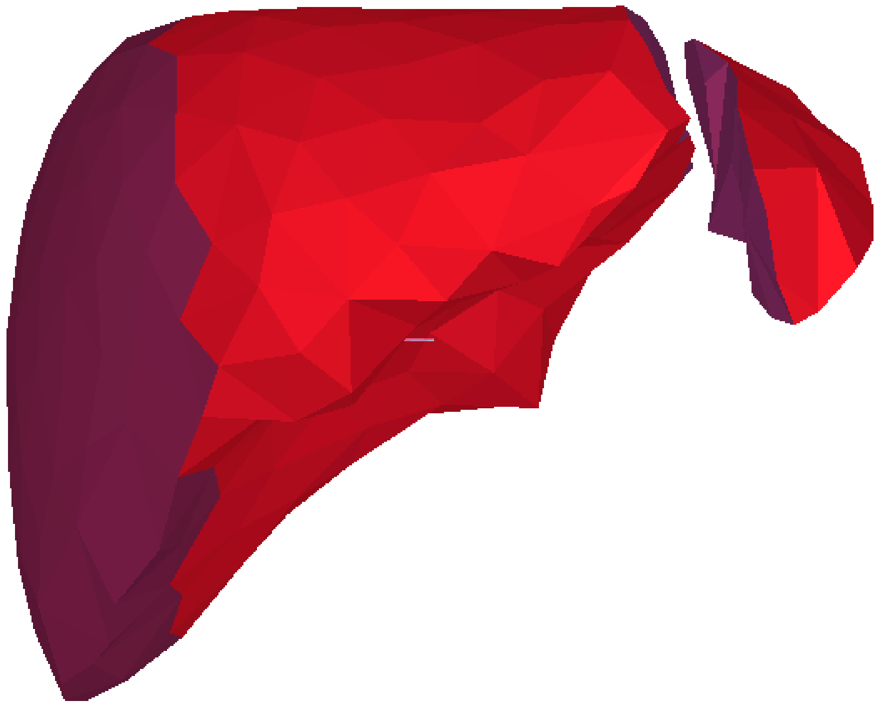
FIGURE 73. Complete cut of the liver, frontal view.

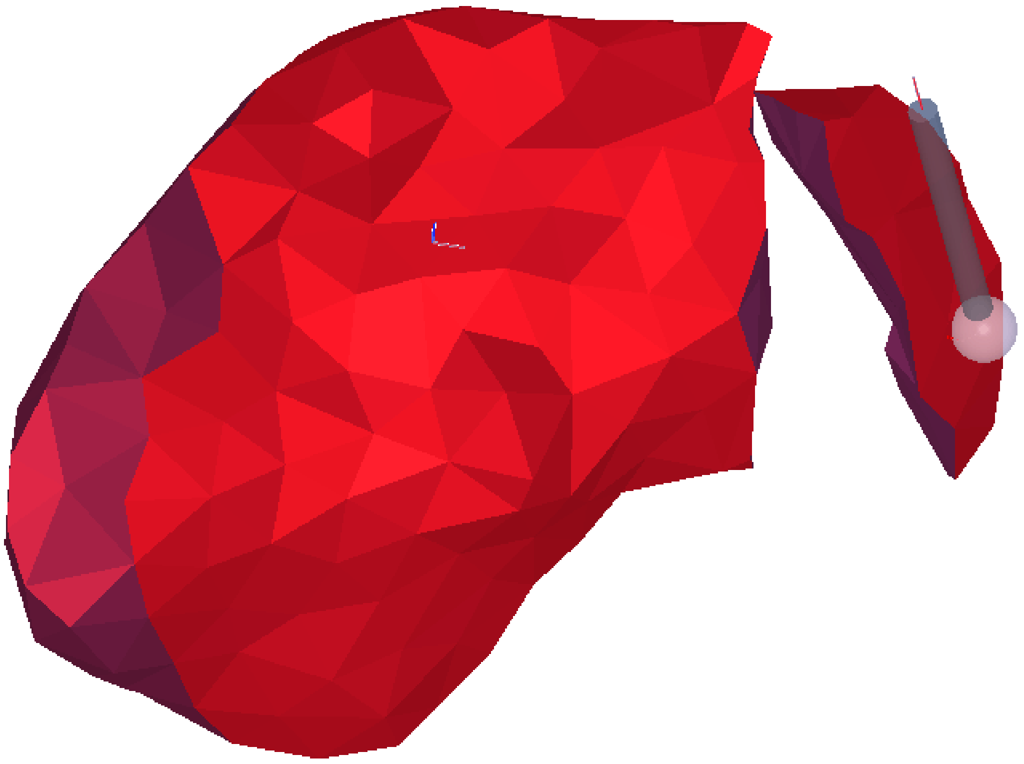
FIGURE 74. Complete cut of the liver, bottom view.

Chapter 11

Conclusions

The main goal of this thesis was to address the problem of cutting tissue within the framework of an interactive physically based soft tissue surgical simulation. Physically based linear elastic finite elements were used as a fairly simple model to generate the simulation of the soft tissue. We focused on cutting of the soft tissue as a surgical technique that occurs with great frequency but that can impact the state of the simulation a great deal. This thesis demonstrated cutting techniques to generate accurate cut surfaces that impact the computational load of the simulator as little as possible. Different methods were shown that traded off accuracy of the cut surface with stability of the resultant model. These techniques can be easily utilized in other systems as methods to model modification of any tetrahedral mesh. By demonstrating these results, we have shown that it is possible to cut through models while maintaining the accuracy of the cut, preserving the volume of the model, and maintaining the underlying efficiency and stability of the simulator.

We also demonstrated other interaction techniques required for surgical simulators: palpation, grasping, and puncture. These were all demonstrated on an interactive system utilizing both a graphical and a haptic interface.

11.1 Contributions

The main thrust of this thesis was to generate accurate cuts through models of soft tissue. The cutting of these tetrahedral models is designed to impact the total number of elements, nodes, and edges as little as possible. In reaching this goal, a method for testing the geometry of the elements was required due to limitations in the underlying soft tissue simulation utilizing a tetrahedral linear elastic finite element mode.

The cutting methods demonstrated follow the surface swept out by the user while preserving the volume of the element. The progressive cutting within elements technique models cut elements while the user is moving within these elements by creating temporary subdivisions, thereby generating as realistic updates of the model as possible. This method creates temporary subdivisions based on the current position of the cutting tool and the true intersection points on faces and edges that the cutting path has already passed through. In this way, there is no lag between the motion of the user and the updating and subdividing of the model. The cut surface within the model also accurately follows the path that the user has traced out with the cutting tool. With the temporary subdivision, the user can see the complete cut as it is created. The main drawback with this technique is that very small elements can be created right after the cutting tool passes a boundary, which can cause instability in the model. Progressive cutting between elements does not generate small temporary elements, but can create small permanent elements.

Progressive cutting between elements with snapping guarantees that the cutting routines do not create any elements small enough to cause instability. The cutting of elements lags behind the motion of the user by approximately one element length, but it does not create any unstable elements. The cut path follows the path that the user traces with the cutting tool, but does not always lie on it, due to the snapping of intersection points to maintain stability. We have demonstrated the ability to generate stable and efficient cuts through tetrahedral meshes that closely approximate the path traced out by the user.

Both of these methods were implemented in such a fashion that they generate the minimum number of new elements to fill the cut element, impacting the computational load of the simulator as little as possible. Other methods either create the maximum numbers of elements to fill up a subdivided element, or remove the intersected element completely. This method generates a minimal set of new elements to replace the cut element, while maintaining the volume of the model.

In addition to cutting, other aspects of interactive simulators were shown. Different methods for interacting with deformable models were developed. New techniques for palpation, with either an implicit model of a sphere or a cylinder, for grasping of triangles or points on the surface, and for simulating needle puncture were described. While previous methods have mainly generated forces based on a penalty method based solely on penetration depth, we looked at the volume of intersection to more accurately model the resultant forces. Lastly, a method for displaying forces based on a local model of an erratically updated simulation was also described.

We have shown a general method for cutting soft tissue within an overall interactive surgical simulator. This cutting generates an accurate cut while maintaining the stability and the efficiency of the model as much as possible.

11.2 Future Work

While we have demonstrated accurate and efficient interaction with and modification of soft tissue models, there are areas within this research and the experimental simulator which could be explored further.

The implementation of these techniques, while leading towards a realistic surgical simulator, is not yet polished enough to be used in an actual system. The cutting routines do not always handle extraordinary cases appropriately. Also, cutting through the model multiple times, for instance if the user was extending a cut, does not always work and can cause the simulator to crash. The force that is displayed to the user can behave erratically, which reduces the realistic nature of the simulator. Forces generated by the needle subroutines can also fluctuate as the needle is pulled out of the model, causing large instantaneous forces which can pull the interface handle out of the user's hand. The current simulator also only generates 3 degrees of freedom of feedback, which can be insufficient when grasping tissue, or levering between objects in the surgical field. Methods to co-locate the haptic device and the graphical image would also greatly increase the realism for practicing open techniques. For minimally invasive techniques, changing the physical setup to match the operating theater would improve the similarity between the two modalities.

The soft tissue model can be more accurately modeled with a technique other than a linear elastic finite element model. Non-linear elasticity, volume constraints, and other new methods for more accurately modeling tissue can be investigated for their applicability in a fast, real time simulator. Other methods for updating the state of the model are also possible. More efficient explicit solvers, or implicit solvers, could be investigated.

With regards to the user interface, the graphical model can be improved, using texture maps and showing surrounding tissue, for example. The haptic modeling could be improved by overlaying texture and friction onto the local haptic model. Interpolation between updates from the haptics client and other methods for improving the intermediate representation are possible.

Interaction between the user and the model can also be improved through better methods to generate forces to apply to the model. A more accurate method for calculating the projected volume of intersection for penalty methods and impulse based methods so that no penetration will actually occur during palpation are possible.

Collision detection routines can also be improved. The method for propagating cuts and interaction through the model utilizes spatial coherency to accelerate collision detection during a cut. While this did prove sufficient for the models tested, a more global method will probably be required in actual use, due to a more crowded and complex field of interest. For example, a palpation tool may actually push on two parts of a liver while holding it back. Or extending a cut may actually bring the scalpel blade into contact with two disjoint surfaces due to deformations caused by the initial cut.

Lastly, the main improvement to the cutting techniques would be to implement a snapping version of progressive cutting within elements. One possible method would be to hold an intersection at the element boundary until the user has moved far enough into the model to generate a stable, temporary subdivision. Then, once the user gets close to another boundary of the element, snap the temporary intersection forward to that boundary. Once the user passes through the boundary, generate the final subdivision as described in the thesis.

References

[1] Adachi, Y., Kumano, T., Ogion, K., “Intermediate Representation for Stiff Virtual Objects”, Proceedings IEEE Virtual Reality Annual International Symposium, pp 203-210, 1995.

[2] Baraff, D. and Witkin, D. “Large Steps in Cloth Simulation.” Proceedings of SIGGRAPH, 1998.

[3] Bashein, G., and Detmer, P. “Centroid of a Polygon.” Graphics Gems IV. Academic Press, Boston, MA, 1994.

[4] Berkelman, P., Hollis, R., Baraff, D., “Interaction with a Realtime Dynamic Environment Simulation using a Magnetic Levitation Haptic Interface Device”, Proceedings of the 1999 International Conference on Robotics & Automation, 1999.

[5] Berkley, J., et al. “Fast Finite Element Modeling for Surgical Simulation.” Medicine Meets Virtual Reality 7, 1999.

[6] Bielser, D., and Gross, M.H. “Interactive Simulation of Surgical Cuts.” Proceedings of the Eighth Pacific Conference on Computer Graphics and Applications, Hong Kong, China, 2000.

[7] Bielser, D., Maiwald, V., Gross, M.H. “Interactive Cuts through 3-Dimensional Soft Tissue.” Proceedings of the Eurographics ‘99 (Milano, Italy, September 7-11, 1999), Computer Graphics Forum, Vol. 18, No. 3, C31-C38, 1999.

[8] Bro-Nielsen, M., and Cotin, S. “Real-time Volumetric Deformable Models for Surgery Simulation Using Finite Elements and Condensation”, Proceedings of Eurographics ‘96.

[9] Bro-Nielsen, Helfrick, Glass, et al., (HT Medical), “VR Simulation of Abdominal Trauma Surgery”, Medicine Meets Virtual Reality 6, 1998.

[10] Chen, D.T. and Zeltzer, D. “Pump It Up: Computer Animation of a Biomechanically Based Model of the Muscle Using the Finite Element Method.” Computer Graphics (SIGGRAPH), num. 26, July 1992.

[11] Chen, E., et al. "Young's Modulus Measurements of Soft Tissues with Applications to Elasticity Imaging." IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control. Vol. 43, No. 1, January 1996.

[12] Cotin, S., Delingette, H., "Real-time Surgery Simulation with Haptic Feedback using Finite Elements", Proceedings of the 1998 International Conference on Robotics & Automation, Belgium, 1998.

[13] Cotin, S., Delingette, H., Ayache, N., "Efficient Linear Elastic Models of Soft Tissues for Real-Time Surgery Simulation", INRIA T.R. No. 3510, October 1998.

[14] d'Aulignac, D., Balaniuk, R., Laugier, C., "A Haptic Interface for a Virtual Exam of the Human Thigh." Proceedings of the IEEE 2000 International Conference on Robotics & Automation, San Francisco, CA, April, 2000.

[15] Debunne, G., et al. "Dynamic Real-Time Deformations Using Space & Time Adaptive Sampling." Proceedings SIGGRAPH 2001, pp. 31-36, 2001.

[16] Eberly, D., "Distance Between Point and Triangle in 3D," Magic Software, http://www.magic-software.com/Documentation/pt3tri3.pdf.

[17] Frank, A., et al. "Finite Element Methods for Real-Time Haptic Feedback of Soft-Tissue Models in Virtual Reality Simulators." Proceedings IEEE Virtual Reality 2001, Japan, March 2001.

[18] Ganovelli, F., et al. "A Multiresolution Model for Soft Objects Supporting Interactive Cuts and Lacerations." Computer Graphics Forum, Vol. 19, No. 3, pp. C271-81, 2000.

[19] Gibson, S., et al, "Volumetric Object Modeling for Surgical Simulation", Medical Image Analysis, vol. 2, num. 2, pp 121-132, 1998.

[20] Gibson, S. and Mirtich, B. "A Survey of Deformable Models in Computer Graphics." TR-97-19, Mitsubishi Electric Research Laboratories, Cambridge, MA, 1997.

[21] Gray, H. Anatomy of the Human Body. Philadelphia: Lea & Febiger, 1918. Bartleby.com, 2000. www.bartleby.com/107/. September, 2001.

[22] Green and Hatch. "Fast Polygon-Cube Intersection Testing," Graphics Gems V. Academic Press, Boston, MA, 1995.

[23] Liu, A., and Joe, B., "Relationship Between Tetrahedron Shape Measures." BIT, Vol. 34, No. 2, pp. 268-87, 1994.

[24] Keeve, E., et al. "Deformable Modeling of Facial Tissue for Craniofacial Surgery Simulation." Computer Aided Surgery, Vol. 3, No. 5, pp. 228-38, 1998.

[25] Kühnapfel, U., Çakmak, H.K., and Maaß, H., “Endoscopic Surgery Training Using Virtual Reality and Deformable Tissue Simulation.” Computer & Graphics, Vol. 24, No. 5, pp. 671-82, Oct. 2000.

[26] Massie, T, Salisbury, J.K., “The PHANToM Haptic Interface”, Proceedings of the ASME Winter Annual Meeting, 1994.

[27] Mazura, A., Seifert, S., “Virtual Cutting in Medical Data”, Medicine Meets Virtual Reality, San Diego, CA, 1997.

[28] Meseure, P., and Chaillou, C. “A Deformable Body Model for Surgical Simulation.” The Journal of Visualization and Computer Animation, Vol. 11, No. 4, pp. 197-208, September 2000.

[29] Miyazaki, Ueno, Yasuda, et al., “A Study of Virtual Manipulation of Elastic Objects with Destruction”, IEEE International Workshop on Robot and Human Communication, 1996.

[30] Möller, T. and Trumbore, B., “Fast, minimum storage ray-triangle intersection”, Journal of Graphics Tools, 2(1):21-28, 1997.

[31] Mor, A., “5DOF Force Feedback Using the 3DOF Phantom and a 2DOF Device”, Salisbury, JK and Srinivasan, MA (Eds.), Proceedings of the Third PHANToM Users Group Workshop, AI Lab Technical Report No. 1643, MIT, December 1998.

[32] Mor, A., Gibson, S., Samosky, J., “Interacting with 3-Dimensional Medical Data-Haptic Feedback for Surgical Simulation”, Salisbury, JK and Srinivasan, MA (Eds), Proceedings of the First PHANToM Users Group Workshop, MIT AI Tech Report No. 1596, 1996.

[33] Mor, A. and Kanade, T. “Modifying Soft Tissue Models: Progressive Cutting with Minimal New Element Creation.” Proceedings of Medical Image Computing and Computer-Assisted Intervention - MICCAI 2000, Vol. 1935, October, 2000.

[34] Moutsopoulos and Gilles, “Deformable Models for Laporascopic Surgery Simulation”, Computer Networks and ISDN Systems 29, pp 1675-1683, 1997.

[35] O’Brien, J. and Hodgins, J. “Graphical Models and Animation of Brittle Fracture.” Proceedings SIGGRAPH 1999, pp. 137-146, 1999.

[36] O’Toole, R., et al, “Assessing Skill and Learning in Surgeons and Medical Students Using a Force Feedback Surgical Simulator”, Proceedings of Medical Image Computing and Computer Assisted Intervention - MICCAI ‘98, Cambridge, MA, 1998.

[37] Picinbono, G., et al. “Anisotropic Elasticity and Force Extrapolation to Improve Realism of Surgery Simulation.” Proceedings of the IEEE International Conference on Robotics & Automation, San Francisco, CA, 2000.

[38] Picinbono, G, Delingette, H., and Ayache, N. “Non-Linear and Anisotropic Elastic Soft Tissue Models for Medical Simulation.” Proceedings of the IEEE International Conference on Robotics & Automation, Seoul, Korea, 2001.

[39] Press, W.H., et al. Numerical Recipes in C. The Art of Scientific Computing. Second Edition. Cambridge University Press, 1992.

[40] Popa, D., Singh, S., “Creating Realistic Force Sensations in a Virtual Environment: Experimental System, Fundamental Issues and Results”, Proceedings of the 1998 International Conference on Robotics & Automation, Belgium, 1998.

[41] Radetzky, A., et al. “Elastodynamic Shape Modeling in Virtual Medicine.” Proceedings Shape Modeling International '99, Japan, 1999.

[42] Reinig, K., “Haptic Interaction with the Visible Human”, Proceedings of the First PHANToM Users Group Workshop, MIT AI Tech Report No. 1596, 1996.

[43] Reznik and Laugier, “Dynamic Simulation and Virtual Control of a Deformable Fingertip”, Proceedings of IEEE International Conference on Robotics and Automation, Minneapolis, MN, 1996.

[44] Shoemake, K., “Arcball Rotation Control.” Graphics Gems IV. Academic Press, Boston, MA, 1994.

[45] Singh, S., et al, “Design of an Interactive Lumbar Puncture Simulator with Tactile Feedback”, Proceedings of the 1994 International Conference on Robotics & Automation, San Diego, CA., 1994.

[46] Siira, J. and Pai, D., “Haptic Texturing - A Stochastic Approach”, Proceedings of IEEE International Conference on Robotics and Automation, Minneapolis, MN, 1996.

[47] Song, G., Reddy, N., “Towards Virtual Reality of Cutting: A Feasibility Study”, Proceedings of 16th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, Baltimore, MD, 1994.

[48] Suzuki, N., et al, “Virtual Surgery System Using Deformable Organ Models and Force Feedback System with Three Fingers”, Proceedings of Medical Image Computing and Computer Assisted Intervention - MICCAI ‘98, Cambridge, MA, 1998.

[49] Swarup, N., “Haptic Interaction with Deformable Objects Using Real-Time Dynamic Simulation”, MS Thesis, MIT, 1995.

[50] Swope, W., et al. “A Computer Simulation Method for the Calculation of Equilibrium Constants for the Formation of Physical Clusters of Molecules: Application to Small Water Clusters.” Journal of Chemical Physics, Vol. 76, No. 1 pp. 637-49, 1982.

[51] Tanaka, A., Hirota, K., Kaneko, T., “Virtual Cutting with Force Feedback”, Proceedings IEEE 1998 Virtual Reality Annual International Symposium, pp 71-75, Atlanta, GA, 1998.

[52] Tarr and Salisbury, “Haptic Rendering of Visco-Elastic and Plastic Surfaces”, Second PHANToM User’s Group Workshop, 1997.

[53] Terzopolous, D., Platt, J., et al. “Elastically Deformable Models”, Computer Graphics Proceedings, Annual Conference Series, Proceedings of SIGGRAPH 87, pp 205-214, 1987.

[54] Terzopolous, D. and Waters, K. “Physically-Based Facial Modeling, Analysis, and animation”, Journal of Visualization and Computer Animation, 1:73-80, 1990.

[55] Trotts, I., et al, “Simplification of Tetrahedral Meshes”, Proceedings of Visualization ‘98, pp 287-95, North Carolina, 1998.

[56] Verlet, L, “Computer “Experiments” on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules.” Physical Review, Vol. 159, 1967.

[57] Zhuang, Y. and Canny, J. “Real-time Simulation of Physically Realistic Global Deformation.” IEEE Vis’99. San Francisco, California. October 24-29, 1999.

[58] Zienkiewicz, O., Taylor, R., <u>The Finite Element Method</u>, McGraw-Hill Book Co., London, Fourth Edition, 1988.