Real-Time Simulation of Deformable Objects: Tools and Application

Joel Brown
Stanford University
Comp. Sci. Dept.

Stephen Sorkin
Stanford University
Comp. Sci. Dept.

Cynthia Bruyns
Stanford-NASA
Biocomputation Center

Jean-Claude Latombe
Stanford University
Comp. Sci. Dept.

Kevin Montgomery
Stanford-NASA
Biocomputation Center

Michael Stephanides
Stanford-NASA
Biocomputation Center

Abstract

This paper presents algorithms for animating deformable objects in real-time. It focuses on computing the deformation of an object subject to external forces and detecting collisions among deformable and rigid objects. The targeted application domain is surgical training. This application relies more on visual realism than exact, patient-specific deformation, but requires that computations be performed in real-time. This is in contrast with pre-operative surgical planning, where computations may be done off-line, but must provide accurate results. To achieve real-time performance, the proposed algorithms take advantage of the facts that most deformations are local, human-body tissues are well damped, and motions of surgical instruments are relatively slow. They have been integrated into a virtual-reality system for simulating the suturing of small blood vessels (microsurgery).

1. Introduction

Real-time simulation of deformable objects is needed in many areas of graphic animation, for example to generate cloth motions in interactive video games and to provide realistic graphic rendering of soft human-body tissues in surgical training systems. Deformable objects raise a complex combination of issues ranging from estimating mechanical parameters, to solving large systems of differential equations, to detecting collisions, to modeling responses to collisions. See [10] for problems and techniques in cloth modeling and [8] for issues arising in surgical simulation. Many issues still lack adequate solutions, especially when simulation must be real-time.

Here, we focus on two main problems: (1) computing the deformation of a viscoelastic object subject to external forces, and (2) detecting collisions among deforming and rigid objects. Our main goal is to develop efficient data structures and algorithms that can process large models at a rate compatible with real-time graphic animation (30 Hz). To achieve this goal, we exploit the fact that many deformations are *local*. By propagating forces in a carefully ordered fashion through an elastic mass-spring mesh, we effectively limit the computations to the portions of objects that undergo significant deformations. To accelerate collision detection, we pre-compute hierarchical representations for all objects in the scene; when objects are being deformed, we only update those parts of the hierarchies that need to be modified.

We have used these algorithms, along with other techniques, to build a system for microsurgical training. Microsurgery involves the repair of approximately 1mm vessels and nerves under a microscope. Using a forceps, the surgeon maneuvers a suture (needle and thread) through the two ends of a severed vessel and ties several knots to stitch the two ends together. The two parts of the vessel undergo deformations caused by their interactions with the suture and the forceps. The surgeon receives only visual feedback, as the vessel is too small to produce any perceptible reaction force. Microsurgeons acquire their initial skills through months of practice in an animal lab. Goals of our system include a decrease in training time, objective evaluation of the training, and an alternative to using lab animals. The need for suture simulation has been previously addressed in [8, 19].

Sections 2 and 3 present our simulation and collision-detection algorithms. Section 4 describes the microsurgical system. Section 5 discusses current and future work.

2. Computation of Object Deformations

Research on modeling deformable objects has increased dramatically in the past few years. Most proposed 3D models fall into two broad categories, *mass-spring meshes* and

finite elements. A mass-spring mesh is a set of point masses connected by elastic links. It represents the tissue geometry and is used to discretize the equations of motion. Mass-spring models have been used in facial animation [25], cloth motion [3], and surgical simulation [11], to cite only a few works. They are relatively fast and easy to implement, and allow realistic simulation for a wide range of objects, including viscoelastic tissues encountered in surgery.

Finite element models (FEMs) use a mesh to decompose the domain over which the differential equations of motion are solved, but do not discretize these equations. The mesh represents the domain initially occupied by the object and the FEM technique computes a vector field representing the displacement of each point in this domain. For example, FEMs have been used to model facial tissue and predict surgical outcomes [13, 15, 22]. They may be more accurate than mass-spring models, but they are more computationally intensive, especially for complex geometries and large deformations. Some systems use either mass-spring or FEM techniques depending on the situation [16]. Others use preprocessing steps to reduce FEM computation [4, 7], and [21] extends the "tensor-mass" model of [7] to nonlinear elasticicty. Other examples of mass-spring models, FEMs, and alternate models are too numerous to cite here.

Mass-spring meshes seem better suited for surgical training – the application domain considered in this paper – which relies more on visual realism than exact, patient-specific deformation, but requires that simulations be performed in real-time. In contrast, FEMs may address better the needs of other applications (e.g., pre-operative surgical planning and predicting the long-term outcome of a surgery), where computations can be done off-line, but must provide accurate, patient-specific results.

2.1. Mass-spring elastic mesh

We represent the geometry of a deformable object by a 3D mesh M of n nodes N_i (i = 1,...,n) connected by links L_{ij}, $i, j \in [1, n], i \neq j$. Each node maps to a specific point of the object, so that the displacements of the nodes describe the deformation of the object. The nodes and links on the object's surface are triangulated, whereas the other nodes and links are unrestricted, though it is often convenient to arrange them in a tetrahedral lattice. Figure 1a shows the surface and underlying links of a mesh representing a severed blood vessel; this mesh contains 98 nodes and 581 links. The more complex mesh in Figure 1b consists of 40,889 nodes and 212,206 links forming a tetrahedral lattice.

The mechanical properties (viscoelastic, in most surgical simulation applications) of the object are described by data stored in the nodes and links of M. A mass m_i and a damping coefficient c_i are associated with each node N_i, and a stiffness k_{ij} is associated with each link L_{ij}. The internal force between two nodes N_i and N_j is $\boldsymbol{F}_{ij} = -k_{ij}\Delta_{ij}\boldsymbol{u}_{ij}$, where $\Delta_{ij} = l_{ij} - rl_{ij}$ is the current length of the link minus its resting length, and $\boldsymbol{u}_{ij}$ is the unit vector pointing from N_i toward N_j. The stiffness k_{ij} may be constant or function of Δ_{ij}. In either case, $\boldsymbol{F}_{ij}$ is a function of the coordinate vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ of N_i and N_j. This representation can describe objects that are nonlinear, non-homogeneous, and anisotropic. We typically initialize the parameter values using available biomechanical data and tune them based on comments given by surgeons interacting with our models.

At any time t, the motion/deformation of M is described by a system of n differential equations, each expressing the motion of a node N_i:

$$m_i\boldsymbol{a}_i + c_i\boldsymbol{v}_i + \sum_{j\in\sigma(i)} \boldsymbol{F}_{ij}(\boldsymbol{x}_i, \boldsymbol{x}_j) = m_i\boldsymbol{g} + \boldsymbol{F}_i^{ext} \quad (1)$$

where $\boldsymbol{x}_i$ is the coordinate vector of N_i, $\boldsymbol{v}_i$ and $\boldsymbol{a}_i$ are its velocity and acceleration vectors, respectively, $m_i\boldsymbol{g}$ is the gravitational force, and $\boldsymbol{F}_i^{ext}$ is the total external force applied to N_i. $\sigma(i)$ denotes the set of the indices of the nodes adjacent to N_i in M.

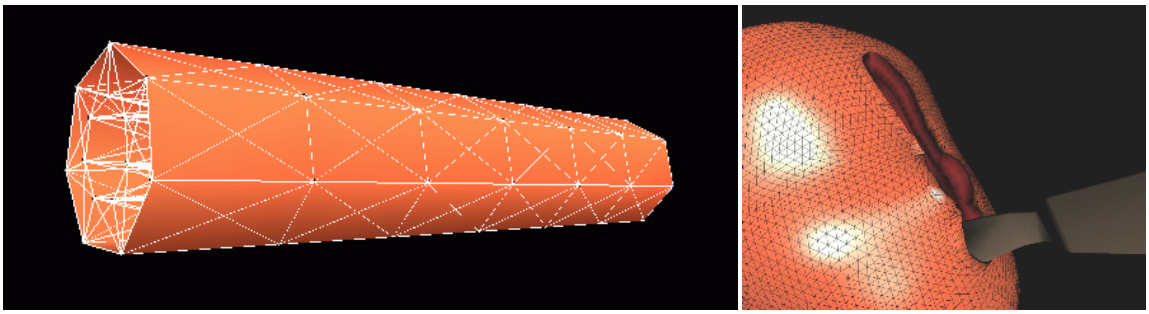
2.2. Simulation algorithm

We have developed a "dynamic" and a "quasi-static" simulator. The dynamic simulator uses classical numerical integration techniques such as fourth order Runge-Kutta to solve Eq. 1. However, in many situations encountered in surgical simulation, a simpler algorithm based on quasi-static assumptions gives realistic results at a much faster rate. We describe this quasi-static simulator below.

Assumptions. We refer to the nodes of M that are subject to external forces as the *control* nodes. We assume that the position of each such node is given at any time. In our surgical simulation system, the control nodes correspond to the portions of tissue that are pulled or pushed by surgical instruments or held fixed by bone structures or clamping tools. The positions of the displaced control nodes are obtained online by reading the positions/orientations of tracking devices. We also assume that the velocity of the control nodes is small enough so that the mesh achieves static equilibrium at each instant. This is a reasonable assumption for soft objects with relatively high damping parameters, which is the case for most human-body tissues. When these assumptions do not hold, the dynamic simulator must be used.

Quasi-static algorithm. Under the above assumptions, we neglect dynamic inertial and damping forces. The shape of M is defined by a system of equations expressing that each non-control node N_i is in static equilibrium:

$$\sum_{j\in\sigma(i)} \boldsymbol{F}_{ij}(\boldsymbol{x}_i, x_j) - m_i\boldsymbol{g} = 0. \quad (2)$$



(a) (b)

Figure 1. Mesh examples

Let I be the set of indices of all the non-control nodes of M, and let δ be a constant time step (in our implementation δ is set to 1/30 s). At each time $t = k\delta$, $k = 1, 2, ...$, the quasi-static simulator solves Eq. 2 for the positions of all the non-control nodes. To achieve real-time animation, it returns these positions within time δ. The algorithm is the following:

Algorithm QSS:
1. Acquire the positions of all the control nodes
2. Repeat until time δ has elapsed
   For every $i \in I$
   (a) $\boldsymbol{f}_i \leftarrow \sum_{j \in \sigma(i)} \boldsymbol{F}_{ij} - m_i \boldsymbol{g}$
   (b) $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i + \alpha \boldsymbol{f}_i$

Step 2 computes the residual force applied to each node and displaces the node along this force. A conjugate-gradient-style method can also be used by moving the node along a combination of the old and the new forces. Ideally, the value of the scaling factor α should be chosen as large as possible such that the iteration converges. This choice typically requires experimental trials.

The timeout condition of Step 2 guarantees that QSS operates in real-time even as the size of the mesh M increases. Hence, Step 2 is not guaranteed to reach exact equilibrium at every step, that is, some N_i's will have a non-zero force $\boldsymbol{f}_i$ acting on them after δ amount of time. As mesh size increases, each iteration of Step 2 will take longer, and thus fewer loops will be possible in the allowed time. By comparing the positions computed by QSS to the actual equilibrium positions (computed without timeout), we can measure how the accuracy of the simulation degrades as the mesh complexity increases.

Step 2(b) updates the position of each non-control node using the most recently computed positions of the adjacent nodes, rather than those computed at the previous iteration of Step 2. This scheme is most advantageous when the nodes are processed in a wave-propagation order starting at the displaced control nodes and expanding towards the

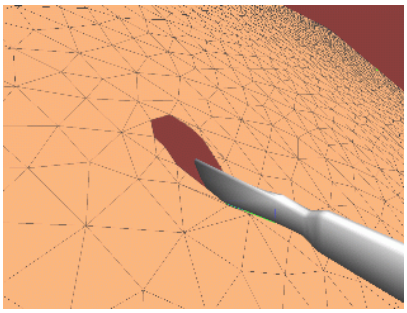


Figure 2. Simulation of a cutting operation

nodes farthest away from any displaced node. This ordering is computed by a breadth-first scan of the mesh M:

Algorithm NODE-ORDERING:
1. Initialize I to the empty list
2. Mark the displaced control nodes in M to be at level 0
3. For $k = 1, 2, ...$, mark the unmarked nodes adjacent to a node at level $k - 1$ to be at level k, and store them in I, until all non-control nodes have been marked.

The displaced nodes may be arbitrarily distributed over the mesh. The outcome of NODE-ORDERING is a list I of nodes such that if index i appears before index j in I then the level of N_i is less than or equal to that of N_j. QSS processes the nodes as they have been ordered in I.

Node ordering enables another major computational savings. During an iteration of Step 2, if the positions of all the nodes at some level k are modified by less than a small pre-specified amount, then the algorithm stops propagating the deformation further. In this way, the number of levels treated at each iteration adjusts automatically. This computation *cutout* is especially useful when object deformations are local.

While QSS is being executed, any change in the set of displaced nodes only requires re-invoking NODE-ORDERING to compute a new ordered set I. Similarly, the mesh's topology may change at any time. For example, in

Figure 2 a cutting operation is being performed. Links are removed from the mesh as they are crossed by the scalpel, so that the mesh used by QSS changes frequently. (In principle, such a cutting operation violates assumptions made above, and a dynamic simulator should be used. Nevertheless, QSS, which was used to produce Figure 2, gives realistic results.)

Performance evaluation. The above algorithms are written in C++ and run on a 450 MHz processor on a Sun Ultra 60 workstation with 1 GB RAM. To address visual realism, we asked surgeons to verify that the deformations were similar in shape and velocity to those encountered in clinical operations. Figure 6 shows such deformations.

We did other experiments to quantitatively evaluate the performance of QSS. In particular, we created a regular mesh whose nodes form a rectangular lattice. Each node is linked to its neighbors in the x, y, and z directions. It is also diagonally linked to neighbors in the xy, xz, and yz planes. All links are given the same constant stiffness. Our meshes ranged from a 3x3x3 box (27 nodes and 64 links) to a 20x20x20 box (8000 nodes and 66120 links).

Given one such mesh, we fix the nodes of the bottom face, and displace the middle node of the top face upwards by one unit. We then run QSS, but instead of running Step 2 for a fixed amount of time, we do a fixed number k of iterations ($k = 100$, 50, 20, 10, or 5), where an iteration involves updating $\boldsymbol{f}_i$ and $\boldsymbol{x}_i$ for all the non-control nodes. We repeat several times this cycle of displacing the control node (by an additional unit) and doing k iterations, and after each cycle we record the position of each node. Errors are computed as the distances between these positions and the actual equilibrium positions. The equilibrium positions are found by running QSS until the force on each non-control node is zero.

Figure 3a shows maximum and average errors for the different mesh sizes for 5 consecutive cycles of unit displacement followed by $k = 10$ iterations. We see that errors increase slightly as the box grows from 27 to 216 nodes, but then minimally as size increases to 8,000 nodes. The larger boxes have many nodes which may be moving hardly at all, which contribute to lowering the average error. However, the maximum error follows the same pattern at about twice the average error, and remains at most about 10% of the magnitude of the displacement of the control node for all mesh sizes.

Figure 3b shows maximum error after one unit displacement for different numbers k of iterations. It compares QSS with and without the computation cutout described previously. The plots for two different boxes are shown together, and we can see that errors are quite low in all cases, even for the largest box and the lowest number of iterations. When using the cutout, errors do not strictly drop off to 0 as the number of iterations increases, but the more important difference is the sharp decrease in the time required per iteration. Table 1 displays the number of iterations that QSS performs while maintaining a 30 Hz update rate, without and with cutout.

These experiments demonstrate that for the objects given, we can reasonably maintain a small relative error given only 10 or fewer iterations of QSS Step 2 per time interval. Furthermore, the cutout method can complete these iterations in the appropriate 1/30 second, and scale to meshes containing thousands of nodes with no significant performance penalty.

3. Collision Detection

Research on collision detection between rigid objects has a long history in robotics, graphics, and solid modeling. Two main families of methods have been proposed: *feature-based* (e.g. [1, 17, 18]) and *hierarchical* (e.g. [6, 9, 14, 20, 23]). A feature-based method exploits temporal and spatial coherence in the geometric model to maintain the pair of closest features. A hierarchical method pre-computes a hierarchy of bounding volumes for every object. During a collision test, the hierarchies are used to quickly discard large subsets of the object surfaces that are too far apart to possibly collide. Hierarchies using various primitive volumes have been proposed. While some volumes allow closer-fit approximation, they also yield more costly intersection checks. Spheres give good results over a broad range of objects.

Although each approach has distinct advantages, the hierarchical approach is better suited when objects are highly concave. The main issue in using it with deformable objects is that pre-computed hierarchies may become invalid when objects deform, while re-computing new ones at each collision query would be too time consuming. Below, we propose an algorithm that does not modify the topology of the hierarchy representing an object, but only updates the size and location of the primitive volumes labeling the nodes of this hierarchy. Our algorithm derives from the one proposed by Quinlan [23] for rigid objects.

Fewer works exist for deformable objects (e.g. [2, 26, 11, 24, 27]). The algorithm in [26] is the closest to ours. It also builds a tree of fixed topological structure. But this tree is not balanced (which may seriously increase the cost of collision queries) and its nodes are AABBs. The main difference, however, is in the tree-maintenance algorithm. Unlike [26], our algorithm exploits the locality of most deformations to minimize the number of node updates. The algorithm in [27] is specifically aimed at detecting self-collisions of cloth-like objects, an issue that we have not carefully studied so far.

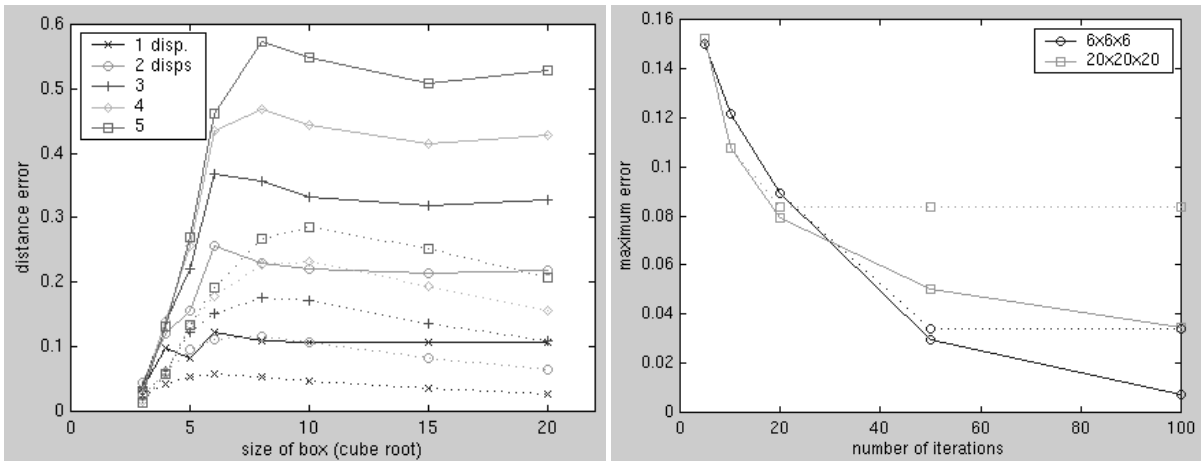


Figure 3. (a) Maximum (solid lines) and average (dashed lines) error vs box size (b) Maximum error without (solid lines) and with (dashed lines) computation cutout vs number of iterations

Mesh	Nodes	Edges	Iterations at 30 Hz without cutout	Iterations at 30 Hz with cutout
6x6x6	216	1440	24	24
8x8x8	512	3696	9.5	13
10x10x10	1000	7560	4.6	8
15x15x15	3375	27090	1.2	7
20x20x20	8000	66120	0.4	6

Table 1. Effect of computation cutout on simulation rate

3.1. Quinlan's algorithm

Sphere tree of an object. Let A be a (rigid) object represented by its triangulated surface S. Quinlan's algorithm covers every triangle in S with small spheres of predefined radius ε and constructs an approximately balanced binary tree T that has one leaf per sphere of radius ε. Each other node N in T is a sphere that encloses all the leaf spheres of the sub-tree rooted at N. T is constructed by recursively partitioning the set E of leaf spheres contained in a sub-tree (initially the set of all leaf spheres in T) into two subsets E_1 and E_2 of equal cardinality, until each subset contains a single leaf sphere. The partitioning operation tries to minimize the intersection and the radii of the two spheres that respectively enclose the leaf spheres in E_1 and E_2. A technique to partition the set E first computes the box that is aligned to the object's coordinate frame and contains the centers of the leaf spheres in E. It then divides the leaf spheres along the longest side of this box.

Collision detection. Let T_1 and T_2 be the respective sphere trees of two (rigid) objects A_1 and A_2. A collision query is specified by the position and orientation of A_1 relative to A_2. Collision detection is performed by a depth-first traversal of T_1 and T_2 during which pairs of spheres from the two trees are examined. If two intermediate spheres have null intersection, then the leaf spheres they contain cannot possibly intersect, and the traversal is pruned; otherwise the children of one of the two nodes are examined. If two leaf spheres intersect, the two triangles tiled by these spheres are explicitly tested for collision. For N_1 and N_2, the root spheres of T_1 and T_2, respectively, the following algorithm returns 1 if it detects a collision, and 0 otherwise:

Algorithm COLLISION(N_1, N_2):
1. If N_1 and N_2 have null intersection then return 0
2. Else
   (a) If both N_1 and N_2 are leaf spheres then test the corresponding two triangles for collision; return 1 if they collide and 0 otherwise
   (b) If N_2 is smaller than N_1 then switch N_1 and N_2
   (c) If COLLISION(N_1,left-child(N_2)) = 1 then return 1
   Else if COLLISION(N_1,right-child(N_2)) = 1 then return 1
   Else return 0

3.2. Application to deformable objects

To use COLLISION, we must maintain the sphere tree of every deforming object. We propose a new sphere tree whose balanced structure is computed only once. When an object deforms, the structure of its tree remains fixed, i.e., no sphere is ever added or removed; only the radii and positions of some spheres are adjusted. Moreover, the maintenance algorithm performs adjustments only where they are needed.

Construction of a sphere tree. Let S be the triangulated surface of a deformable object A in some initial shape. The pre-computed tree T for A differs from the one in [23] in two ways:

(1) Instead of tiling the triangles of S with small equal-sized spheres, we assign each triangle a single leaf sphere of T – the smallest sphere enclosing the triangle. Hence, when S undergoes a deformation, the number of leaf spheres of T remains constant. Moreover, updating the radius and position of the sphere enclosing a deforming triangle is faster than computing a new tiling.

(2) The approximately balanced structure of T is still generated by recursively partitioning the leaf spheres into two subsets of equal size. But the radius and position of each non-leaf sphere is computed to enclose the sphere's two children. This yields a slightly bigger sphere than the one computed to contain the descendant leaf spheres, but the computation is much faster.

Collision detection. COLLISION is used unchanged.

Maintenance of a sphere tree. Each deformation of one triangle of S requires adjusting the radius and position of the corresponding leaf sphere and of all its ancestors up to the root of T. Our algorithm performs those changes only prior to processing a query. The operation is done bottom-up, using a priority queue Q of spheres sorted by decreasing depths in T. Q is initialized to contain all the leaf spheres that enclose triangles that have been deformed since the last update of T. It is then used as follows:

Algorithm MAINTENANCE:
While Q is not empty do
1. $w \leftarrow$ extract(Q)
2. Adjust the radius and position of w
3. Insert(Q,parent(w))

The only spheres that are modified are those that contain at least one deformed triangle. Each such sphere is modified only once, even if it contains several deformed triangles.

Clearly, MAINTENANCE will perform better when deformations are local than when they are scattered throughout S. More specifically, a local deformation of S affecting $k \ll s$ triangles, where s is the total number of triangles in S, results in a total update time of O($k + \log s$). Instead, if the k triangles are spread over the leaves, this cost can be O($k \log s$). However, in the worst case, if many triangles have changed shape, the maintenance operation only takes time O(s).

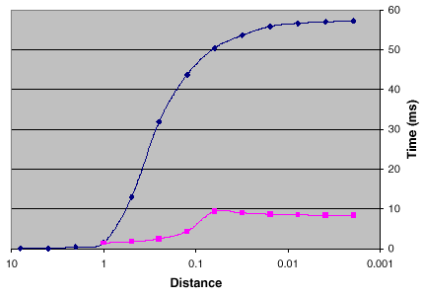


Figure 4. Collision-detection performance

3.3. Performance evaluation

The above algorithms were implemented in C++. We give results of experimental performance tests on an Intel 400-MHz Pentium II processor, with 256-MB memory and running Windows 2000.

Sphere tree construction. The pre-computation of an object's sphere tree need not be particularly efficient, since it is done only once per object, prior to any simulation. Our software runs in time proportional to the number of triangles and takes on the order of 0.1 milliseconds per triangle.

Sphere tree maintenance. To evaluate MAINTENANCE we considered a surface S initialized to a flat horizontal 100x100 grid, with each square split into two triangles. Hence, S consists of 20,000 triangles. To create a local deformation of S, we pick a vertex V, a radius ρ of deformation between 1 and 10, and a direction (upward or downward), all at random. Each vertex within distance ρ of V is translated by a distance inversely proportional to its distance to V. To create scattered deformations of S, we repeat this process several times. After computing the sphere tree for the flat surface, we ran MAINTENANCE to update this tree after various deformations. We measured a running time for MAINTENANCE of 0.06 milliseconds per deformed triangle.

Collision queries We considered two objects. One is modeled as a flat square mesh of 8 by 8 units tessellated with 8,192 triangles. The other object is a spherical ball, 2 units in diameter and tessellated with 1,024 triangles. We moved the ball along a straight path though the center of the square mesh, from 64 units separation to one unit penetration. The query times for different relative positions of the objects are shown in Figure 4. When the objects are far apart, each query is extremely fast and takes on the order of tenths of a millisecond. When they get closer than 1 unit together, query time grows quickly (dark curve) to an

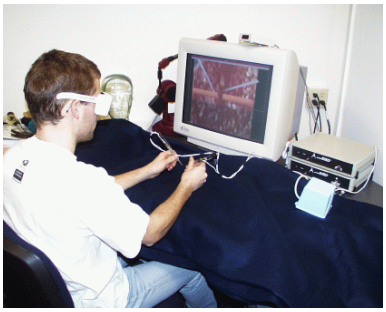


Figure 5. Setup for microsurgery

asymptote of just under 60 milliseconds, as more spheres in the trees had to be examined to rule out a possible collision. Once the objects are in collision, the query time drops sharply to under 10 milliseconds (grey curve). This sharp drop suggests that a timeout could be imposed on COLLISION, with a relatively minor risk of not detecting a collision once in a while. We did similar experiments after deforming the two objects and updating their sphere trees using MAINTENANCE. We observed no significant degradation of the query times, even for rather large deformations of the objects.

4. Microsurgical Simulation System

System Overview. Figure 5 shows a user interacting with our system, by manipulating real surgical instruments (here, forceps) mounted on electromagnetic tracking devices. The positions/orientations of the trackers are read at 100 Hz, and the opening and closing of the forceps is obtained online by reading a sensor. The instruments are rendered on the graphic display, along with the deformable objects (in microsurgical simulation, a severed blood vessel and a suture made of a needle and thread). The graphic display is updated at 30 Hz. The user has complete control of the viewpoint and may use stereo glasses.

The simulator QSS of Section 2 computes the deformations of the vessel, while the suture's deformations are computed as described below. All collisions are handled using the algorithm of Section 3. The most frequent collisions are between the forceps and the vessel, the needle and the vessel, the thread and the vessel, the thread and the forceps, the thread with itself, and one half of the vessel with the other. Our implementation on a dual-processor machine (Sun Ultra 60, two 450 MHz processors) allows the processing of deformations and collision detection to not conflict with the rendering.

Models. Each half of the vessel is modeled as a truncated double-hulled cylinder (Figure 1a). The inner and outer cylinders are modeled by several layers of nodes, with the layers evenly spaced, and each layer consisting of several nodes evenly spaced around a circle. Each node is connected by deformable links to its neighbors within a layer and in neighboring layers. There are also connections between the nodes of the inner and outer cylinders. One end layer of each vessel is fixed in space, representing the fact that the vessels are clamped down during surgery, and only a portion of their length can be manipulated. When the two parts of the vessel touch one another, their meshes are merged (we assume sticking contact) and QSS computes the deformation of this new mesh.

Simulation of the suture. The thread is deformable, but not elastic. We model the suture as an articulated object with 200 short straight links sequentially connected at nodes acting as spherical joints. Any node of the suture may be constrained by another object. For example, one node might be grasped by a forceps, and thus its position is constrained by the forceps. If the suture has pierced through a vessel, a node will be constrained by the position of the vessel. The motion of the suture is calculated using a "follow-the-leader" technique presented in [5]. Pulling on one or both ends of the suture always results in deforming the thread, except when there is no remaining slack and the suture has pierced through the vessel; then the displacement causes the vessel to deform.

Examples of interactions. Figures 6a and 6b show deformations generated by forceps holding one or two parts of the blood vessel. In Figures 6c and 6d the suture is pierced through a vessel, and pulls down first on one end of the vessel and next on both ends to bring them together. Figures 6e and 6f show an example where the suture is wrapped around a forceps and another where it collides with itself and the vessel. In these examples, the full set of computations (deformation, collision detection, tool simulation, etc.) is done at 30 Hz. Note the diversity of deformations achieved by the vessels.

5. Conclusion and Future Work

We have designed new fast algorithms for simulating the deformations of soft objects and detecting collisions among deforming and rigid objects. These algorithms take advantage of several characteristics of surgical training: (1) visual realism is more important than accurate, patient-specific simulation; (2) most deformations are local; (3) human-body tissues are well damped; and (4) surgical instruments have relatively slow motions. Our simulator exploits these characteristics to solve quasi-static equations using a "wave-propagation" technique that has an automatic computation cutout when deformations become insignificant. The collision algorithm exploits deformation locality to minimize the number of updates in the hierarchical representations of the deforming objects.

These algorithms have been integrated into a virtual-

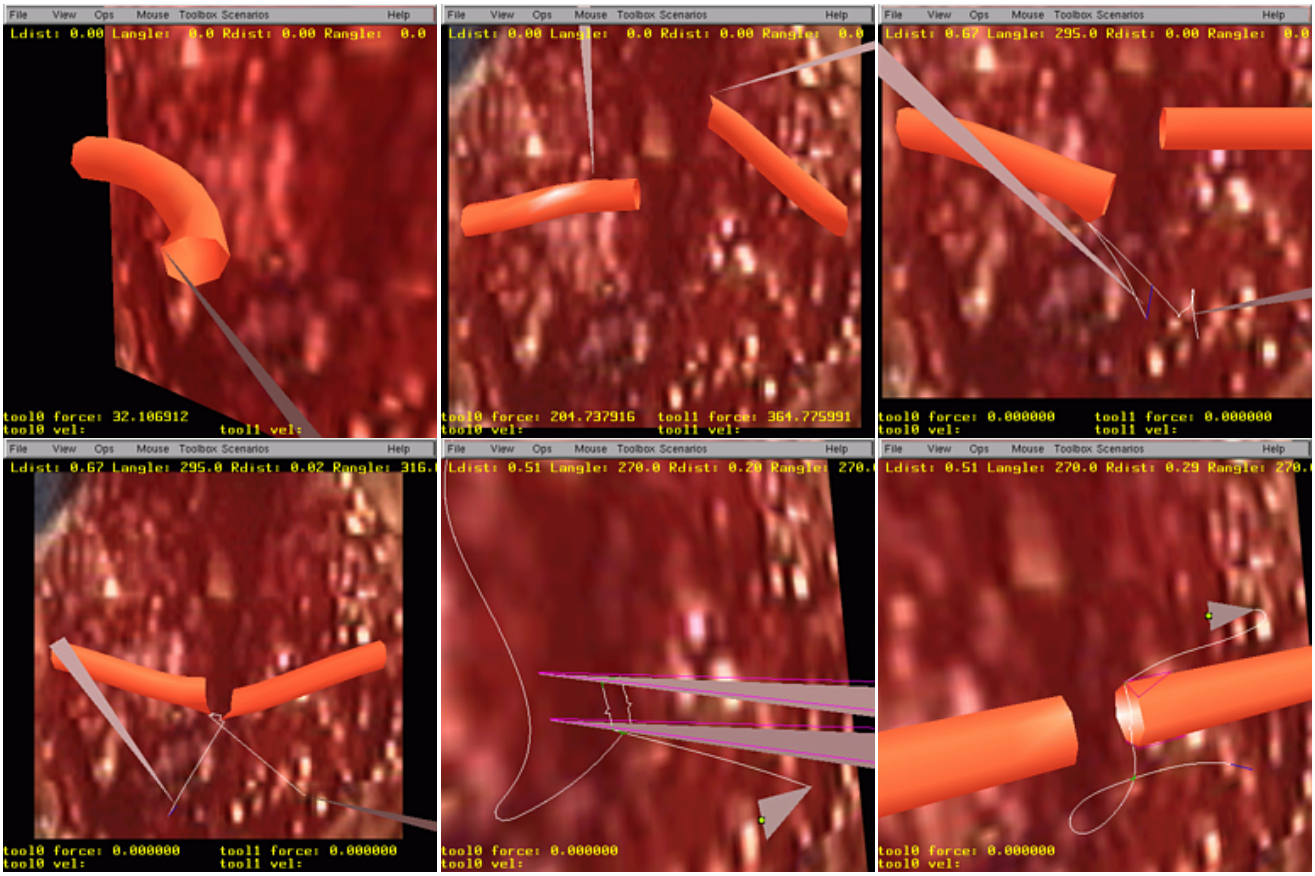


Figure 6. (a),(b) Forceps deforming severed vessel, (c) Suture pulling vessel down, (d) Suture pulling two vessels together, (e) Suture wrapped around forceps, (f) Suture colliding with self and vessel end

reality system for simulating the suturing of small blood vessels. This system has been used by plastic and reconstructive surgeons in our lab and at various exhibits, and deemed realistic and potentially very useful. Our next step is experimental and clinical verification, by having surgeons who are learning the procedure use this tool, and assessing the quality of their virtual repairs through measurements such as angle and position of vessel piercing. We will then try to establish quantitatively how practicing with the simulator affects future quality of real vessel repairs.

We are also investigating other surgical applications. While force feedback is irrelevant in microsurgery, it is critical in many other applications [8]. QSS can compute the force applied to each displaced control node in an elastic mesh. But it does not achieve an update rate compatible with haptic interaction (roughly 1000 Hz). To connect our simulator to haptic devices, we are developing fast techniques to interpolate between forces computed by QSS. The elastic mesh model does not allow the explicit representation of incompressibility constraints often encountered in human-body tissues. A technique proposed in [12] to overcome this limitation is to apply artificial corrective forces to surface nodes to keep the object's volume approximately constant. Extending our collision-detection module to efficiently detect collisions of an object with itself is another short-term goal.

There are many other issues to consider, such as the detection of mesh degeneracies (e.g., when a link crosses another) and the modeling of collision responses. As more issues are addressed in a simulation system, more algorithms will run concurrently, and their efficiency will become even more critical.

Acknowledgments: This research was conducted in the Stanford-NASA Biocomputation Center. It was supported by grants from the National Aeronautics and Space Administration (NAS-NCC2-1010), the National Science Foundation (IIS-9907060), and the NIH National Libraries of Medicine (NLM-3506). It has also benefited from equipment gifts made by Sun Microsystems and Intel. This paper was improved by discussions with Frédéric Mazzella. Yanto Muliadi produced some of the performance data for the collision-detection algorithms.

References

[1] D. Baraff. Curved surfaces and coherences for non-penetrating rigid body simulation. *Computer Graphics*, 24(4):19–28, 1990.

[2] D. Baraff and A. Witkin. Dynamic simulation of non-penetrating flexible bodies. *Computer Graphics*, 26(2):303–308, 1992.

[3] D. Baraff and A. Witkin. Large steps in cloth simulation. In *ACM SIGGRAPH 98 Conference Proceedings*, pages 43–52, 1998.

[4] M. Bro-Nielsen and S. Cotin. Real-time volumetric deformable models for surgery simulation using finite elements and condensation. *Computer Graphics Forum (Eurographics '96)*, 15(3):57–66, 1996.

[5] J. Brown, K. Montgomery, J.-C. Latombe, and M. Stephanides. A microsurgery simulation system. In *Fourth International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI 2001)*, Oct. 2001.

[6] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proceedings of ACM Interactive 3D Graphics Conference*, pages 189–196, 1995.

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

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

[9] S. Gottschalk, M. C. Lin, and D. Manocha. OBB-tree: A hierarchical structure for rapid interference detection. In *ACM SIGGRAPH 96 Conference Proceedings*, pages 171–180, 1996.

[10] D. H. House and D. E. Breen, editors. *Cloth Modeling and Animation*. A.K. Peters, Ltd., 2000.

[11] A. Joukhadar and C. Laugier. Dynamic simulation: Model, basic algorithms, and optimization. In J.-P. Laumond and M. Overmars, editors, *Algorithms For Robotic Motion and Manipulation*, pages 419–434. A.K. Peters, Ltd., 1997.

[12] E. Keeve, S. Girod, and B. Girod. Craniofacial surgery simulation. In *Proceedings of the 4th International Conference on Visualization in Biomedical Computing (VBC '96)*, pages 541–546, Sept. 1996.

[13] E. Keeve, S. Girod, P. Pfeifle, and B. Girod. Anatomy-based facial tissue modeling using the finite element method. In *Proceedings of IEEE Visualization '96*, 1996.

[14] J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volumes hierarchies of *k*-DOPs. *IEEE Transactions On Visualization and Computer Graphics*, 4(1):21–36, 1998.

[15] R. M. Koch, M. H. Gross, F. R. Carls, D. F. von Büren, G. Fankhauser, and Y. I. H. Parish. Simulating facial surgery using finite element models. In *ACM SIGGRAPH 96 Conference Proceedings*, pages 421–428, 1996.

[16] U. G. Kühnapfel, H. K. Çakmak, and H. Maaß. Endoscopic surgery training using virtual reality and deformable tissue simulation. *Computers & Graphics*, 24:671–682, 2000.

[17] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance calculation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1008–1014, 1991.

[18] B. Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, 1998.

[19] R. V. O'Toole, R. R. Playter, T. M. Krummel, W. C. Blank, N. H. Cornelius, W. R. Roberts, W. J. Bell, and M. Raibert. Measuring and developing suturing technique with a virtual reality surgical simulator. *Journal of the American College of Surgeons*, 189(1):114–127, July 1999.

[20] I. J. Palmer and R. L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2):105–116, 1995.

[21] G. Picinbono, H. Delingette, and N. Ayache. Non-linear and anisotropic elastic soft tissue models for medical simulation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, May 2001.

[22] S. D. Pieper, D. R. Laub, Jr., and J. M. Rosen. A finite-element facial model for simulating plastic surgery. *Plastic and Reconstructive Surgery*, 96(5):1100–1105, Oct. 1995.

[23] S. Quinlan. Efficient distance computation between non-convex objects. In *Proceedings of the IEEE International Conference On Robotics and Automation*, pages 3324–3329, 1994.

[24] A. Smith, Y. Kitamura, H. Takemura, and F. Kishino. A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 136–145, 1995.

[25] D. Terzopoulos and K. Waters. Physically-based facial modelling, analysis, and animation. *The Journal of Visualization and Computer Animation*, 1:73–80, 1990.

[26] G. van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.

[27] P. Volino and N. Magnenat-Thalmann. Collision and self-collision detection: Efficient and robust solutions for highly deformable surfaces. In *Eurographics Workshop on Animation and Simulation*, 1995.