Real – Time Surgery Simulator

Francisco Melendez Blas
blasfm@dcs.gla.ac.uk
0211930

24th April 2003
I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Francisco Melendez Blas
# Contents

Contents ................................................................. 1
Figures ................................................................. 2

1 Introduction. ....................................................... 3
2 Project definition. ................................................ 4
   2.1 3D-Matic Research interests. ......................... 4
   2.2 Project Requirements. .................................... 5
      2.2.1 Project Description ................................ 5
      2.2.2 Technical requirements. ......................... 5
2.3 Technical background. ...................................... 6
   2.3.1 Surgery Simulation .................................... 6
      2.3.1.1 Soft-tissue models. ............................ 6
      2.3.1.2 Collision Detection. .............................. 7
   2.3.2 Real time applications. .............................. 8
   2.3.3 3D-Graphics. ............................................ 8
      2.3.3.1 Java3D. ............................................. 8
      2.3.3.2 VRML ............................................. 9
3 Project design. ..................................................... 10
   3.1 General Description. ....................................... 10
   3.2 IO. .................................................................. 10
   3.3 Surgery Simulation ........................................ 11
      3.3.1 Soft tissue Model .................................... 12
      3.3.2 Mass spring elastic mesh. ....................... 12
      3.3.3 Simulation algorithm. .............................. 13
      3.3.3.1 Soft-tissue models. ............................ 13
      3.3.3.2 VRML ............................................. 9
      3.3.3.3 Java3D. ............................................. 8
   3.4 Interaction .................................................... 19
      3.4.1 Selecting a tool. ...................................... 19
      3.4.2 Moving the tool ..................................... 19
4 Implementation .................................................. 21
   4.1 Interaction Package ....................................... 21
   4.2 Simulation Package ....................................... 23
      4.2.1 Simulator class. ...................................... 23
      4.2.2 Collision Package. .................................. 24
      4.2.3 Deformation Package. ............................ 24
      4.2.4 Model Package. ...................................... 25
      4.2.5 Tool Package. ....................................... 26
5 Evaluation. ......................................................... 27
   5.1 Performance evaluation. ............................... 27
   5.2 User Interface test. ....................................... 27
6 Conclusion and Future Work. ............................ 28
   6.1 Project Status. ............................................. 28
      6.1.1 Positive Aspects ..................................... 28
      6.1.2 Negative Aspects ................................... 28
   6.2 Future Work. ................................................ 28
7 User Manual. ....................................................... 29
   7.1 Main window. .............................................. 29
   7.2 Tool Bar. ...................................................... 29
      7.2.1 Move the camera. ................................. 29
7.2.2 Move the tool ................................................................. 29
7.2.3 Capture. ................................................................. 29
7.3 Tools Tray. ................................................................. 30
  7.3.1 Include a tool ................................................ 30
  7.3.2 Remove a tool ................................................ 30
7.4 3D World .................................................................. 30
  7.4.1 Moving the camera ........................................ 31
  7.4.2 Selecting a tool ............................................. 31
  7.4.3 Moving the tool ............................................. 32
8 References ................................................................. 34

Figures

Figure 1: Package hierarchy of the Surgery simulator. ................................................................. 10
Figure 2: Conceptual components and data flow ................................................................. 11
Figure 3: Wave node expansion ............................................................................................ 14
Figure 4: Triangle Circumcentre ........................................................................................... 16
Figure 5: Wave triangle expansion ........................................................................................ 17
Figure 6: Simulation sequence diagram ............................................................................... 18
Figure 7: Changing tool process ........................................................................................... 19
Figure 8: .................................................................................................................................. 20
Moving in the surface.  Rotating the tool. Moving up/down ................................................. 20
Figure 9: 3D interaction use case diagram ........................................................................... 20
Figure 10: Internal class structure of the ss.interaction package ......................................... 21
Figure 11: Simulation main procedure .................................................................................. 23
Figure 12: Deformation result of the intersection with the tool ........................................... 25
Figure 13: .................................................................................................................................. 26
Tool rendered.  Tool intersection triangle ........................................................................ 26
1 Introduction.

The 3D MATIC Research laboratory has developed techniques for the generation of photorealistic 3D surface models of the human body. These models have been used for years in medical applications dealing with shape analysis of patients with body deformities. They are currently developing volumetric models - with anatomical structures such as organs, bones and muscles, which are suitable for open surgery applications. One of the challenges of this laboratory is to create a real-time surgical simulator.

The design of simulators for surgical training and planning poses a great number of technical challenges encompassing the efforts of various disciplines, Graphics, Vision, Mechanics, Robotics, and Medicine. This project is the first step of this surgical simulator and 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. We have integrated these algorithms in a 3D real-time user interface, allowing the use of several tools.

First we give a review of the 3D technologies involved in the project and different techniques used in surgery simulation and possibilities in designing soft-tissue models, then we show how we detect the collisions and deform the objects, and our implementation of spring-mass model. Finally we present some performance results and how could be oriented the next steps in this challenging project.
2 Project definition.

2.1 3D-Matic Research interests.

The 3D-MATIC research laboratory is the computer graphics and vision laboratory of the Department of Computing Science of the University of Glasgow. This department is one of the foremost in the UK. 3D-MATIC Research Laboratory with its team of 20 researchers covers a wide range of application areas: 3D capture, 3D graphics, medical imaging, animation and image representation. It has developed the new technology that is revolutionising 3D data capture. This technology now makes it possible to capture 3D models of moving people that are both metrically accurate and photo-realistic in appearance, using digital camera technology. Moreover this system should be soon completed with a set of thermo-cameras, which will provide useful data about several medical conditions such as skin diseases and breast cancer.

The 3D-MATIC research laboratory, with its unique equipment, research team and its collaborations, in particular, with the Glasgow Multimodal Interaction Group (specialised in haptics or force-feedback devices), the bioengineering department of Strathclyde university) and the Glasgow dental hospital (specialised in craniofacial surgery) is the ideal place for carrying projects whose applications are in the medical field.

The following subjects are being investigated (entertainment & medical):

- Exploration of the area of capturing and creating realistic and usable models of humans, animals and inanimate objects for manipulation and/or animation.
- Development of an infrastructure and tools for research into whole body imaging, for applications in the creative media, biomedical and other sectors.
- Investigation of the potential use of a novel computer assisted three-dimensional technique to standardise analysis of facial clefts and assess the changes following surgical repair.
- Investigation of special effects techniques in movies. It aims to explore the extraction of 3D information from movie sequences in order to allow a far more sophisticated manipulation of the images.
- Design of a common animation platform allowing different research teams to co-operate in a distributed virtual environment. In this environment, each client can use its own software to manage his or her objects and display the scene.
- Exploration of the area of digitisation and manipulation of 3D objects in real-time, bringing about a radical progression in the qualitative tasks that can be accomplished using 3D vision technology.
2.2 Project Requirements.

2.2.1 Project Description.

The 3D MATIC Research laboratory has developed techniques for the generation of photorealistic 3D surface models of the human body. These models have been used for years in medical applications dealing with shape analysis of patients with body deformities. They are currently developing volumetric models - with anatomical structures such as organs, bones and muscles, which are suitable for open surgery applications. One of the aims of this laboratory is to create a real-time surgical simulator.

The main tasks that were involved in this simulator are the followings.

- Creation of a GUI with a 3D viewer (in Java3D)
- Implementation of a real time physically based soft-tissue model (spring-mass system)
- Implementation of a fast collision detection algorithm
- Implementation of an efficient mesh subdivision algorithm
- Simulation of surgical procedures: pulling, pushing, cutting, suturing, etc.

The project enclose the first steps of this surgical simulator having been implemented the followings points.

- Creation of a GUI with a 3D viewer (in Java3D)
- Implementation of a real time physically based soft-tissue model (spring-mass system)
- Implementation of a fast collision detection algorithm
- Simulation of pulling and pushing procedures.

2.2.2 Technical requirements.

The application should be able to read and save VRML files and allow deforming them with the action of several tools at the same time. In order to obtain it, should be implemented a spring mass soft tissue model and a collision detection algorithm. User interface will be as intuitive as possible. Deformations and interaction should performance concurrently and in real time, so will be implemented a time monitor for the main tasks and we will give the possibility of changing their timing. Some surgical procedures will be implemented, at least pulling and pushing. Realistic visual results are supposed to be achieved.

It will be implemented in Java combining 2D and 3D interfaces.
2.3 Technical background.

2.3.1 Surgery Simulation.

The design of surgical simulators poses a great number of technical challenges, encompassing the efforts of various disciplines, including Graphics, Vision, Mechanics, Robotics, and Medicine.

First problem we find is design a 3D real time physically based soft-tissue model. In order with that, different strategies have been advocated. Most notably, [Gi97] suggests methods to accelerate the conventional FEM setting. Another interesting approach is the 3D ChainMail, as introduced by [Ku97]. Rather than computing physical deformations on the fly, the method uses a two-pass hybrid scheme, where in a first pass, pure geometric deformation fields are applied. In the second pass, the tissue is post relaxed by some iterative solvers. The topology of the discretization is restricted to tensor product grids. Others, like [Su97] employ surface based mass-spring systems for their real-time simulators. Similar Euler type methods on regular grids are reported in [Re97]. Pure geometric and topological manipulations based on marching cubes techniques can be found in the algorithms of [Re96] and [Re97], whose visual quality is amazing. In most approaches, force feedback devices are utilized to implement the interface to the user. Furthermore, [Te95] for instance, developed a soft tissue model for facial animation and [Ne98] used mass-spring and particle systems for the representation of human muscles.

In summary, existing several solutions to be implemented. We have implemented a 3D spring-mass model, according with the project proposal. The deformation and collision real-time algorithms are based on Quasi-static algorithm and Quinlan’s collision algorithm [Bro] respectively. Both of them will be detailed further on.

2.3.1.1 Soft-tissue models.

Many approaches to soft tissue modeling are based on surface models. Early works restricted themselves to pure geometric deformation[ Ko88]. Another approach has been to use models based on implicit surfaces [Ma95], [Mo97] and [Ne98]. These models are composed of a skeletal model upon which parameterized muscles are built up manually and the entire body form is then skinned, i.e. covered in virtual skin. However realistic deformations can only be achieved by using physically based models. Surface based models were developed using the finite element method (FEM) for facial animation [Ko98] and surgery simulation [Br96]. They have proved to be very powerful in the context of animation, but they have obvious limitations since they were aimed to show visually convincing deformations. Two types of volumetric models have been developed depending on the need of interaction with the 3D model. Real time deformations have been demonstrated using models based on the Hooke’s law and Lame equation [De99] and [Aube00] and or combining elastic surface and geometric constraints [Tu98]. The most realistic models are based on volumetric mass spring system or FEM simulation. Usually soft tissues are divided in different layers (Skin, fat, muscle.), which have distinct physical properties.
The first models were based on mass spring system and are still used because they have a lower computational cost. 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 [Te90], cloth motion [Ba98], and surgical simulation [Jo97], to cite only a few works. They are relatively fast, and allow realistic simulation for a wide range of objects, including viscoelastic tissues encountered in surgery.

The finite element method imposed itself as the most accurate way of simulating soft tissue deformation since non-linear elasticity and incompressibility can be simulated. 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 [Ke96],[Ko96],[Pi95]. One of the barriers to using finite element analysis in soft tissue deformation is the generation of the 3D volumetric mesh on which the simulation will be applied. The mesh generation is a critical feature of the pre-processing stage since the accuracy of the numerical results is strongly related to the quality of the underlying meshes. 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 [Ku00]. Others use preprocessing steps to reduce FEM computation,[Br96],[Co00] and [Pi01] extend the “tensor-mass” model of [Co00] to nonlinear elasticity. Other examples of mass-spring models, FEMs, and alternate models are too numerous to cite here.

2.3.1.2 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. br [Ba90],[Lin91],[Mi98]) and hierarchical (e.g. br[Co95],[Go96],[Kl98],[Pa95],[Qu94]). 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.

There are some solutions to this problem like the proposed in [Bro] 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. This algorithm derives from the one proposed by Quinlan [Qu94] for rigid objects. Fewer works exist for deformable objects (e.g. [Ba90],[Be97],[Jo97],[Sm95],[Vo95]).
2.3.2 Real time applications.

There are several definitions of real-time, most of them contradictory. Unfortunately the topic is controversial, and there doesn't seem to be 100% agreement over the terminology.

1. The canonical definition of a real-time system is the following:

"A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."

A good example is a robot that has to pick up something from a conveyor belt. The piece is moving, and the robot has a small window to pick up the object. If the robot is late, the piece won't be there anymore, and thus the job will have been done incorrectly, even though the robot went to the right place. If the robot is early, the piece won't be there yet, and the robot may block it.

Nevertheless there is another kind of systems that are called real-time but they do not have so severe consequences if the time planning is not conformed. In this case they are called soft real-time systems. Much of what is done in real time programming is actually soft real time system. Good system design often implies a level of safe/correct behaviour even if the computer system never completes the computation. So if the computer is only a little late, the system effects may be somewhat mitigated.

Our system is “soft real time” own to our restrictions are based in visual realism so we have implemented correct behaviours when time is not enough for complete the collision detection and deformation, but it is not allowed to control time rendering time, so we only can expect a good performance of the Java3D rendering.

[Real Time]

2.3.3 3D-Graphics.

2.3.3.1 Java3D.

The Java 3D API is an interface for writing programs to display and interact with three-dimensional graphics. Java 3D is a standard extension to the Java 2 JDK. The API provides a collection of high-level constructs for creating and manipulating 3D geometry and structures for rendering that geometry. Java 3D provides the functions for creation of imagery, visualizations, animations, and interactive 3D graphics application programs.

More specifically the Java 3D API is a hierarchy of Java classes, which serve as the interface to a sophisticated three dimensional graphics rendering and sound rendering system. The programmer works with high-level constructs for creating and manipulating 3D geometric objects. These geometric objects reside in a virtual world, which is then rendered. The API is designed with the flexibility to create precise virtual worlds of a wide variety of sizes, from astronomical to subatomic.
Despite all this functionality, the API is still straightforward to use. The details of rendering are handled automatically. By taking advantage of Java threads, the Java 3D renderer is capable of rendering in parallel. The renderer can also automatically optimize for improved rendering performance. A Java 3D program creates instances of Java 3D objects and places them into a scene graph data structure. The scene graph is an arrangement of 3D objects in a tree structure that completely specifies the content of a virtual universe, and how it is to be rendered.

Java 3D programs can be written to run as stand alone applications, as applets in browsers which have been extended to support Java 3D, or both.

[Ja00]

### 2.3.3.2 VRML

The Virtual Reality Modeling Language (VRML) is a file format for describing interactive 3D objects and worlds. VRML is designed to be used on the Internet, intranets, and local client systems. VRML is also intended to be a universal interchange format for integrated 3D graphics and multimedia. VRML may be used in a variety of application areas such as engineering and scientific visualization, multimedia presentations, entertainment and educational titles, web pages, shared virtual worlds, and why not, surgery simulation.

VRML is capable of representing static and animated dynamic 3D and multimedia objects with hyperlinks to other media such as text, sounds, movies, and images.

VRML browsers, as well as authoring tools for the creation of VRML files, are widely available for many different platforms.

VRML supports an extensibility model that allows new dynamic 3D objects to be defined allowing application communities to develop interoperable extensions to the base standard. There are mappings between VRML objects and commonly used 3D application programmer interface (API) features.

[Vrml95],[Vrml97]
3 Project design.

3.1 General Description.

The tool result of this project is called Surgery Simulator. The main Java class is SS. We can distinct three main parts in the structure. Firstly we find the user interaction with the 3D world, tool behaviors. The second one encloses IO system, including .WRL and .OBJ loaders, VRML 1.0 writer and image capture. The final one and the most proper is Surgery system. It includes spring mass model, tools, collision and deformation.

Figure 1: Package hierarchy of the Surgery simulator.

3.2 IO.

The first problem we faced was read VRML files. In order to that, we started with a previous application (FilmEditor3D)[1] that implements a VRML 1.0 reader and conversion to Java3D structures. Basically we have used the “loaders” package with small modifications. Instead of loading the model as “TriangleArray” geometry we load it as “IndexedTriangleArray”. That simplifies the update of the mesh when it is deformed; we only have to worry about the nodes and not bout the triangles it belongs to.

Have been done some modifications in the loading actions as well as in the capturing. Nevertheless these changes are not significant so it is already documented in (FilmEditor3D documentation).
3.3 Surgery Simulation.

A surgery simulator has some procedures to control: collision detection, cutting, deformation, relaxation, render, force feedback... Some papers present all of them as individual components running concurrently and some of them sequentially. We have chosen a sequential model thinking that sequential execution has more logical sense because deformation is consequence of a collision and there is not relaxation if first there has not been cutting. Furthermore this way is easier to control.

In future steps of the simulator, the next sequence of individual processes could be followed.

Basically our system executes concurrently the movement of the tool and the simulation procedure. Simulation can be paused to allow others behavior or actions in order to not over charge the system. For more details see the implementation.

Figure 2: Conceptual components and data flow.
3.3.1 Soft tissue Model.

Mass-spring meshes seem better suited for surgical training 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. That is the reason we chose this soft tissue model in this project.

3.3.2 Mass spring elastic mesh.

The geometry of a deformable object usually is represented 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 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 $F_{ij} = -k_{ij} \Delta_{ij} u_{ij}$, where $\Delta_{ij} = l_{ij} - r_{ij}$ is the current length of the link minus its resting length, and $u_{ij}$ is the unit vector pointing from $N_i$ toward $N_j$. The stiffness $k_{ij}$ may be constant or function of $\Delta_{ij}$. In either case, $F_{ij}$ is a function of the coordinate vectors $x_i$ and $x_j$ 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 $N_i$:

$$m_i \ddot{x}_i + c_i \dot{x}_i + \sum_{e \in \sigma(i)} F_{ij}(x_i, x_j) = m_g + F_{ext,i} \quad \text{Eq. (1)}$$

where $x_i$ is the coordinate vector of $N_i$, $v_i$ and $a_i$ are its velocity and acceleration vectors, respectively, $m_g$ is the gravitational force, and $F_{ext,i}$ 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 $M$.

As is explained below, at the moment we do not need many of this coefficients and physic properties so our elastic-mesh only links with spring coefficient and nodes with coordinates and different status. The model also stores in memory all the triangles. Future models can extend these classes adding necessary properties.
3.3.3 Simulation algorithm.

We have implemented a “quasi-static” simulator. Dynamic simulators use 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 are based in the quasi-static simulator described in (Brown99) but some modifications have been done. We describe the bases of the system below and our changes.

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 (some of them not yet integrated). 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.

Quasi-static algorithm.

Under the above assumptions, we neglect dynamic inertial 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:

\[ \sum \sigma_{ij} F_j(x_i, x_j) - mg = 0 \quad \text{Eq.(2)} \]

Let I be the set of indices of all the non-control nodes of M, and let \( \delta \) be a constant time step. At each time \( t = k \delta, k = 1, 2, \ldots \), 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 \( \delta \). The algorithm is the following:

Algorithm QSS:

1. Acquire the positions of all the control nodes
2. Repeat until time \( \delta \) has elapsed
   For every \( i \in I \)
   (a) \( f_i = \sum \sigma_{ij} F_j \)
   (b) \( x_i = x_i + \alpha f_i \cdot mg \)

Step 2 computes the residual force applied to each node and displaces the node along this force. Moving the node along a combination of the old and the new forces can also use a conjugate gradient style method. 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 non-zero force \( f_i \) acting on them after \( \delta \) 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 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.

![Figure 3: Wave node expansion.](image)

**Modifications.**

We also assume that gravity acceleration has not significant effect in the deformation. It is also difficult to know what the mass of the nodes is without having definitive models. Anyway probably in the future it should be included so we will have to define the gravity direction because we allow rotation of the scene and not all the models are loaded in the same orientation.
After some tests our algorithm did not converge correctly so we included a scale factor to reduce the action of same direction forces. This factor evaluates the importance of each force in relation with the most important force. So if we have two similar forces we will divide the total force by almost 2 because in other way we would get a very big deformation what would suppose divergence in the algorithm.

\[ f_i = \sum_{j \in \sigma(i)} F_{ij} \alpha \]

\[ \alpha = 1 / \sum F_{ij} \text{length} / \text{MaxF}_{ij} \text{length} \]

We also have modified the wave expansion system. The proposed algorithm stores all the nodes before start to calculate them. We have included the cutout in the node expansion. We initialize I with the control nodes deformed, and it is dynamically wave expanded. When all the nodes of a level are not modified more than a small pre-specified amount we stop getting more nodes.

3.3.3 Collision detection.

As is commented above our collision detection system is based in Quinlan’s algorithm for rigid objects and modifications described in (Brown99). Nevertheless we designed our proper system that is the one integrated in simulator. We resume here the Quinlan’s concept and its application in deformable object as well as our simplification.

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 " and constructs an approximately balanced binary tree T that has one leaf per sphere of radius ". 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\textsubscript{i} and E\textsubscript{j} 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\textsubscript{i} and E\textsubscript{j}. 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\textsubscript{i} and T\textsubscript{j} be the respective sphere trees of two (rigid) objects A\textsubscript{i} and A\textsubscript{j}. A collision query is specified by the position and orientation of A\textsubscript{i} relative to A\textsubscript{j}. Collision detection is performed by a depth-first traversal of T\textsubscript{i} and T\textsubscript{j} 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\textsubscript{i} and N\textsubscript{j}, the root spheres of T\textsubscript{i} and T\textsubscript{j}, respectively, the following algorithm returns 1 if it detects a collision, and 0 otherwise:
Algorithm COLLISION(N₁, N₂):

1. If N₁ and N₂ have null intersection then return 0
2. Else
   (a) If both N₁ and N₂ are leaf spheres then test the corresponding two triangles for collision;
       return 1 if they collide and 0 otherwise
   (b) If N₂ is smaller than N₁ then switch N₁ and N₂
   (c) If COLLISION(N₁, left-child(N₂)) = 1 then return 1
       Else if COLLISION(N₁, right-child(N₂)) = 1 then
       return 1
       Else return 0

Application to deformable objects.

To use COLLISION, we must maintain the sphere tree of every deforming object. (Brown99) proposes 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 equalsized spheres, we assign each triangle a single leaf sphere of T – the smallest sphere enclosing the triangle. (When the circumcentre is outside of the triangle we take the central point of the longest segment) 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:

\begin{algorithm}
  \textbf{MAINTENANCE}:
  \begin{enumerate}
    \item While $Q$ is not empty do
    \item w extract($Q$)
    \item Adjust the radius and position of w
    \item Insert ($Q$, parent(w))
  \end{enumerate}
\end{algorithm}

\textbf{Modifications.}

We implemented this collision detection system including timeout in the COLLISION algorithm. Afterwards we started to work with the user interaction with the tool. We realized that it was very difficult to control a tool in a 3D world with a 2D device, as the mouse is, giving the usual $x$, $y$, $z$ movement. We designed a new way to move the tool where the user chose the point to interact before starting the possible deformation. It gave us the possibility of knowing where the tool was and what triangles was close to. With this new information we remove the sphere tree of the system but keeping the leaf spheres, because we still can disregard some triangles, designing a wave expansion collision detection algorithm. It means that once we know the triangle chosen we get all the adjacent leaf spheres that collision with the tool bound and then we compress the wave quitting the triangles that do not have collision. In that way if the time assigned to detect the collision is over, we will get only the triangles with bigger possibility of intersection.

![Figure 5: Wave triangle expansion.](image)

This algorithm is faster that the other but is dependent of the tool behavior, so probably we will have to use the previous one with some types of tool, when they are included.
3.3.4 Real Time System.

It has been repeatedly commented that the surgery simulator has to perform in real time. As is described above, deformation and collision algorithms have time limit to finish their tasks, and implements a save/correct status when they have not had time enough to finish them. So we can control them separately assigning the time they use to detect the collision and the time for deformation. These times are assigned by a timer controller and can also be fixed by the user.

To improve the performance of the system it is required to monitor the time usage of the different components or, at least, the final status of the task when the time assigned elapses, to give the timer controller information in order to reassign the times. For instance, at the beginning all the time should be assigned for the collision detection, until we have collision, then the system should reduce the time for collision to the time it needed to detect it and the rest for deformation. This gives some flexibility that is required because, obviously, not all the systems have the same computing capabilities and not all the actions have the same importance in the different status of the simulation.

Basically our system has a time controller that monitors the time usage and takes decisions in the time assignment. These decisions are quite simple at the moment, but more complex ones are easy to integrate. User can also fixed the timing in order to get more exact results even forgetting the visual realism.

![Simulation sequence diagram](image-url)
3.4 Interaction.

We differentiate two aims to solve in our project related with user interaction. The first one is provide the functionality for interactively controlling the position of the camera. This problem is well solved in the FilmEditor3D (software we started with) so basically we have included this capability in our system. It is perfect explained in (Ewan02). On the other hand we have to design a 3D user interface to select and move the different tools in the scene. Best way to include this kind of behaviour in a Java3D world is to extend the Behaviour class.

We design two behaviours, one to select the active tool and one to move it. Both of them should be as intuitive as possible, own to they are user interface components.

3.4.1 Selecting a tool.

Most of 2D interfaces to select active windows with a mouse have two steps. Firstly user has a previous selection moving the mouse over the object, with a change of its visual state, and secondly definitive selection pressing the mouse button, with another change of state or with an action. This has the model we have followed.

![Figure 7: Changing tool process.](image)

3.4.2 Moving the tool.

Move the tool intuitively in a 3D world with a 2D device is not trivial. We tried with Java3D interaction tools, but they did not give a good solution for the movement freedom and control we need in surgery. Discarding them we decided to implement new tool behaviour to allow pushing and pulling actions. The most intuitive and realistic option we found was the next procedure.

1. Select the point of the object to pull or push where we want to actuate.
2. Allow rotation to select the angle of the deformation.
3. Move the tool in Y-axis up or down.
In each moment we give a different appearance to the tool to know what action we are doing.

Moving in the surface. Rotating the tool. Moving up/down.

This use case diagram defines interaction system.
4 Implementation.

The SS class is the main class for the Surgery Simulator 1.0 Beta application and functions primarily as a user interface for the functionality that is provided by the different packages. It structures the layout of tools and options around the 3D canvas. It is a JApplet, but due to Java2 security restrictions it cannot actually run as an applet. Instead it uses a Java3D utility class called MainFrame12, which enables it to run as an application.

It implements the PropertyChangeListener interface, however only some of the actions in the program make use of the possibilities of this utility. Anyway with a sort time it can be properly used improving the user interface. In fact all the interface implementation is quite unfinished. We wanted to add some functionalities but time wasn’t enough. Anyway program is quite robust in the functionalities it has.

Some of the components included in this program are simple adaptation of the initial tool that was provided (FilmEditor3D). So we are not going to describe these packages. We focus this chapter in the two main aims have been implemented: 3D interface and surgery simulation.

4.1 Interaction Package.

Adding to the user 3D interaction included in FilmEditor3D, that consist in modifying the camera position of the 3D scene, we implemented the two functionalities described in the design. Both of them extend Behavior class. This class is provided by Java3D API, and is the easiest way to implement interaction with the 3D world. A Behavior has to implement the procedure processStimulus() that is waken up by some event conditions.

Figure 10: Internal class structure of the ss.interaction package.
SelectBehavior class.

This class uses move and pressed mouse events. Move event detects if there is a tool under the mouse cursor using PickCanvas class, which is capable to detect if there is a collision between a ray traced from two coordinates in the screen and an object in the scene. If there is collision and the object is a tool, we move a select green ball in the position of the tool. If there is not the ball is quitted of the scene. Pressed event is used to concret the selection of the tool. If it is concreted a red ball in the position of the tool is move in its position.

ToolBehavior class.

This class uses the Java3D PickRotateBehavior class, pressed and dragged mouse events and pressed key event. Also uses PickCanvas class for moving the tool over the model surface.

Getting the closed intersection of this model and the mouse is done by the following code.

```java
int x = ((MouseEvent)event[i]).getX();
int y = ((MouseEvent)event[i]).getY();
pickCanvas.setShapeLocation(x, y);
Point3d eyePos = pickCanvas.getStartPosition();
pickResult = pickCanvas.pickAllSorted();
if (pickResult != null) {
    // Get closest intersection results
    PickIntersection pi = pickResult[0].getClosestIntersection(eyePos);
    GeometryArray curGeomArray = pi.getGeometryArray();
    int []coordidx = pi.getPrimitiveCoordinateIndices();
}
```

Pressing the right button of the mouse we can change the status of the tool. At first we give a different status depending of the number of times it was pressed.

1. Simple click: Move the tool.
2. Double click: Rotating.

As is exposed in the evaluation, triple click was quite difficult for some no-experimented users. So we decided to change it, giving the next status with only one click.

Rotation also is quite difficult to control, but this change suppose bigger changes in the program and it has not been implemented yet.
4.2 Simulation Package.

4.2.1 Simulator class.

The main class taking in account the functionality inside the program is contained is this package. **Simulator** class integrates the different components for the surgery simulation. This class implements runnable to allow the concurrent execution with the rest components of the program as the user interface. This executes an infinite loop with a pause condition that can be changed by external classes. This pause is turned true with IO actions and camera position changes. This is the basic execution diagram.

Public function setPause(Boolean) takes charge of changing Pause value. To make it possible setPause() and the procedure called by run() must be synchronized.
4.2.2 Collision Package.

This package implements the collision detection functionality. For this aim it implements three classes. **SphereLeaf**, **SphereSurface**, **CollisionDetectorSurface**. Actually there are three more classes included but they are only used in the tool. **SphereTree**, **Sphere**, **CollisionDetector**. These three classes implement the previous version mentioned in the design and have been included because can be useful in the future.

Intersection algorithms are implemented in the respective objects that are involved in the checking. Is proposed to implement an abstract class to enclose more kinds of bounds as cubes or cones. This class should implement a procedure called `checkIntersection()`.

To check the intersection between two objects, it is necessary to set corresponding sphere surfaces, and call the function `checkIntersection(a,b)` where `a` and `b`, are the Transform3D(Matrix with the position, orientation and scale factors in the scene) of each object.

**CollisionDetectorSurface** implements runnable making a thread sleep for the time assigned for this task.

```java
public void run()
{
    try
    {
        Thread.sleep((long)(this.time));
    }
    catch(InterruptedException e)
    {
        System.out.println(e.getMessage());
    }
    flagStopIntersectionCalculate = true;
}
```

The time controlling of this algorithm is done creating a new thread executing this procedure.

```java
Thread timeControler = new Thread(this);
timeControler.setPriority(java.lang.Thread.MAX_PRIORITY);
timeControler.start();
```

4.2.3 Deformation Package.

Deformation package has three classes. **Deformation**, **DeformationCollection** and **Deformer**. **Deformation** contains basically a list of the nodes deformed and the action point of the tool involved in the deformation. **DeformationCollection** is basically a list of deformations. And Deformer is the class that implements the algorithm described in the design. At the moment the deformer is who make effective the first deformation of the control nodes. We only have one type of deformation in the simulator, but in the future we will have several tool, with different actions. The future structure we propose is that the deformation gives to the **Deformer** the nodes and the forces applied over them. Creating as well an abstract class called **Deformation**. Each type of deformation will be tool dependant and would
deform the mesh and give the forces. In that way the process of relaxation could be done using the deformer algorithm.

![Figure 12: Deformation result of the intersection with the tool.](image)

Similar timing control, that the presented above, is implemented in Deformer.

### 4.2.4 Model Package.

Model package includes the 3D model; render capabilities and functionalities, basic spring-mass data structures and respective actions. The classes are Link, LinkKey, LinksTable, Node, Triangle, InteractiveModel and ScaleFactorDialog. Link, Node and Triangle classes are used in the collision detection and deformation. Link and Node contain the physical parameters of the model. LinksTable allow a faster access to the links of a node. LinkKey is used to obtain this faster access. InteractiveModel integrates the rest of the classes, besides of render and visual attributes. It is also in charge of getting appearances, Nodes and triangles of the model loaded.

We store two copies of the model, one to be modified and one with the original geometry. It allows us a faster render when we want to come back to the first equilibrium status, that is, when there is not tool action over the model. We use IndexedTriangleArray Java3D class. We have detected that when we use the compact mode some of the links are lost. It has a bad visual effect because some holes appear in the model. However we save a lot of memory, due to using the non-compact mode we have to store each Node multiplied per the number of triangles it belongs to. That is necessary to keep the index consistence with the geometry. So we encourage using the compact mode. Therefore it is probably a Java3D bug that will be solved.

InteractiveModel class extends `javax.media.j3d.BranchGroup`, because the BranchGroup class is the only class with capability to be added or removed of a live group.
4.2.5 Tool Package.

This package is composed by two classes and some and a sub-package “utils”. The classes of this sub-package are used to visualize the tool status. The proper classes of the tool package are Tool and ToolGroup. ToolGroup is basically a list of tools and give the functionality of adding and removing them of the scene. Tool class extends TransformGroup and stores the rendered model, triangles used in collision detection, action points and a spheres tree.

The most remarkable thing of the Tool class is that the model and the list of triangles used in the collision are independent. It means that we can render a very complex and realistic shape, and use a very simple model in the collision. Obviously them must keep some size and shape relation. The collision algorithm is designed to work with a simple model although it is capable to work with complex spheres trees.

Figure 13:

Tool rendered. Tool intersection triangle.
5 Evaluation.

The application carries out the functionalities required by the project. Each component has been optimized and the result obtained from the main algorithms is satisfactory. Functionality is correct but some detected bugs must be fixed. The simulator itself is quite provisional because the usage of time spent in optimizations of the algorithms has incurred in a non-definitive implementation. Therefore some improvements in the implementation and some redesigning in order to include new surgery procedures should be done. It is the intentions of the author to make effective these improvements in the next weeks.

Because of this provisional status, no formal evaluation experiments have been realized for lack of time. However supervisor and voluntaries have checked the program in order to get GUI conclusions. Anyway the product is still in a research phase and the actual application had good algorithm performance preference.

5.1 Performance evaluation.

General performance of the application has been tested running on a 960MHz Pentium III processor in a Compaq station with 256MB of RAM. Real time results are obtained with models that contain an amount of approximately 7,000 triangles. But the render is still a bit slow getting a total frequency of 9.4Hz. Machines with better graphical hardware solve rendering time problems getting a frequency between 20-25Hz with realistic visual results. This frequency can be increased changing the times assigned to the different components.

Deformations are quite realistic with appropriate time assigned, and collision algorithm works perfectly in the system being very fast.

5.2 User Interface test.

The interface has been informally tested with several kinds of users. The result of this testing is that once the user has understood the process, it quite easy to realize the tasks commended. However we has detected some difficulties with the rotation when the camera is also rotated. We also have tested the procedure to change the status. In general users find more intuitive change the status with only one click. However some times user cannot see the graphic symbol of the status, so associate simple, double and triple click with the different status has been the final choice. See 9.4.3 for more details.
6 Conclusion and Future Work.

6.1 Project Status.

6.1.1 Positive Aspects

The prototype application has obtained good results in the main aspects of the project that were: implement a fast collision algorithm, a soft-tissue model and integrate them in a real time system. We also have found some handicaps of Java3D in real time. Tool motion solution founded is satisfactory.

6.1.2 Negative Aspects

Actual implementation is not definitive and has little bugs that have not been fixed yet. Same case occurs with the design in order to get a structure easy to extend. Time has been a problem in this project and the constant change in the approach of some aims has affected in the implementation and design as is commented above. Anyway the final Surgery Simulator will not be implemented in Java3D so implementation is not so critical

6.2 Future Work.

Solve the implementation and design problems commented above are the main objective of the author for the next weeks. A part of this, future work is very extensive. Next step should be to include the mesh subdivision algorithm in the simulator. We recommend the algorithm presented in [Neb00] because it is thought to be co-actuate with a deformation system based in equilibrium statuses, but some ideas can be taken from [Bi99] and [Bi02]. Suturing and relaxation procedures should be adaptations of the algorithms already included. Use a 3D input device to manipulate the tools is hardly recommended by the author, even before including the cutting. It would save a lot of time designing non-definitive interfaces. Also dynamic deformation algorithms will be probably needs to be integrated in the simulator for open surgery. Force feedback system could be a last step.

Integrating all this functionalities we would obtain a good surgery simulator, however there are not really good solutions for all the problems involved in virtual surgery. Actual systems use to solve some of them separately. We hope this project will get a good simulator, and at least contribute to this challenge.
7 User Manual.

7.1 Main window.

Surgery Simulator presents a main window that can be divided in three parts: tool bar, 3D world, and the tray for the tools. The third one can be hidden.

7.2 Tool Bar.

7.2.1 Move the camera.

To allow moving the camera you must unfix the model using the open lock icon. It will allow you to change the selected tool.

7.2.2 Move the tool.

Once you have selected a tool (see 7.4.2) you can move it. To move the tool you must fix the model first. Use the close lock to do that.

7.2.3 Capture.

Any time in the simulation you can make a picture of the 3D world. Press the camera in the tool bar to this. You can choose the quality and the file name in Options.Capture settings.
7.3 Tools Tray.

The tools tray can be hidden and showed using the triangles (up-left corner). It also can be resized dragging the separator.

7.3.1 Include a tool.

To include a tool in the scene press the tool you want to include in the tray. It will appear in the scene. Maybe it is not visible depending of the camera position. Find it moving the camera or using View.Camera.GoHome.

7.3.2 Remove a tool.

To remove a tool you must select a tool (see 7.4.2) and press Esc. The tool will disappear.

7.4 3D World.

The 3D world is the scene. It will be in black until you include a tool or load a model.
7.4.1 Moving the camera.

You can configure the camera movement using camera controls. View.Camera.Show camera controls.

Move.
Press the right button of the mouse and move it keeping the button pressed.

Rotate.
Press the left button of the mouse and move it keeping the button pressed.

Zoom.
Press the centre button of the mouse and move it keeping the button pressed.

7.4.2 Selecting a tool.

To select a tool you must unfix the camera (see 7.2.1). Once you unfix the camera move the mouse cursor over the tool to choose, a semi transparent green ball will appear over the tool, press the right button of the mouse. Then a semi transparent red ball will appear over the tool. This means you have selected this tool.
7.4.3 Moving the tool.

When you have a tool selected you can move it, but first you have to fix the camera (see 7.2.2).

Move.

With the camera fixed and a tool selected, press left button of the mouse once to change the status of the tool. The “move icon” will appear. Once you have the tool in this status pick over the model in the point you want to move the tool to. The tool will move to this point in the surface of the model with the same orientation it had before.

Rotate.

Press the left button of the mouse twice (double click) to choose the rotate status. Once you have the tool in this status pick the tool with the right button and move the mouse keeping the button pressed.
Moving up.

To move the tool up, press left button of the mouse three times (triple click) to choose the moving up/down status and, once you have the tool in this status, press Shift key and the right button of the mouse. Move the mouse keeping both pressed.

Moving down.

To move the tool down use the same procedure that the described above to choose the moving up/down status. Once you have the tool in this status press the right button of the mouse. Move the mouse keeping it pressed.

Clip the model.

Once you have moved the tool close to the model you can clip it and pull or push it. Press the Enter key to fix the skin to the tool and then use “Moving up” and “Moving down” procedures. To unfix the skin press Enter Key again or left button.
8 References.

[Mo97] L. Moccuet, N.Magnenat-Thalmann, "Multilevel Deformation Model Applied to

[Neb00] Jean-Christophe Nebel Soft tissue modelling from 3D scanned data.
Department of Computing Science, University of Glasgow, 2000.


