Real-time Collision Detection for Virtual Surgery

Jean-Christophe Lombardo,
EPIDAURE/SINUS, INRIA, 2004 route de Lucioles,
06192 Sophia Antipolis Cedex, France
Jean-Christophe.Lombardo@sophia.inria.fr

Marie-Paule Cani and Fabrice Neyret
iMAGIS[†]-GRAVIR / IMAG
BP 53, 38041 Grenoble cedex 09, France
Marie-Paule.Cani@imag.fr, Fabrice.Neyret@imag.fr

Abstract

We present a simple method for performing real-time collision detection in a virtual surgery environment. The method relies on the graphics hardware for testing the interpenetration between a virtual deformable organ and a rigid tool controlled by the user. The method enables to take into account the motion of the tool between two consecutive time steps. For our specific application, the new method runs about a hundred times faster than the well known oriented-bounding-boxes tree method [5].

Keywords: Collision detection, Virtual Reality, Physically-based simulation, Graphics hardware.

1. Introduction

Collision detection is considered as a major computational bottleneck of physically-based animation systems. The problem is still more difficult to solve when the simulated objects are non-convex and when they deform over time. This paper focuses on the specific case of collision detection for a surgery simulator aimed at training surgeons at minimally invasive techniques (ie. laparoscopy).

1.1. Virtual surgery

Non-invasive surgery is rapidly expending, since it greatly reduces operating time and morbidity. In particular, hepatic laparoscopy consists in introducing several tools and an optic fiber supporting a micro-camera through small openings cut into the patient's abdomen. The surgeon, who has to cut and to remove the pathologic regions of the liver, only visualizes the operation onto a screen. Learning to coordinate the motion of the tools in these conditions is a very difficult task. Figure 1 shows a typical tool used for laparoscopic surgery and a view of the control screen during an operation.

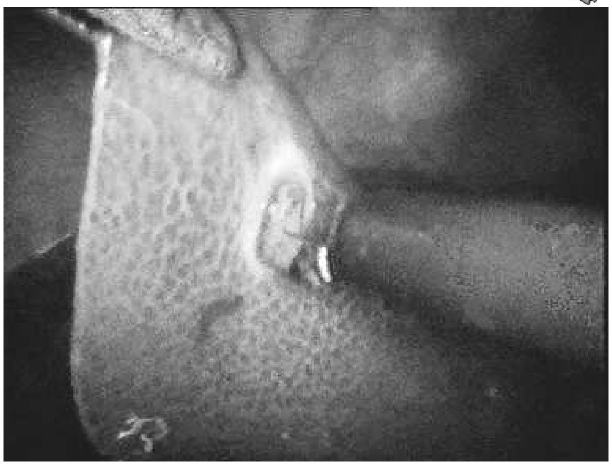

Figure 1. A minimally invasive surgery tool (top). View from the control screen (bottom).

The aim of surgery simulators is to offer a platform enabling the surgeons to practice on virtual patients, thus getting rid of financial and ethical problems risen by training on living animals or on cadavers.

Virtual surgery brings a number of difficulties: It requires both the abilities to interact in real-time with the virtual organs through a force-feedback device and to perform

[†]iMAGIS is a joint project of CNRS, INRIA, Institut National Polytechnique de Grenoble and Université Joseph Fourier.

a real-time visualization of the deformations. Moreover, the computed images should include as much visual realism as possible (texture of the organs, specular effects due to the optic fiber light, etc). In this context, the time that remains for performing collision detection at each simulation step is extremely small. The remainder of this paper focuses on this specific aspect of the problem. This work is a part of a wider project[1] that studies all the aspects of the problem, including real-time deformable models devoted to the physically-based simulation of the organs [3].

1.2. Collision detection techniques

Due to its wide range of applications, collision detection between geometric models have been studied for years in various fields such as CAD/CAM, manufacturing, robotics, simulation, and computer animation. The solutions vary according to the geometric representation of the colliding objects and to the type of query the algorithm should support. For instance, softwares that maintain the minimal Euclidean distance between the models are often required in motion planning application.

In our background of a surgery simulator, we are interested into methods that detect interpenetrations between polygonal models, since the latter are the most convenient for real-time rendering. We do not need to know the Euclidean distance between non-colliding objects. However, when a collision occurs, the precise knowledge of the intersection region is needed, since it will allow a precise computation of subsequent deformations and of response forces.

Most of the previous work in collision detection between polygonal models has focused on algorithms for convex polyhedra [1, 8]. These algorithms, based on specific data-structures for finding the closest features of a pair of polyhedra, exploit temporal and geometrical coherence during an animation. They are very efficient: the algorithm in [8] runs in roughly constant time even when the closest features change. However, they are not applicable in the case of a surgery simulator, since organs are generally non-convex, and deform over time.

Among the collision detection methods that are applicable to more general polygonal models [10, 2, 12, 4, 11, 13, 5, 7], almost all of the optimizations rely on a pre-computed hierarchy of bounding volumes. The solutions range from axis-aligned box trees, sphere trees [12, 11, 7], or BSP trees, to more specific data structures [2]. All these techniques, which perform very efficient rejection tests, may considerably slow down when objects are very close, ie. when the bounding volumes have multiple intersections. Among the recent approaches for finding bounding volumes that tighter fit the object's geometry, Gottschalk [5] obtains very good results by using hierarchies of oriented bounding boxes instead of axis-aligned boxes. Section 5 will compare our new method with the public domain software package *RAPID* that implements this technique.

A last issue in collision detection is the ability to perform *dynamic* rather than *static* detection [10, 4]: moving objects may interpenetrate between consecutive time steps, so the intersections should be computed between the 4D volumes that represent the solids' trajectories during a time step rather than between static instances of the solids. In the context of a large environment with lots of moving objects, using space-time bounds on the object's motion may lead to the quick rejection of a number of intersection tests [9, 6, 7].

In previous works on laparoscopic surgery [3], a dynamic collision detection was performed by testing for an intersection between the segment traversed by the tool extremity during a time step and the polygonal mesh representing the organ. A bucket data-structure discretizing the organ's bounding box, and storing local lists of polygons, was used to optimize this test. Real-time performances were obtained with a scene consisting in an organ and two tools, when no update of the bucket data-structure was needed. However, each tool was modeled as a single point, which resulted into possible penetrations of the body of the tools into the organ when an unexperimented user was trying to position them. Moreover, considering no update of the bucket structure was very restrictive concerning the possible deformations of the organ.

1.3. Overview

In the context of surgery simulation, the collision detection problem is enhanced by the non-convexity of most of the organs, and by the fact they deform over time. These deformations are far from negligible : laparoscopy typically involves large scale deformations and even topological changes in the structure of the liver since some parts are cut down and removed. In this context, spending time for pre-computing complex bounding volumes does not seem adequate, since this computation will need to be redone at each time step.

A second point is that, even if the number of colliding objects remains small (usually: an organ of interest and few surgical tools), objects usually stay in very close configurations. Collisions or contacts may occur at almost each time step, since the surgeon uses the tools to manipulate the virtual organ. Basically, whatever the method, an intersection test between each tool and the organ will be needed at each time step.

Thirdly, collisions need to be detected even during a fast motion of the tools, otherwise incorrect response forces would fed back to the user. So using dynamic detection, at least for the tools motion seems indispensable.

[1] http://www.inria.fr/epidaure/AISIM

Fortunately, the sum of features of the problem ease its resolution: only one of the objects (the organ) has a complex shape since the tools used in non-invasive surgery can be represented by thin and long cylinders (see Figure 1(a)). Moreover, the tools have a constrained motion since they enter into the patient's abdomen through small circular openings. These two properties enable us to take benefits of the graphics hardware for detecting collisions in real time.

The remainder of this paper develops as follows: Section 2 explains how the graphics hardware may bring a solution to our problem. Section 3 gives a method for performing static collision detection between a tool and the polygonal model of an organ. This method is extended in Section 4 in order to take the dynamic motion of the tool into account. Section 5 presents our results, including a numerical comparison of computational times with the public domain software *RAPID*.

2. Collision detection with the graphics hardware

Our aim is to find a real-time collision detection method that allows us to take the whole tool into account instead of just considering its extremity. Detecting a collision between two objects basically consists in testing if the volume of the first one (ie. the tool, which has quite a simple shape), intersects the second one. This process is very close to a scene visualization process: in the latter, the user specifies a viewing volume (or *frustum*), characterized by the location, orientation and projection of a camera; then, the first part of the visualization process consists in clipping all the scene polygons according to this frustum, in order to render only the intersection between the scene objects and the viewing volume. Specialized graphics hardware usually performs this very efficiently.

Thus, the basic idea of our method is to specify a viewing volume corresponding to the tool shape (or alternatively to the volume covered by the tool between two consecutive time steps). We use the hardware to "render" the main object (the organ) relatively to this "camera". If nothing is visible, then there is no collision. Otherwise we can get the part of the object that the tool intersects.

Several problems occur: firstly, the tool shape is not as simple as usual viewing volumes. Secondly, we don't want to get an image, but we need meaningful information instead. More precisely, we would like to know which object faces are involves in a collision, and at which coordinates. The *OpenGL* graphic library provides features that will allow us to model our problem in these terms. We review them in the next sections.

2.1. Viewing volumes

The most common frustum provided by *OpenGL* are those defined by an orthographic camera and by a perspective camera. In both cases, viewing volumes are hexahedra, respectively a box and a truncated pyramid, specified by six scalar values (see Figure 2).

Moreover, the user may add extra clipping planes for further restricting of the viewing volume, using `glClipPlane()`. All the versions of *OpenGL* can treat at least six extra planes, so the viewing volume can be set to a dodecahedron. However, we must keep in mind that efficiency decreases each time an extra clipping plane is added.

2.2. Picking

The regular visualization process is divided into a *geometrical* part and a *rasterization* part. The geometrical part converts all the coordinates of the scene polygons into the camera coordinate system, clips all the faces relatively to the viewing volume, and achieves the orthographic or the perspective projection in order to get screen coordinates. The rasterization part transforms the remaining 2D triangles into pixels, taking care of the depth by using a Z-buffer in addition to the color buffer.

Computing the first part of the process is sufficient for the applications that only require meaningful informations about visible parts of the scene. A typical example is the picking feature in 3D interaction: a 3D modeler needs to know which object or face is just below the mouse, in order to operate on it when the user clicks. If several objects project on the same pixel, it can be useful to know each of them. In 3D paint systems, the program rather needs to know the texture coordinate corresponding to the pixel which is below the mouse.

OpenGL provides two *picking* modes, that may be selected alternatively to the usual *rendering* mode `GL_RENDER` thanks to the function `glRenderMode()`. For these two modes, no rasterization is performed. Moreover, costly operations such as lighting are usually turned off. The picking modes differ from the informations they give back:

- the *select* mode `GL_SELECT` provides information about the visible groups of faces. A group name is given using `glPushName()` before each group of faces drawing, and *OpenGL* fills an array (provided by `glSelectBuffer()`) during the geometric pass of rendering, writing an entry per group that appears in the viewing volume. Thus one can know the faces that appear on screen. If the window has been reduced to a single pixel around the mouse, one gets the faces that appear below the mouse. If the camera geometry has been set in order to specify a given viewing volume,

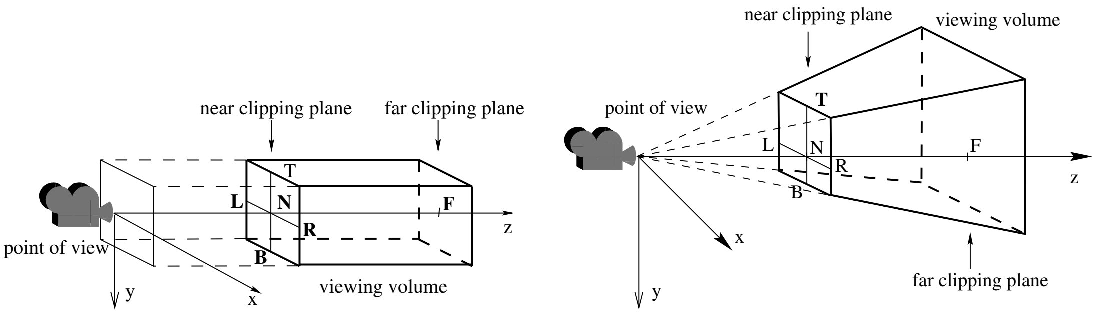

Figure 2. (a) The *OpenGL* orthographic camera (left) and the *OpenGL* perspective camera (right). The viewing volumes, which are either a box or a truncated pyramid, are characterized by the distances to the far and near clipping planes and by the two intervals [left,right] and [top,bottom] which define their section in the near clipping plane.

one gets the faces that intersect this viewing volume. Each entry contains some extra information, e.g. the z min and max inside the group, which can be used to sort or choose between multiple answers.

- the *feedback* mode `GL_FEEDBACK` provides extended information about the transformed and clipped scene. Basically, all the produced data can retrieved. The programmer indicates which kind of information he is interested in (positions, normals, colors, texture coordinates, ...), and provides an array with `glFeedbackBuffer()` that is filled by *OpenGL* during the geometrical rendering pass. In the same way that above, the scene may be clipped to a 1 pixel size window around the mouse, in order to get the geometric data corresponding to the mouse location. A naming mechanism similar to the previous one, using `glPassThrough()`, allows to get in addition the information of the faces (or groups of faces) numbers appearing in the viewing volume.

Since hardware is used to compute transformations and clipping, and since no rasterization is performed (which means that almost all interpolations are suppressed), both picking processes are particularly efficient.

3. Static Collision Detection

Laparoscopic surgery tools can be seen as cylinders of constant section s and of varying length, since user may pull or push them more or less widely into the patient's abdomen. In the remainder of the paper, we call $\mathbf{P_0}$ the fixed point where the axis of a tool starts ($\mathbf{P_0}$ is the center of the small opening the tool passes through), and $\mathbf{P}$ the extremity of the tool. Static collision detection between a tool and the polygonal mesh representing the organ can easily be performed by associating an orthographic camera to the tool.

The camera is positioned at point $\mathbf{P_0}$ and the viewing direction is set to $(\mathbf{P_0}, \mathbf{P})$, thanks to the function `gluLookAt()`. Near and far parameters are respectively set to 0 and to $\|\mathbf{P} - \mathbf{P_0}\|$. The tool section is taken into account by setting the left, right, top and bottom parameters of the camera according to the shape of the real tool extremity. The corresponding code is:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();

// compute distance between
// far and near clipping planes
l = norm(P-Po);

// push the orthographic camera on
// projection matrix stack
glOrtho(-s,s, -s,s, 0, l);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();

// move the camera to set eye at Po
// and looking at P
gluLookAt(Po[0], Po[1], Po[2],
   P[0], P[1], P[2],
   up[0], up[2], up[1]);

// redraw the scene with some glNames
// pushed
redraw();
```

For our application, we simply want to detect which faces of the liver are in contact with the tool. Thus we use the *select* picking mode, with one different primitive name per liver face: each `glBegin(GL_TRIANGLES)` is preceded by `glLoadName(t)` where t is the triangle number. At the end of the rendering, the first row of the select array contains the number of hit triangles, then for each triangle items consisting in the z min and max and the face number. The exact coordinates of the intersection points could be obtained using the *feedback* mode.

4. Taking the motion of the tool into account

The simple solution presented in the previous section tests the collision between a static position of the tool and the organ at a given time step. This suffers from the classical flaws of time discretization: if the user hands move quickly, the tool may deeply penetrate inside the organ before being repulsed. It may even cross a thin part of the organ without any collision being detected.

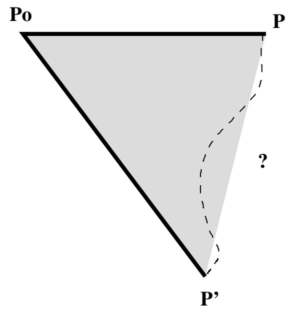

Figure 3. Tool movement between two simulation steps.

In order to avoid these classical problems, we present an extension which takes into account the volume covered by the tool during a time step (we still neglect the dynamic deformations of the organ during this time period). In our model, the tool goes through the patient's abdominal wall at the fixed point $\mathbf{P_0}$, and is able to slide through this point, so its length varies over time. We assume that the active extremity of the tool follows a straight line trajectory from $\mathbf{P}$ to $\mathbf{P'}$. The area covered by the axis of the tool is thus the triangle $\mathbf{P_0}, \mathbf{P}, \mathbf{P'}$ (see Figure 3). Since the tool may be seen as a cylinder of radius s, the volume covered by the tool during the time-interval is obtained by enlarging and thickening the triangle by the distance s. It is thus an hexahedron, as shown in Figure 4. As in the previous section, our aim is to model this volume using *OpenGL* cameras and clipping planes.

The simplest way to do this consists in using an orthographic projection, which parallelepipedic viewing volume corresponds to the bounding box of the hexahedron (see Figure 5): bottom and top, near and far correspond to the hexahedron; two extra clipping planes are used to model the sides $\mathbf{P_0P}$ and $\mathbf{P_0P'}$. However, this naive construction has some flaws. For instance, the orthographic viewing volume will be excessively large when $\mathbf{PP'}$ is far from orthogonal to $\mathbf{P_0P}$ (see Figure 6). The consequence is that a lot of faces will be accepted during the clipping with the frustum, and rejected later during clipping with the user-defined clipping-planes. This increases the cost, since the latter is more computationally intensive than clipping with the canonical viewing volume.

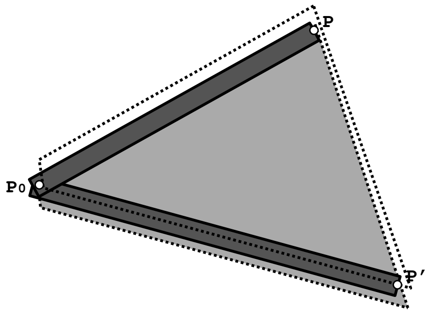

Figure 4. Volume covered by the tool during a time interval.

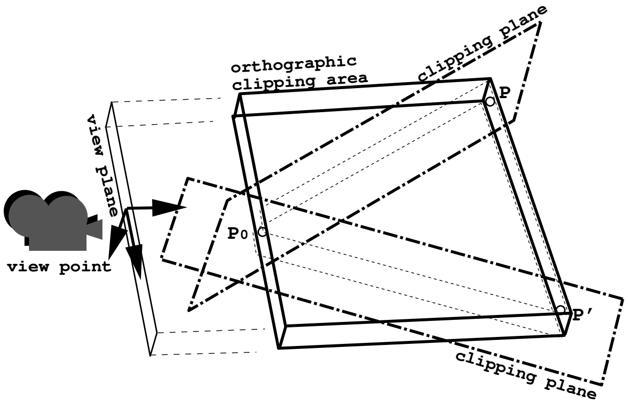

Figure 5. Naive approach using an orthographic camera.

Thus, we construct the test-volume using *OpenGL* in a more complex way, in order to use intermediary volumes that are as small as possible. Our construction is based on a perspective viewing volume whose cone follows the segments $\mathbf{P_0P}$ and $\mathbf{P_0P'}$, as shown in Figure 7. This is done by setting the camera axes to $\mathbf{PP'}$ for the x axis, $\mathbf{P_0P'} \times \mathbf{P_0P}$ for the y axis, and $\mathbf{PP'} \times (\mathbf{P_0P'} \times \mathbf{P_0P})$ for the z axis. As previously, the triangle is enlarged on each side by the tool section s. We set the $(top, bottom)$ interval in the near clipping plane to $2s$. Since the camera is a perspective camera, we have to add two extra clipping planes in order to limit the vertical extent of the volume to

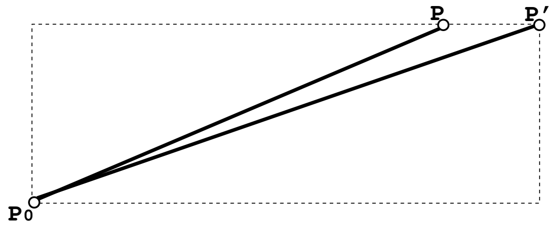

Figure 6. Configuration where the viewing volume is much too large before the addition of the two extra clipping planes.

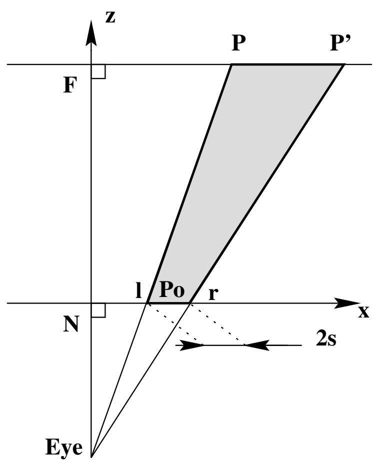

Figure 7. (x,z) plane of the perspective camera

$2s$ everywhere (see Figure 8).

To set the camera to this configuration, the eye position E must be computed from the points $\mathbf{P_0}, \mathbf{P}, \mathbf{P'}$. Let $\mathbf{u}$ be:

$$\mathbf{u} = \frac{\mathbf{PP'}}{\|\mathbf{PP'}\|}$$

We use it to set the left and right limits of the viewing volume in the near and far clipping planes:

$$\mathbf{P_{0l}} = \mathbf{P_0} - s\mathbf{u}$$

$$\mathbf{P_{0r}} = \mathbf{P_0} + s\mathbf{u}$$

$$\mathbf{P_l} = \mathbf{P} - s\mathbf{u}$$

$$\mathbf{P'_r} = \mathbf{P'} + s\mathbf{u}$$

From Thales theorem we get:

$$\frac{\|\mathbf{EP_{0l}}\|}{\|\mathbf{EP_l}\|} = \frac{\|\mathbf{EP_{0r}}\|}{\|\mathbf{EP'_r}\|} = \frac{\|\mathbf{P_{0l}P_{0r}}\|}{\|\mathbf{P_lP'_r}\|}$$

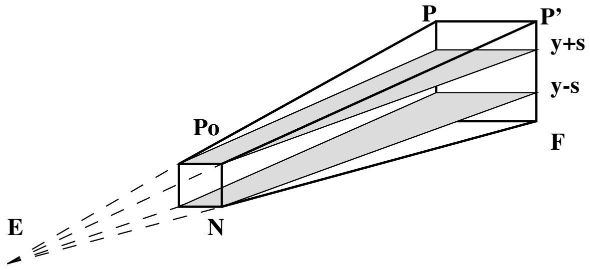

Figure 8. Reducing the viewing volume with clip planes

This yields:

$$\mathbf{E} = \mathbf{P_{0l}} - \frac{\|\mathbf{P_{0l}P_{0r}}\|}{\|\mathbf{P_lP'_r}\| - \|\mathbf{P_{0l}P_{0r}}\|}\mathbf{P_{0l}P_l}$$

Thus we set the *OpenGL* perspective camera parameters to:

$$L = \mathbf{EP_{0l}} \cdot \mathbf{u}$$

$$R = L + 2s$$

$$N = \|\mathbf{EP_{0l}} - L\mathbf{u}\|$$

$$F = \|\mathbf{EP_l} - (\mathbf{EP_l} \cdot \mathbf{u})\mathbf{u}\|$$

$$T = +s$$

$$B = -s$$

We finally add the two extra clipping planes $y = -s$ and $y = s$ depicted in Figure 8. This leads to the following pseudo-code, where `fixed` is $\mathbf{P}_0$, `oldPos` is $\mathbf{P}$, and `newPos` is $\mathbf{P}'$:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
u = (newPos - oldPos)
   /norm(newPos-oldPos);
P0l = fixed - s*u; P0r = fixed + s*u;
Pl = oldPos - s*u; Pr = newPos + s*u;
E = P0l;
E -= norm(P0l - P0r)
   /(norm(Pl - Pr) - 2*s) * (Pl - P0l);
L = dot(P0l-E, u);  R =  L+2*s;
B = -s; T = s;
near = norm (P0l-E - L*u);
far = norm (Pl-E - dot(Pl-E,u)*u);

// define the projection
glFrustum(L, R, B, T, near, far);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();

// clipping planes have to be placed in
// MODELVIEW matrix, but we define them
```

```
// if camera referential, so define them
// BEFORE gluLookAt()
GLdouble plan1[4] = {0,1,0,s};
GLdouble plan2[4] = {0,-1,0,s};
glClipPlane(GL_CLIP_PLANE0, plan1);
glClipPlane(GL_CLIP_PLANE1, plan2);
up = cross(E-Pr, E-Pl);
F = (Pl - dot(Pl-E, u)*u);

// move the camera to set eye at E
// and looking at F, with up set up[]
gluLookAt(E[0], E[1], E[2],
   F[0], F[1], F[2],
   up[0], up[1], up[2]);

// activate the clipping planes
glEnable(GL_CLIP_PLANE0);
glEnable(GL_CLIP_PLANE1);

// redraw the scene with some glNames
// pushed
redraw(NULL);
glDisable(GL_CLIP_PLANE0);
glDisable(GL_CLIP_PLANE1);
```

5. Results

We have done a series of cross-tests to bench our collision methods:

- using our liver geometry (1224 triangles) or a simple tetrahedron (4 triangles),
- testing either static collisions with the tool at a time step ('*static*') or collision with the volume covered by the tool during a time interval ('*dynamic*'), as depicted in Figures 9 and 10,
- testing dynamic collision with different numbers of colliding faces (between 5 and 25 for the liver, between 0 and 3 for the tetrahedron).
- comparing our method with the reference software package *RAPID*[2] implementing Obb trees [5],
- running on various hardwares and graphic accelerators.

Figure 11 sums up the comparisons of computational times between our method and the *RAPID* software on various platforms (each given time is a mean value between ten trials of different collision configurations). Since the same compiler (gcc/egcs) was used on all platforms for compatibility reasons, the results cannot be used for a direct comparison between platforms (gcc uses to produce inefficient code on SGI). The meaningful comparison is the ratio between the two methods depending on the graphics and computational performances of the platform[3].

The Obb tree method used in *RAPID* needs precomputing the hierarchical data structure. In our application where the liver deforms over time, RAPID's datastructure would have to be updated at each time step. Since there is no method for doing so to the authors knowledge, we compared our method with the use of RAPID where pre-computations are redone at each time step. Our method then brings an acceleration factor from 150 on highend hardwares to 12 with a software implementation of *OpenGL* (however,Obb trees would probably give better results if an efficient update algorithm taking advantage of temporal coherence was developed). To be fair, we also computed the acceleration factor without taking RAPID's pre-computation into account. Even in this case which is only applicable to *rigid objects*, our method nearly brings an acceleration factor of five for each collision detection on high-end hardware. All these results are summarized in Figure 12.

6. Conclusion

We have presented a simple and very efficient method for detecting collisions between a general polygonal model and one or several cylindrical tools. Due to its impressive performances, the method is directly applicable in the context of a real time surgery simulator.

Since *no pre-computation* is required, our methods ideally fits to dynamic scenes where objects move and deform over time. As a comparison, the reference code *RAPID*, that is particularly fast, is five times slower assumed that pre-computations are already done, which is not possible for deformable bodies. Our method could thus be useful in many other applications, such as interactive sculpturing where the user manipulates a rigid tool for editing a 3D deformable shape.

The approach could also be generalized to be applied in more general collision configurations: here, one of the colliding objects has a simple geometry. In the general case with complicated shapes, our approach could be used to quickly test the collision between an objet and a non axisparallel bounding box (or even a bounding dodecahedron) surrounding another object. If the second object is embedded into a hierarchy of bounding boxes, this idea could lead to an acceleration of the general Obb tree method. Lastly, since one of the objects can be a mere soup of polygons changing over time, the method could be applied to the

[2]http://www.cs.unc.edu/ geom/OBB/OBBT.html

[3]Concerning our method, we can note that the relatively bad results on the 3Dfx may be due to the fact that this architecture is not pipelined. On pipelined architectures (Onyx and 4D60), the collision detection time is almost constant when the scene size varies from 4 to 1224 triangles.

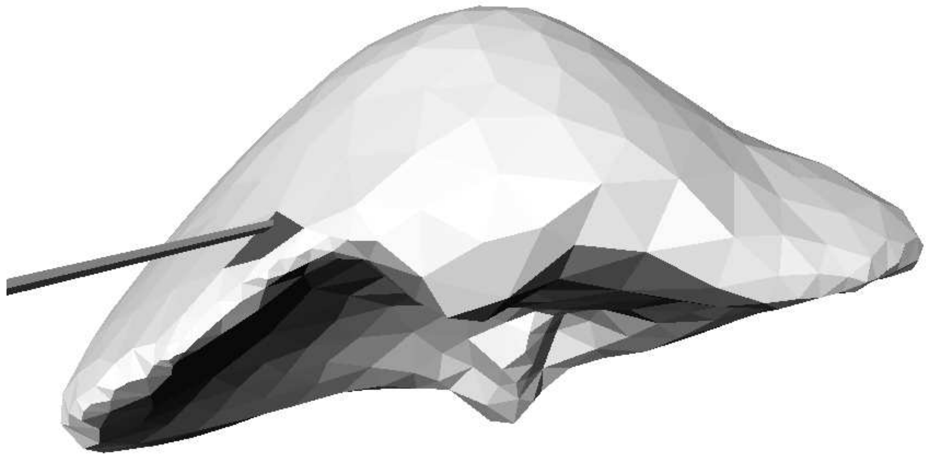

Figure 9. Collision detection between a triangular mesh modeling a human liver and a static position of a tool (which is visualized as a segment).

real-time collision detection between any deformable object (from an elastic surface or volume to a liquid substance) and rigid obstacles embedded into pre-computed hierarchies of bounding volumes.

Moreover, our method is extremely easy to implement (only few dozen lines of codes in an application using *OpenGL* for visualization), portable (*OpenGL* exists on most platforms) and benefits from different graphics hardware as constructors generally offer an optimized implementation of *OpenGL*.

References

[1] D. Baraff. Curved surfaces and coherence for non-penetrating rigid-body simulation. *Computer Graphics*, 24(4):19–28, Aug. 1990. Proceedings of SIGGRAPH'90.

[2] V. Bouma and G. Vanecek. Collision detection and analysis in a physically-based simulation. In *Second Eurographics Workshop on Animation and Simulation*, pages 191–203, Vienna, Austria, 1991.

[3] S. Cotin, H. Delingette, and N. Ayache. Real-time elastic deformations of soft tissues for surgery simulation. *IEEE Transactions on Visualization and Computer Graphics*, (in press), 1998.

[4] A. Garica-Alonso, N. Serrano, and J. Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, 13(3):36–43, 1994.

[5] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. *Computer Graphics, Proceedings of SIGGRAPH'96*, pages 171–180, Aug. 1996. A public domain software package is available at : http://www.cs.unc.edu/ geom/OBB/OBBT.html.

[6] P. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[7] P. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, 1996.

[8] M. Lin and J. Canny. Efficient collision detection for animation. In *Third Eurographics Worshop on Animation and Simulation*, Cambridge, England, Sept. 1992.

[9] M. Lin and D. Manocha. Fast interference detection between geometric models. *The Visual Computer*, 11(10):542–561, 1995.

[10] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, 22(4):289–298, Aug. 1988. Proceedings of SIGGRAPH'88 (Atlanta, August 1988).

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

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

[13] P. Volino, M. Courchesne, and N. M. Thalmann. Versatile and efficient techniques for simulating cloth and other deformable objects. *Computer Graphics*, pages 137–144, Aug. 1995.

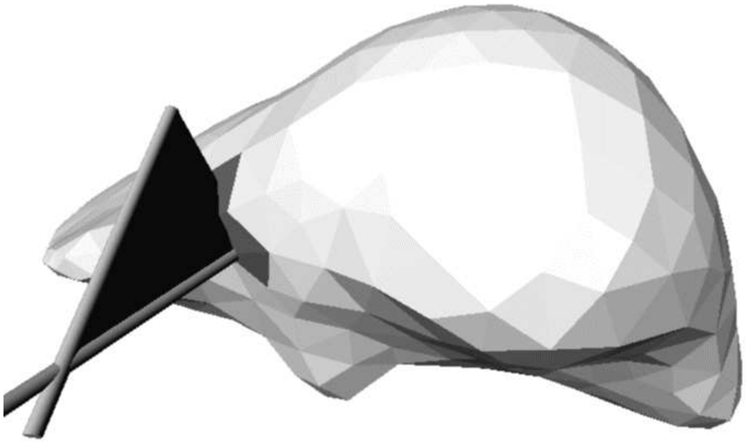

Figure 10. Dynamic collision detection, where the tool motion during a time interval is taken into account (this volume covered by the tool is visualized as a single triangle).

Using our *OpenGL* based method:

processor	R10000	DEC alpha	Pentium2	Pentium2
	195 MHz	500 MHz	333Mhz	333Mhz
graphic	Onyx2 IR	4D60	software	3Dfx Voodoo2
			(Linux Mesa)	(Linux Mesa)
static	0.13 ms	0.09 ms	2.2 ms	1.7 ms
dynamic	0.16 ms	0.11 ms	3.0 ms	2.3 ms

Using the Obb tree method:

processor	R10000	DEC alpha	Pentium2
	195 MHz	500 MHz	333Mhz
Precomputations	24.1 ms	15.7 ms	35.6 ms
static	0.63 ms	0.44 ms	1.0 ms
dynamic	0.76 ms	0.48 ms	1.2 ms

NB: *static* means considering a single position for the tool
dynamic means considering the tool positions during a time interval

Figure 11. Collision detection times

Acceleration factor	Deformable objects		Rigid objects	
	static	dynamic	static	dynamic
SGI Onyx	190	155	4.8	4.75
DEC alpha	179	147	4.9	4.4
Pentium (soft)	16.6	12.2	0.45	0.4
Pentium (3Dfx)	21.5	16	0.59	0.52

NB: *Deformable* objects means considering RAPID's precomputation time,
Rigid objects means igonring RAPID's precomputation time.

Figure 12. Acceleration factor provided by our method w.r.t. *RAPID*