Physical Response to Collision between Deformable Objects

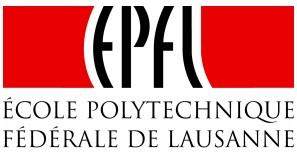
by
Fabiana Benedetti

November 2002

Supervisors: Anderson Maciel and Sofiane Sarni
Director: Prof. Daniel Thalmann

Postgraduate Course on Graphics Visualization and Communication

Virtual Reality Lab (VRLab)
Institute of Computing and Multimedia Systems (ISIM)
School of Computer and Communication Sciences (I&C)
Swiss Federal Institute of Technology (EPFL)



« Voyez-vous dans la vie,
il n'y a pas de solutions.
Il y a des forces en marche :
il faut les créer
et les solutions les suivent. »

Antoine de Saint-Exupéry

Contents

List of Figures

List of Tables

Abstract

When several objects move in a virtual environment, there is the possibility that they interpenetrate. This is an undesired state, because objects do not interpenetrate in the real world. In this manner, a computerized physical simulation must impose a non-penetration constraint so that two objects do not share the same space. In order to do that, collisions between objects should be detected, collision response should be evaluated, and this response should in some way affect the simulation.

This work deals with the collision/contact avoidance. It describes the most widely used methods for preventing interpenetration between contacting virtual objects. Collisions are detected adapting existent libraries, and a penalty method is used to calculate response forces between deformable objects.

A collection of test situations is presented, where the parameters defining the behavior of collision detection and response methods vary and results are compared.

Keywords: Collision detection, collision response, deformable objects

Acknowledgements

I would like to thank my boyfriend for this love, support and guidance at the more difficult moments. Many thanks to my mother and brother for their support and love.

This work could not have been completed without the encouragement, support, enthusiasm and expert guidance of my supervisors Anderson Maciel and Sofiane Sarni. I also would like to thank Professor Thalmann for having allowed me to realize my diploma project at VRLab. Thanks to VRLab's people for their cooperation and good will.

Many thanks to my friends in Brazil for all those e-mails of encouragement and enthusiasm (*Valeu!!!*). Special thanks to my new friends Jelena Debljovic and Belkiss Chattaoui. I also would like to thank my colleagues of the postgraduate course on Graphics Visualization and Communication, Daniel Gutierrez, Lionel Egger, Mich Gaudry, Meeteo Ashvin, Sébastien Hugues and Razakanirina Mahaleo for their help and support.

1 Introduction

1.1 Context

The NCCR-COME[1], more specifically the Project 10: *A generalized approach towards functional modeling of human articulations* [COME 02] aims to provide medical applications to aid on diagnosis of joint disease and planning of surgical interventions. Such applications will be based on a biomechanical model of the tissues present in the joint and a framework for visualization and interaction with this model, both being developed in VRLab. Such model and framework rely on the mechanical and physical properties of biomaterials, and then they depend of the force exchange between the different structures of joints' anatomy to provide correct motion and deformation. Treating these contact forces is the problem of the work presented here.

1.2 Project Goals

According to the situation described above, the correct calculation of forces produced due to contact between different joint elements is essential to the faithfulness of the results.

Thus, the general objective of this diploma project is to create a set of the classes that will generate a 3D surface from the deformation model, detect collisions between the objects, and provide the deformation model with the new contact forces derived from collisions. Such library will be integrated in the framework developed at VRLab, and a test application to evaluate the results will be developed. In this application, a virtual world is specified and simulated, in which all defined objects move and deform according to physical laws. Besides, the user will be allowed to change material properties of the objects in order to verify how objects behave.

1.3 Organization

The plan of this document is presented as follows:

- *Chapter 2 – State of the Art in the Collision/Contact Avoidance.* Within this chapter we review the main previous works for the problem of force calculation to prevent interpenetration of objects.

- *Chapter 3 – Contribution.* This chapter describes the theoretical approach used for the force calculation between colliding objects.

[1] National Center of Competence in Research – Computer Aided and Image Guided Medical Interventions

- *Chapter 4 – Demo Application*. The present chapter describes a software environment where the method to compute non-penetration force is implemented. Besides, we present the main results of this work.

- *Chapter 5 – Conclusion*. We overview the contribution of this work.

1.4 Notation and Units

Throughout this document, we use italic symbols with an arrow on top for vector quantities, such as, $\vec{v}$, $\vec{a}$ and $\vec{F}$; unit vectors have a caret on top, such as $\hat{n}$; and other vectors are denoted by small boldface letters such as **v**. The three basis vectors of a coordinate frame are denoted by x, y and z. Scalars are denoted by small italic letters such as *s*. Matrices are denoted by capital letters such as M and points are denoted by capital boldface letters such as **P**.

In this document SI[2] units are used, e.g., SI unit for force magnitude is Newton (N).

[2] International System or SI (abbreviation for its French name, *Système International*) of units used by scientists and engineers around the world.

2 State of the Art in Collision/Contact Avoidance

2.1 Introduction

In the real world, bodies are controlled by nature's laws that automatically avoid them to interpenetrate. They are made of matter and matter is impenetrable. Just remember that elementary law of Physics that says that two bodies cannot occupy the same space at the same time. In Computer Graphics' virtual environments, however, bodies are not made of matter and consequently are not automatically subjected to nature's laws. It means that they can pass right through each other unless we create mechanisms to impose the same nature's constraints.

In virtual worlds as in the real world, interactions between objects and other environmental effects are mediated by forces applied onto them. In particular, if we wish to influence the behavior of objects, we must do so through application of forces. Thus, a computerized physical simulation must enforce non-penetration by calculating appropriate forces between contacting objects and then use these forces to derive their actual motion.

Over the last two decades, a number of approaches to this problem have appeared in the Computer Graphics literature. Through this chapter, after a description of forces and their types from the physical point of view, we present the currently most widely used approaches for preventing interpenetration between contacting virtual objects.

2.2 Physical Problem

When we throw a ball straight up in the air, how high does it go? Why the apple falls from the tree? Why when we press the trigger of a gun, the projectile is launched in high speed? The answers to these and similar question take into the subject of **dynamics**, the relationship of motion to the forces that cause it.

Force is a quantitative measure of the interaction between two bodies or between a body and its environment. From daily life experience, we can point four properties of force [Young 96][Nave 00]:

1. Force is a vector quantity, since a push or pull have both magnitude and direction.
2. Forces occur in pairs. If object *A* exerts a force on object *B*, then *B* also exerts a force on *A* (action-reaction).
3. A force can cause an object to accelerate. For example, if we kick a football, the ball velocity changes.
4. A force can deform an object and the force exerted on an object is the result of interaction between this object and some other object.

To fully describe the force acting on an object, we must describe the *direction* in which it acts, as well as its *magnitude*, indicating "how much" or "how hard" the force

pushes or pulls. The SI unit for magnitude of force is Newton, abbreviated by an "N", which is defined and may be seen from Newton's second law (see Appendix A) by

$$1\ Newton = 1Kg * \frac{m}{s^2} \tag{2.1}$$

The several forces acting on a body combine by vector addition, that is, two forces $\vec{F}_1$ and $\vec{F}_2$ acting simultaneously at a point **P** of a body have the same effect on the body's motion as a single force $\vec{R}$ equal to the *vector sum* of $\vec{F}_1$ and $\vec{F}_2$ (Figure 2.1) [3]:

$$\vec{R} = \vec{F}_1 + \vec{F}_2 \tag{2.2}$$

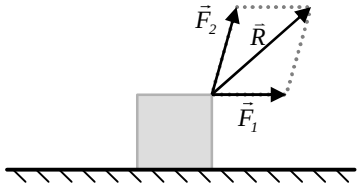


Figure 2.1: Vector sum

The vector sum of all the forces acting on the body is called the **net force** on a body. If the forces are labeled $\vec{F}_1, \vec{F}_2, \vec{F}_3,$ and so on, we abbreviate the sum as

$$\vec{R} = \vec{F}_1 + \vec{F}_2 + \vec{F}_3 + ... = \sum \vec{F} \tag{2.3}$$

We can describe a force $\vec{F}$ in terms of its x- and y-components in the 2 space $\vec{F}_x$ and $\vec{F}_y$. The component version of Eq. (2.3) is

$$R_x = \sum F_x, \quad R_y = \sum F_y \tag{2.4}$$

Once we have R_x and R_y, the magnitude and direction of the net force $\vec{R} = \sum \vec{F}$ acting on the body is

$$R = \sqrt{R_x^2 + R_y^2} \tag{2.5}$$

The forces can have z-components in three-dimensional problems. In this case we add the equation $R_z = \sum F_z$ to Eqs. (2.4). Then, the magnitude of net force is

$$R = \sqrt{R_x^2 + R_y^2 + R_z^2} \tag{2.6}$$

A force resulting from the direct contact of body with another body is called a *contact force*. Examples are a stretched spring exerts forces on the bodies attached to ends; the upward force exerted by a table on a book resting on it and the force on a bone

[3]Each force is represented by an arrow pointing in the direction the force is acting, and having a length equal to the magnitude of the force. The head or the tail of this arrow is placed at the point where the force is acting on the body.

by a contracting muscle. These forces viewed on an atomic scale result from the electrical attractions and repulsions of the electrons and nuclei in the atoms of the materials [Young 96].

In contrast, gravitational forces, as well as magnetic and electric forces can be exerted between objects that are not in contact. The gravitational force on a body is referred to as its weight. This force pulls a body downward along the direction it will fall if it is not supported. Thus, the force of gravity is always found by the equation:

$$F_{grav} = mg \tag{2.7}$$

where g = 9.8 m/s^2 (on Earth) and m = mass (in kg).

2.2.1 Contact Forces

As already mentioned, contact force is a type of force in which two interacting objects are physically in contact with each other. Frictional and normal forces are both contact forces. These and others contact forces are discussed below.

- *Normal Force*

When a body rests or slides on a surface, that surface must exert a contact force on the body. This force can be represented in terms of a force perpendicular to the interaction surface. We call this force perpendicular the *normal force*, denoted by $\vec{F}_n$.

The normal force is equal and opposite of the weight of the body if it is sitting on a horizontal surface as shown in Figure 2.2-a. But the normal force is not always equal to the weight of the body as shown in Figure 2.2-b; it is the force pressing the surfaces together [Young 96][Physics 01].

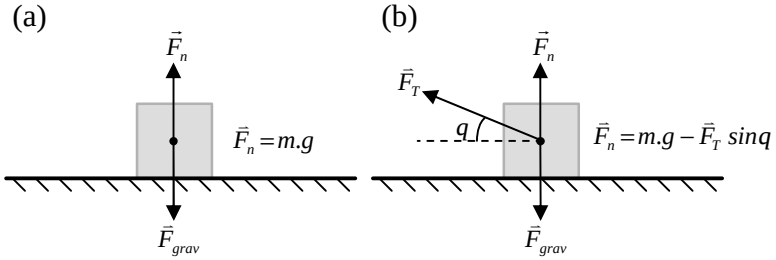


Figure 2.2: Normal force

- *Frictional Force*

A surface can always supply a normal force. However, a surface often also supplies a friction force parallel to the plane.

The friction force is the force exerted by a surface as an object moves across it or makes an effort to move across it [Young 96]. Suppose we push a box along the floor to the right with an external force $\vec{F}$ as shown in Figure 2.3. We know the gravitational force pulls down with a force $\vec{F}_{grav}$, the weight of the box. Besides, the plane responds by

exerting a *normal force* $\vec{F}_n$ and the surface responds by exerting a *parallel force* $\vec{F}_s$. This is the force of *static friction*. The static friction force must be overcome to cause the object to start moving, that is, when we first push the box it does not move, it is held in place by this force of static friction.

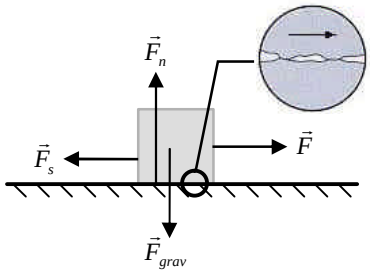


Figure 2.3: Static friction force

If we increase the external force $\vec{F}$, the box starts to move to the right (Figure 2.4) and a *kinetic friction force*, $\vec{F}_k$, occurs that is opposite to the motion of the sliding box. This force is less than the maximum value of the force of static friction, that is

$$\vec{F}_k < \vec{F}_s \tag{2.8}$$

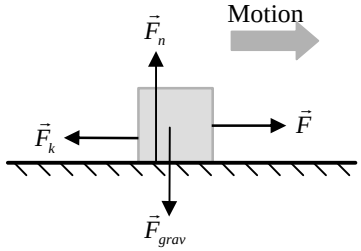


Figure 2.4: Kinetic friction force

The $\vec{F}_s$ and $\vec{F}_k$ are often directly proportional to the normal force. These forces can be calculated using the equations:

$$\vec{F}_s = m_s \vec{F}_n \tag{2.9}$$

$$\vec{F}_k = m_k \vec{F}_n \tag{2.10}$$

where m_s and m_k are called the static and kinetic coefficient of friction, respectively. These coefficients vary widely depending of the nature of the two surfaces and on the degree with which they are pressed together. For example, metal on metal: m_s = 0.10 and m_k = 0.07; copper on glass: m_s = 0.68 and m_k = 0.53.

- *Spring Force*

The spring force is the force exerted by a compressed or stretched spring on any object that is attached to it [Nave 00]. This force acts to restores the object, which compresses or stretches a spring, to its rest or equilibrium position. The magnitude of the spring force is given by

$$\vec{F}_{spring} = -kx \tag{2.11}$$

where k is a constant called force constant (or spring constant) of the spring, and x is equilibrium position relative to the origin. That linear dependence of displacement upon stretching force is called Hooke's law. We can see in Figure 2.5 that the direction of the spring force is opposite to the displacement of the end of the spring to the equilibrium position.

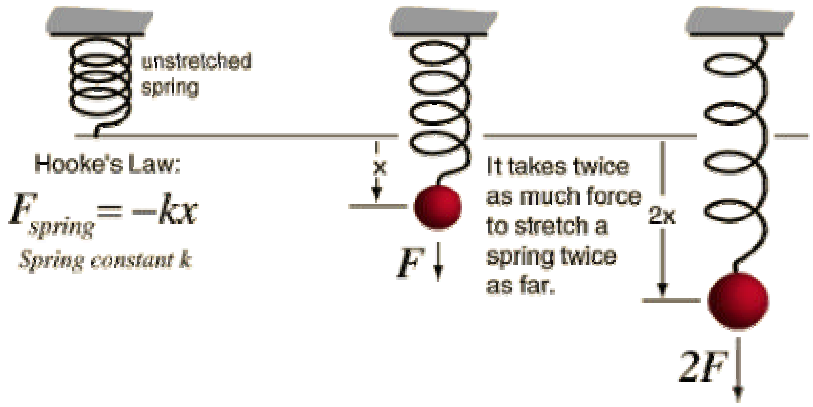


Figure 2.5: Spring force [Physics 01]

2.2.2 Impulse

The impulse of the force is the product of force and the time interval [Young 96]:

$$\vec{J} = \sum \vec{F} \Delta t \tag{2.12}$$

where $\vec{J}$ denote the impulse, $\sum \vec{F}$ is the constant net force and *Dt* is time interval. An impulse is a vector quantity, just like a force, but it has the units of momentum. The SI unit of impulse is Newton•Second (Ns). So, the impulse exerted on an object depends directly on both how much force is applied and for how long the force is applied [Nave 00].

However, when the forces are not constant, the impulse of the force is a vector defined as the integral of the force during a time interval,

$$\vec{J} = \int_{t_1}^{t_2} \sum \vec{F}\, dt \qquad \text{(general definition of impulse).} \tag{2.13}$$

2.3 Existent Methods to Avoid Interpenetration

When trying to avoid interpenetration we need to deal with two types of overlap: *colliding* and *resting contact* [Baraff 98]. We call c*olliding contact* the situation where two objects are in contact at some point **P** and they have a velocity towards each other. This contact requires an instantaneous change in velocity; whenever a collision occurs, the state of an object undergoes a discontinuity in the velocity. In the other hand, we say that objects are in *resting contact* whenever objects are resting on one another at some point **P**. That is, they are touching but their relative contact velocity is zero. In this case, we compute a resting contact force that prevents an object to accelerate.

A variety of approaches have been taken for computing contact reactions, each with their own strengths and weaknesses. This section presents three widely used approaches.

2.3.1 Penalty Method

The physical interpretation of the penalty method is a rubber band that attracts the physical state to the subspace g(x)=0 [4]. The penalty method adds a quadratic energy term that penalizes violations of constraints [Platt 88]. Thus, it converts a constrained optimization problem

$$\textit{minimize } f(x) \quad \textit{such that} \quad g(x) = 0 \tag{2.14}$$

to an unconstrained problem

$$\textit{minimize } f(x) + kg(x)^2 \quad \textit{as} \quad k \to \mu \tag{2.15}$$

where deviation from the constraint is penalized; that is, in the new problem, satisfaction on the constraint is encouraged, but not strictly enforced. In Eq. (2.15) $kg(x)^2$ is called the *penalty function*. The idea of the method is that

- as k grows larger, potential solutions for x must make $g(x)^2$ smaller, to minimize Eq. (2.15), and
- as k goes to infinity, the solution of Eq. (2.15) must satisfy $g(x) = 0$ while minimizing $f(x)$.

The method presents a theoretically firm basis, but, in practice, it is not a very robust numerical method. This occurs because as k grows, Eq. (2.15) can become very poorly conditioned and difficult to solve [Baraff 93].

Let us see how the penalty method can be applied to calculate collision and contact forces. Suppose two objects *A* and *B* collide. The penalty method allows objects in the simulation to penetrate each other. Upon penetration, a temporary spring is attached between the contact points. This spring compresses over a very short time and applies equal and opposite forces to each body so that they will separate. As objects interpenetrate, the forces generated increase with the penetration distance.

[4] $g(x)$ is a scalar function of x which is zero when a constraint is met.

Thus, the penalty force can be written as

$$\vec{f} = -k(x(t) - p(x(t))) \tag{2.16}$$

Here, *f* is the force applied against each object, *p(x(t))* denote the closest point on the surface to *x(t)*, and *k* is the spring constant which is a measure of the spring stiffness.

Often, there is friction or other dissipation in the spring [Barzel 92]. A damping[5] term can be added to help to limit oscillations that springs induce in a model. Thus, the force equation can be then written as:

$$\vec{f} = -(k_s(||\boldsymbol{x}|| - \boldsymbol{x}_0) + k_d\vec{v})\frac{\boldsymbol{x}}{||\boldsymbol{x}||} \tag{2.17}$$

where **x** is the difference vector from the fixed point to the surface point, $\vec{v}$ is the velocity, x_0 is the initial length of the spring and k_s is the spring constant and k_d is the damping coefficient.

The penalty method has been employed in a vast number of simulations in Computer Graphics and Robotics [Terzopoulos 87][Platt 88][Moore 88][McKenna 90][Joukhadar 98][Desbrun 99][Jansson 00] aiming to enforce non-interpenetration constraints. Their applications include the simulation of deformable bodies, cloth, and articulated rigid bodies. Moore and Wilhelms [Moore 88] pioneered the method by introducing penalty forces to prevent bodies in resting contact from penetrating.

The main drawback of this method is finding penalty constants that are effective for different objects in different environments. The choice of theses constants interacts with the choice of the integration time step, because to keep penetration to a minimum, the penalty constants need to be set as high as possible, but this imply large contact forces and these contact forces demand small integration time steps. Consequently, it is computationally expensive, with stiffer spring needing smaller time steps to solve the resulting equations accurately.

Despite their limitations, the penalty method presents advantages as computational simplicity, facility of incorporating static friction models, and, ability to simulate a variety of surface characteristic.

2.3.2 Constraint-based Method

Constraints are used to describe the interactions between objects, which often occur only through physical contact. Constraint-based method computes constraint forces that are designed to cancel any external acceleration that would result in interpenetration. This method results in simulations where interpenetration is completely eliminated. However, it is required solving nonlinear systems of equations [Baraff 89].

Suppose that *S* is a surface in three-space (R^3) and *p* is a particle. The particle's position *p* at time *t* is a function *p(t)* and an external force $\vec{F}(t)$ acts on the particle at time *t*. Now suppose that the particle *p* is constrained to always remain on the surface *S*. Thus, the constraint on the particle can be expressed by

$$C(p(t)) = 0 \tag{2.18}$$

[5] It is the decrease in amplitude of an action or response over time caused by dissipative force [Young 96].

where *C* is a scalar function that models the surface *S* implicitly. This is an example of an *equality-constrained problem* (Figure 2.6-a).

However, the particle *p* can be constrained to lie either on or "above" *S*, then this constraint can be written as

$$C(p(t)) \geq 0 \tag{2.19}$$

Here we have the example of an *inequality-constrained dynamics problem* (Figure 2.6-b).

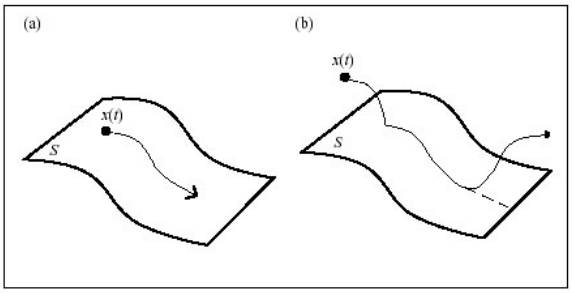


Figure 2.6: (a) C(p(t)) = 0. (b) C(p(t)) ³ 0 [Baraff 93]

Consider again that we introduce a constraint force $\vec{F}_c(t)$ into the inequality-constrained problem. Since $\vec{F}_c(t)$ acts in a direction normal to the contact surface at the contact point, we can write

$$\vec{F}_c(t) = f(t)\hat{n}(t) \tag{2.20}$$

where $\hat{n}(t)$ is surface normal at the point *p(t)*, f is an unknown scalar and must satisfy

$$f(t) >= 0. \tag{2.21}$$

Constraint-based method can be used to solve the problem of resting contact, that is, objects are in resting contact, but are not effectively colliding. For a single contact point, the scalar f (Eq. 2.20) is easily computed and must satisfy the following conditions:

1. $f(t) \geq 0$, since the constraint force must be repulsive; that is, it can push bodies apart, but never pull.
2. $\vec{F}_c(t)$ (Eq. 2.20) must be strong enough to prevent that the objects being pushed towards each other.
3. When the objects begin to separate, $\vec{F}_c(t) = 0$. This condition is written as $f(t)a = 0$ which ensures that if $a > 0$, $f(t) = 0$. Here *a* denotes the acceleration.

As *a* and *f* are linearly related, we can write

$$a = cf(t) + d \tag{2.22}$$

where *c* and *d* are variables of the system. Using Eq. (2.22) we can say that *f* must satisfy the conditions

$$f \geq 0, \quad cf(t) + d \geq 0 \quad \text{and} \quad f(t)(cf(t) + d) = 0. \tag{2.23}$$

Now consider a system of frictionless bodies contacting at *n* different points. For each contact point there will be some force $\vec{F}_c(t)$ normal to the surface at the contact point. For that reason, it is necessary to calculate contact force magnitude, *f(t)*, at each contact point.

Thus, for each contact point p_i between two bodies we have a relative acceleration a_i and a contact force magnitude f_i at time *t*. We will represent the collection of all a_i by the vector **a**, and the collection of all f_i is a vector **f**, in this manner we can write [Baraff 89]

$$\mathbf{a} = A\mathbf{f} + \mathbf{b} \tag{2.24}$$

where A represents the masses and contact geometries of the bodies and **b** represents the external and inertial forces [Baraff 94]. The matrix A and the vector **b** are determined from the known configuration of the system.

At each contact point the same conditions as in Eq. (2.23) must be satisfied, yielding the system

$$\mathbf{f} \geq 0, \quad \mathbf{a} \geq 0 \quad \text{and} \quad f_i a_i = 0 \qquad \text{for } 1 \leq i \leq n. \tag{2.25}$$

Therefore, using the Eq. (2.24) we can rewrite the conditions on f_i as

$$\mathbf{f} \geq \mathbf{0}, \quad \mathbf{a} = A\mathbf{f} + \mathbf{b} \geq \mathbf{0} \quad \text{and} \quad \mathbf{f}^T\mathbf{a} = \mathbf{f}^T(A\mathbf{f} + \mathbf{b}). \tag{2.26}$$

The Eq. (2.26) is known as a linear complementary problem (LCP). A detailed explanation of LCP can be found in [Cottle 92]. Furthermore, Baraff salient that the conditions described in Eq. (2.26) can be considered as a quadratic program (QP) that is, a vector ***f*** that satisfy those conditions is a solution to the QP

$$\min_f f^T(Af + b) \qquad \text{subject to} \quad \begin{Bmatrix} Af + b \geq 0 \\ f \geq 0 \end{Bmatrix} \text{ [Baraff 94].} \tag{2.27}$$

We can find simulation methods that have used LCP to calculate forces between contacting rigid bodies [Baraff 89][Baraff 94][Pang 96][Faure 96][Popovic 00]. According to Baraff [Baraff 89], solving for the accelerations in the contact points and substituting the result into the constraint equations results in a system of equations which can be used to compute the contact forces. So, linear programming techniques are used to formulate and heuristically solve a system of inequality and equality constraints on the forces. This system must assure that the contact forces will prevent the interpenetration and satisfy the Newton's laws (Appendix A).

When there is no friction, the LCP is convex and solutions can be computed using algorithms that run in worst case exponential time but expected polynomial time in the number of contacts. Friction can be incorporated to the algorithms by modifying or adding constraints to the LCP. In [Baraff 94] an algorithm for computing the contact forces between objects with static and dynamic friction is presented. This algorithm is an adaptation of the one described by Dantzig, which is related to pivoting methods for solving linear and quadratic programming. Dantzig's algorithm for solving LCP is described in [Cottle 92]. Baraff's algorithm presents the followings problems:

- *Convergence*. It was not proven with friction and in this case we have no guarantee that the algorithm would terminate;
- *Control of computation time*. An unpredictable number of iterations (unless the contacts are frictionless) is necessary to maintain the previously computed values for the forces and accelerations within the correct bounds.

Faure presented a method to compute resting contact forces based on energy transfer between the bodies in contact that satisfies both the conservation of energy and the inequality constraints [Faure 96]. The first iteration of the algorithm consists of a global dynamic solution involving inertia and external forces that satisfy the conservation of energy. The subsequent iterations consist of global redistributions of energy through the solids. This method simultaneously handles both static and sliding friction, like the approach presented in [Baraff 94], and avoid the problems of convergence and computation time. The former is proven in the frictionless case, and the later occurs when either desired precision or an allowed computation time is reached.

2.3.3 Impulse-based Method

In Physics, when we apply a force on an object, we also exert an impulse on it. Suppose that two objects are in contact at the point $\mathbf{P_c}$ at time t_c, and they have a relative velocity towards each other. Unless the relative velocity is abruptly changed, interpenetration will immediately occur after time t_c. Thus, we apply an impulse, which will instantaneously change the velocities of the two objects. As already mentioned in the section 2.2.2, an impulse $\vec{J}$ is the product of a force $\vec{F}$ and the time interval *Dt* that the $\vec{F}$ acts on an object,

$$\vec{J} = \vec{F}\Delta t. \tag{2.28}$$

If we apply an impulse $\vec{J}$ to a rigid body with mass *m*, then the change in linear velocity *Dv* of the body is

$$Dv = \frac{\vec{J}}{m}. \tag{2.29}$$

Let us see how an impulse is treated in impulse-based dynamic simulation of rigid bodies. Impulse-based method for dynamic simulation was pioneered by Hahn [Hahn 98] and extended by Mirtich and Canny [Mirtich 94]. The central idea of this approach is to model all contacts between objects through a series of impulses [Mirtich 94]. It is based on the treatment of contacts as momentary collisions, where two objects are separated by applying a brief impulsive force. When a collision is detected between a pair of objects, a collision impulse will be calculated. This impulse must prevent interpenetration and obeys certain physical laws relating to friction and energy restitution. The collision response is calculated as an impulse *p* which is applied to one object and, from Newton's third law, an impulse *–p* is applied to other object (see Figure 2.7).

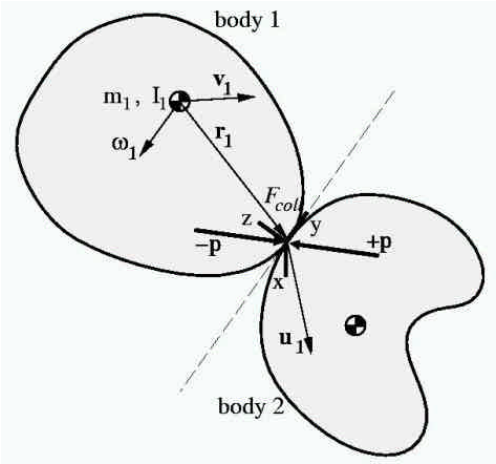


Figure 2.7: Collision between two objects [Mirtich 94]

The frictional force is dependent on the relative sliding velocity of the objects in contact, and this velocity is not constant during a collision. Because of that, the dynamics of the object must be analyzed during the collision to compute the collision impulse. So, at each time frame the initial relative velocity between the two objects, $\vec{u}$, is computed from

$$\vec{u} = \vec{u}_1 - \vec{u}_2 \tag{2.30}$$

where $\vec{u}_1$ and $\vec{u}_2$ are the absolute velocity of the object 1 and of the object 2 respectively. This velocity is calculated as

$$\vec{u}_i = \vec{v}_i + \vec{w}_i \times R_i. \tag{2.31}$$

Here $\vec{v}$ is the linear velocity of center of mass, $\vec{w}$ is the angular velocity of the rigid body around the center of mass, and **R** is the position relative to the center of mass. Then $\vec{u}$ is projected to the collision frame[6]. If u_z is non-negative, no action needs to be taken, because the objects are not in contact with each other; if u_z is negative, a collision impulse must be applied to prevent interpenetration. Thus we numerically integrate[7] $\vec{u}$ using Eq. (2.32).

$$\begin{bmatrix} u_x' \\ u_y' \\ u_z' \end{bmatrix} = M \begin{bmatrix} -m\dfrac{u_x}{\sqrt{u_x^2 + u_y^2}} \\ -m\dfrac{u_y}{\sqrt{u_x^2 + u_y^2}} \\ 1 \end{bmatrix} \tag{2.32}$$

[6] It is established around the collision plane. This plane is defined as follow. If one of the colliding features is a face, this face is used as a collision plane. If vertices or edges are the closest features, the collision plane is perpendicular to the line which represents the shortest distance between them.

[7] Details about the integration method of the Eq. 2.32 can be found in [Mirtich 96] and [Zhang 96].

where M is a matrix dependent only upon the masses and mass matrices of the colliding bodies, and the location of the contact point relative to their centers of mass. m is the coefficient of friction.

During integration[8], u_z will increase until it reaches zero. When this occurs, the point of maximum compression is reached. As can be seen in Figure 2.8, this point is the limit between compression and restitution phase in a collision process, where *f(t)* and *p(t)* are the force and total impulse delivered at time *t* in the collision respectively.

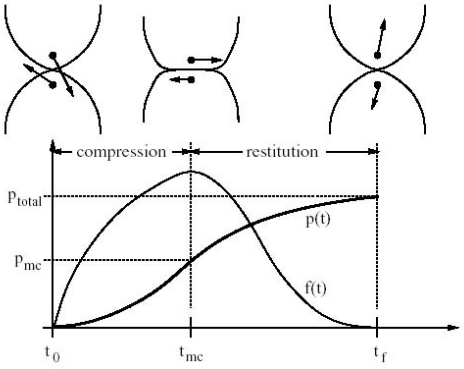


Figure 2.8: Phases of a collision [Mirtich 94]

At this point, the integration variable p_z is the impulse that has been applied. Based in the Poisson's hypothesis, p_z multiplied by $(1 + e)$ gives the terminating value for p_z. Here e is the coefficient of restitution that can range from 0 to 1. A value of 0 means that practically all energy is lost (the objects do not separate after collision; plastic collision), and value of 1 means that no energy is lost (elastic collision) [Mirtich 94].

So, we continue the integration until the termination value $p_z(1 + e)$ in order to find the final value for *Du* that is the change in the contact point velocity of an object over the course of the collision. The final value of the relative velocity *Du* is used to calculate the impulse by reversing Eq. (2.32), that is,

$$p = M^{-1} Du \tag{2.33}$$

With this calculated impulse, the positions and orientations of all the objects are recalculated by assuming ballistic trajectories.

Hahn presented a method to calculate collision impulse with friction at a single point. This method models sliding and rolling contacts using impact equations. The contact forces are not computed explicitly, but occur only as time averages of reaction impulses [Hahn 98].

Mirtich and Cany extended the applicability of Hahn's method to resting contacts, and gave a more unified treatment for multiple objects in contact and a fully general treatment of frictional collisions. The interaction between objects is modeled as a series of tiny micro-collisions that are frequent collisions between objects in continuous contact, for example, a box resting on a floor. The effect of a micro-collision is to reverse the motion direction of the object [Mirtich 94].

[8] If sticking occurs during this integration ($u_x=u_y=0$), the model changes and a simpler set of differential governs the evolution of u.

This work, in turn, has been extended in [Mirtich 95] to support generalized articulated bodies, but the constraints equations are still fundamentally based on equation of motion for simple rigid body systems.

The major disadvantage of this method is its inability to efficiently handle simultaneous and persistent contacts. Consider for example a box resting on an inclined plane (see Figure 2.9). Under impulse-based simulation, the box gradually slides downward. This happens because constant micro-collisions are bouncing the box off the surface at a high frequency. This means that the frictional forces that would normally prevent the box from sliding are only acting intermittently[9] and the box slides, regardless of the properties of the surface.

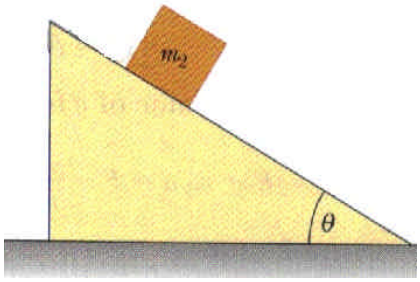


Figure 2.9: A box resting on an inclined plane

Despite of that, impulse-based method presents advantages like simplicity and robustness in comparison with constraint-based method, real-time speed and physical accuracy [Mirtich 94].

2.4 Comparison

We have presented the main methods for calculating forces between two objects in resting contact or collision. We analyze these methods from a critical point of view and present the advantages and disadvantages of each of them which will offer us the guidelines to select a method suitable to our case.

[9] During collisions but not while the box is airborne.

Table 2.1: Comparative analysis of the different methods

	Penalty Method	**Constraint-based Method**	**Impulse-based Method**
Type of Objects	Rigid and deformable	Rigid, deformable and articulated	Only rigid
Concept and Implementation Complexity	Simple	Complex	Medium
Computational Cost	High	Low	Medium
Number of Time Steps Required	High	Low	Low to Medium
Supported Contact Types	Problems with stiff contacts	Problems when contact modes change frequently	Problems with resting contacts
Parallel Computation	Possible	Complicated and difficult	Potential
Physical Accuracy	Depends on time discretization	Accurate in most cases	Accurate
Accuracy Verification	Very difficult	Easy	Easy

Penalty forces are usually computed as elastic forces that depend on the interpenetration between objects. The main problem with penalty method is that it causes instabilities or unwanted vibration. This can happen if the stiffness of contacts is too high, or the force update rate is not high enough. Besides, penalty for rigid bodies are often computational expensive, give only approximate results and may require adjustments for different simulation conditions. In particular, the differential equations that arise using penalty methods may be “stiff” and require an excessive number of time-steps during simulation to accurate results. Additionally, the correctness of the simulation is very difficult to verify. In their defense, penalty method for rigid bodies presents computational simplicity and effectiveness. Add to this, the ease of incorporating static friction models and the ability to simulate qualitatively a variety of surface characteristics. This method for rigid bodies is easily extendible for flexible bodies.

In contrast, ***constraint-based method*** is based on finding exact contacts between the rigid bodies. It gives exact answers and produce differential equations that require far fewer time steps during simulation. The correctness of simulation when using constraint-based method is easily provable because it is directly based on the laws of Newtonian dynamics. Unlike penalty method, this method is computationally more efficient unless the collision is very gentle. In this case, the penalty method is more adequate. However, constraint-based method is much more complex to derive and implement. Besides, the computation is too complicated, the assumptions of perfectly rigid bodies interacting without friction are too restrictive and it must declare each contact to be a resting contact or colliding contact.

Unlike constraint-based method, in the ***impulse-based method*** non-penetration constraints do not exist because the collision is responsible for enforcing separation between two bodies in collision. Besides, in comparison with constraint-based method, it is conceptually and algorithmically simpler. The impulse-based method works well on systems of bodies where the contacts are changing rapidly, but has difficulty to adequately simulate frequent and prolonged contact. In contrast, the penalty method is a

poor choice for simulating brief rigid body collision, which demand high spring constants, but provides an efficient and flexible qualitative model of prolonged contacts.

Finally, there are simulations in which constraint-based method is more appropriate than impulse-based method. Consider a hinge joint (see Figure 2.10). We can model this hinge by micro-collisions between the hinge pin and its sheath. However, due to the enormous amount of collision detection and resolution that would be necessary to model this contact, the simulation would be too slow. However, the constraint-based method is not well suited to situations as shown in Figure 2.11. Under constraint-based simulation, the constraints change as the ball begins travelling up the ramp, leaves the ramp and settles into a roll along the ground. All these occurrences must be detected and processed and new equations of motion for the system must be derived at every transition [Mirtich 95].

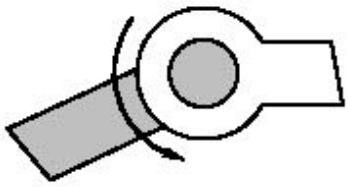


Figure 2.10: A suitable situation for constraint-based method

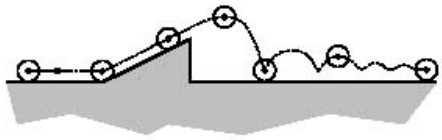


Figure 2.11: A suitable situation for impulse-based method

2.5 Conclusion

Precise detection of contacts and calculation of non-interpenetration forces resulting from object-object interactions is complex, time consuming, and in the general case, an open area of research. There is no single best approach to simulate contact dynamics in arbitrary environments. In fact, the choice of the method to be used usually depends on the application requirements.

In the present work, contact can happen between two deformable objects. So, we believe that the penalty method has a number of characteristics that make it a good candidate for contact resolution when used in simulations of deformable objects. This method is simple and easy to implement, supports simultaneous and persistent contacts, is easily parameterized to simulate a variety of surface characteristics, and is amenable to parallelization. An implementation of this method and the results obtained are presented and discussed respectively in chapter 3 and 4.

3 Contribution

3.1 Introduction

The general goal of this work is to calculate forces to prevent interpenetration of objects in a virtual environment. Two issues are involved in order to achieve this goal: *detecting* that a collision has occurred and *responding* to it. The former is a fundamental problem involving the positional relationship of the objects in the virtual world and is a mandatory issue to prevent interpenetration in a multi-object simulation. The later is a dynamic problem that involves predicting behavior according to physical laws.

Both detecting and responding to collisions are dependent on how virtual objects are represented. The objects concerned with this work are deformable, and the deformation method is based on the discretization of the objects in sets of spherical regions (more details in the section 3.2). Consequently, their surfaces are not smooth and it could cause the collisions to be non-realistic. That is why another step, *generating a smooth surface* on the top of objects, is performed before detecting and responding to collisions.

These three phases (surface generation, collision detection and response forces) are presented and discussed in detail in the sequence of this chapter. However, let us first introduce the deformation model in the section 3.2.

3.2 Deformation Model

The objects we deal with in this project are built on the top of a deformation model currently being developed at VRLab. It is a generalized mass-spring system, where mass points are, in fact, spherical mass regions called molecules. The idea of molecules has first been presented by Jansson in [Jansson 00], and is based on the relationship of the real matter molecules with they neighborhood in a piece of material. Elastic forces are established between molecules by a spring-like connection where properties of materials are taken into account. Figures 3.1 and 3.2 illustrate the objects representation.

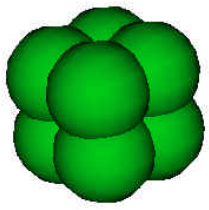


Figure 3.1: Cube – an aggregate of molecules

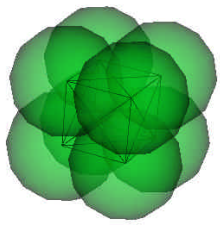


Figure 3.2: Connections between molecules

3.3 Surface Generation

As mentioned in the section 3.2, each virtual object is modeled as a collection of spheres connected by generic springs. The surfaces of these objects are, then, not smooth, which could cause the collisions to be non-realistic. To obtain visual realism during a simulation, it is necessary to apply a surface on the virtual objects. In this work, an implicit surface is created on the top of these objects. To generate this implicit surface, we use a library developed at VRLab [Aubel 02].

Implicit surfaces have been progressively used in modeling and animation during the past decade. Also known as "Metaballs", "Blobbies" or "Soft objects", implicit surfaces are surfaces that are contours (isosurfaces) through some scalar field in 3D [Bourke 97]. An implicit surface is defined by an implicit function, a continuous scalar-valued function over the domain R^3. The implicit surface of such a function is the locus of points at which the function takes on the value zero. For example, a unit sphere may be defined using the implicit function $f(x) = 1 - |x|$, for points $x \in R^3$. Points on the sphere are those locations at which $f(x) = 0$. This implicit function takes on positive values inside the sphere and is negative outside the surface [Turk 02].

The surface generation library used in this work implements a marching cubes algorithm to create a polygonal surface representation of an implicit surface through a set of spheres. This algorithm was adapted by [Aubel 02] who created a library on SGI[10] platform from the basic code written by [Bourke 97]. Our port of this library to PC presented results like the examples of Figure 3.3. However, to refine these results a key issue is to find the best values for two input parameters: mesh resolution and potential field.

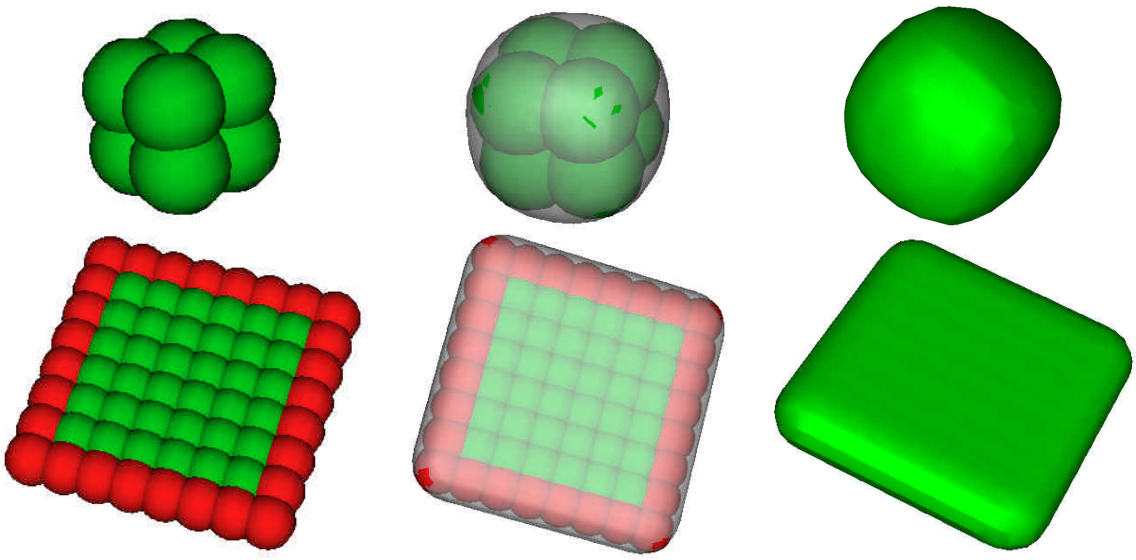
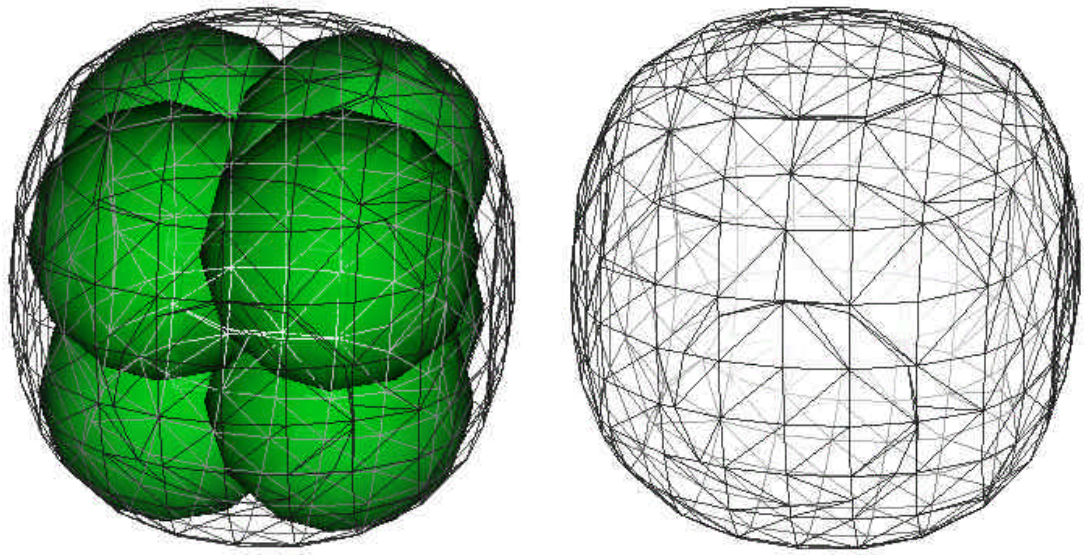


Figure 3.3: Objects with surface

[10] Silicon Graphics, Inc.

- *Mesh Resolution*

The resolution of a surface mesh is the overall spacing between vertices that comprise the mesh. The mesh resolution parameter determines the amount of surface detail the mesh contains and is closely related to the number of vertices, edges and faces in the mesh. A coarse resolution mesh will contain a small number of vertices while a fine resolution mesh will contain a large number of vertices. Therefore, it is necessary that this parameter is set according to shape of each object in a virtual environment, and your needs in terms of precision.



Number of Faces = 648

Figure 3.4: Mesh resolution

In this work, the mesh resolution parameter for each object is automatically calculated according to its volume. The influence of the object's volume about the mesh resolution had been verified from a series of tests realized. In these tests, for each object, we chose a value for the mesh resolution parameter so as to have a more or less smooth surface with few triangles (see Figure 3.4). These values were plotted in the graph bellow from which, we can write the following equation

$$Mesh\,Re\,solution = \frac{\sqrt{volume}}{20} + 0.005\ . \tag{3.1}$$

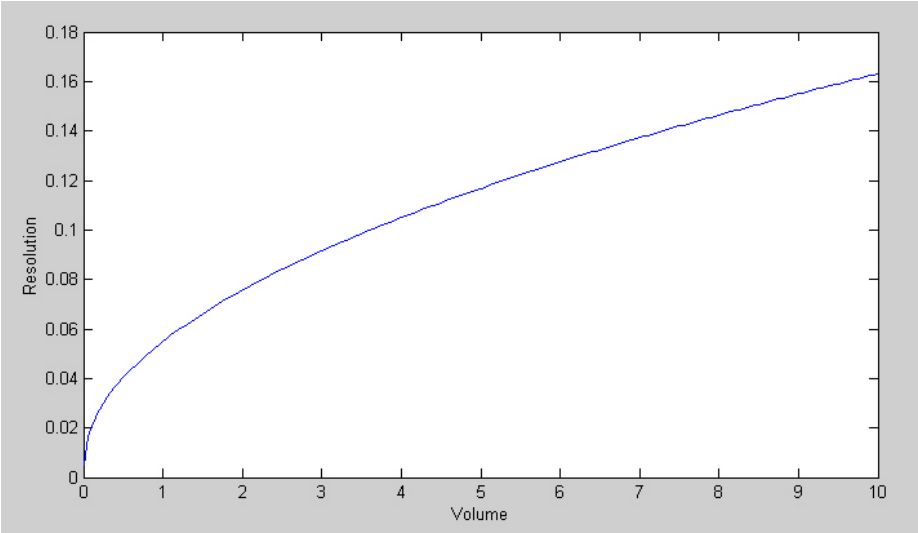


Figure 3.5: Volume and Resolution

- *Potential Field*

Potential field parameter describes the maximum distance between the spheres and the surface. Thus, we need to choose a value for this parameter that best approximates the implicit contours of the spheres. Figure 3.6 shows a sequence of images of the same object in which the potential field parameter varies until the desired approximation is reached, following the surface of the spheres.

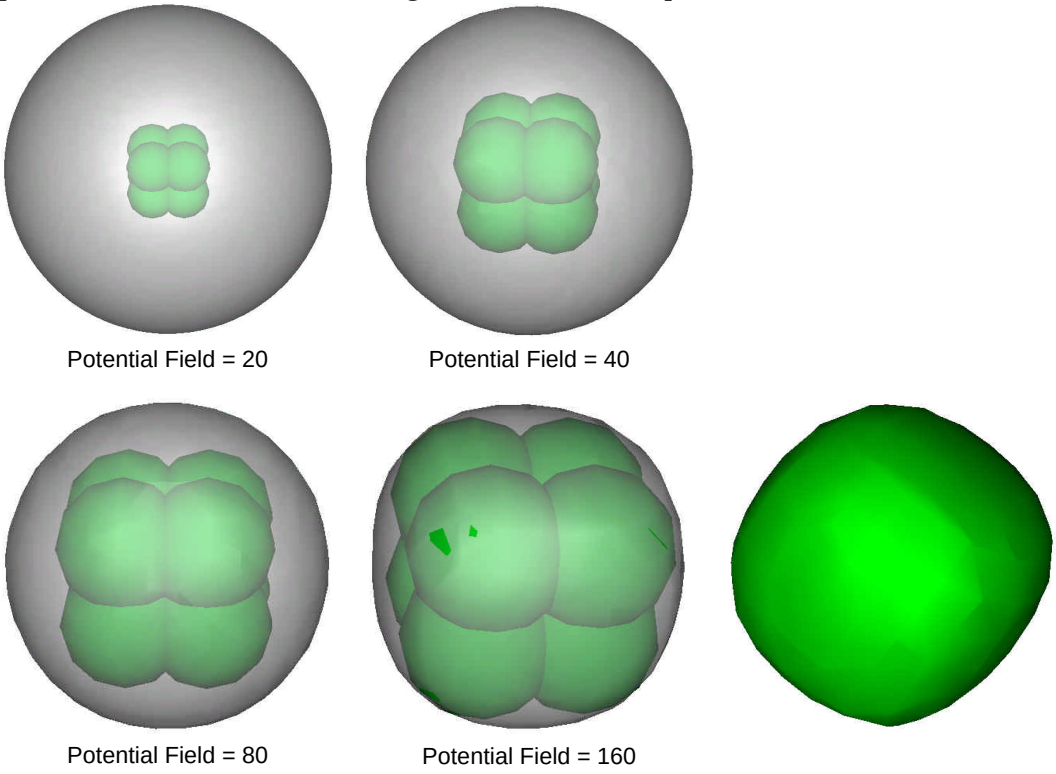


Figure 3.6: Potential field (mesh resolution = 0.005)

This parameter is mostly influenced by the radius of the spheres, that is, the radius describes the maximum distance at which the surface will touch the spheres. To automatically determine a good value for this parameter let us consider a set of objects. For each object we choose the best value for the potential field parameter according to radius of its spheres. The graph in Figure 3.7 shows the relation between radius and this parameter. From this relation, we can define the following equation to calculate the value of the potential field parameter

$$PotentialField = \frac{2}{radius} \quad . \tag{3.2}$$

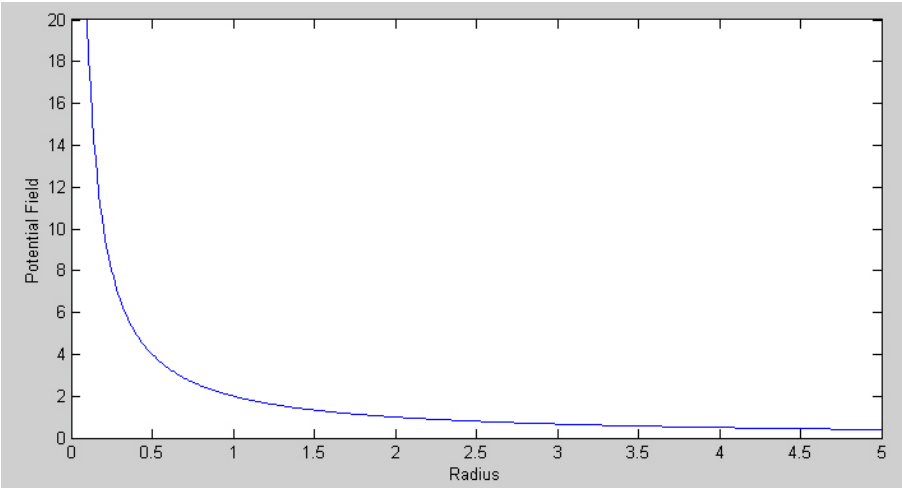


Figure 3.7: Radius and Potential field

3.4 Collision Detection

The problem of collision detection is fundamental to computer animation, robot motion planning, and virtual reality in general. The objective of collision detection is to report a geometric contact when it is going to occur or has actually occurred. So, it is one of the most important aspects in a multi-object simulation to avoid inter-penetration and to calculate resulting forces of interacting objects. In this project we tested two approaches to detect collisions. One of them is based on the spheres that compose the objects, and the other is based on the implicit surface described in section 3.3 and uses V-Collide and RAPID collision detection libraries developed by the Gamma group of the University of North Carolina called [Gamma 02].

3.4.1 Spheres based Collision Detection

One of the simplest and more straightforward methods to detect collisions is the sphere-to-sphere collision detection. Every sphere corresponds to a molecule in the deformation model and is described as a central position and a radius, both obtained from

the molecule. If the distance between the centers of two spheres is smaller than the sum of their radius they collide; otherwise they do not (Figure 3.8).

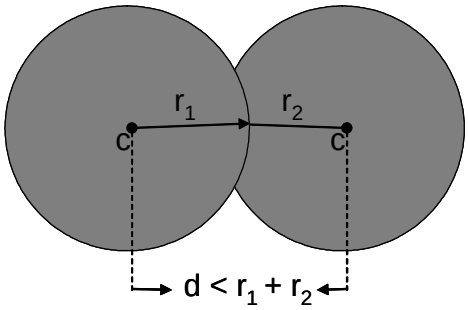


Figure 3.8: Spheres colliding

3.4.2 Mesh based Collision Detection

V-Collide is a "n-body" processor that works with polygon soups. This C++ library uses a fast n-body algorithm to decide which pairs of objects are potentially in contact, and then for each potential contact pair it calls RAPID functions to determine whether the objects actually collide [Hudson 97].

For each collision, V-Collide reports only the objects involved. However, for computing non-penetration force, it is necessary to identify which faces of which objects are involved in the collision. For this reason, we use V-Collide for identifying all scene objects that are in collision and then RAPID to find exactly which pairs of triangles collide.

RAPID is the acronym for "Robust and Accurate Polygon Interference Detection". It is a C++ library that is applicable to polygon soups and operates only with triangles. Unlike V-Collide, RAPID only processes two objects simultaneously. RAPID is based on two algorithms. The first algorithm uses a top-down decomposition technique to build a hierarchy of Oriented Bounding Boxes[11] (OBBs) of an input polygon soup model. The second one realizes collision tests among OBB pairs. These tests consist to verify whether two high-level OBBs overlap; if they overlap, then the algorithm verifies the overlapping of lower level OBBs. Otherwise, the two objects do not collide and the algorithm ends. Thus, RAPID returns a list of contact pairs, where each contact pair is a triangle taken from each of object [Gottschalk 96]. See the guides for basic usage of V-Collide and RAPID in the Appendices B and C, respectively.

3.5 The Non-Penetration Model

Once collision has been detected we need to produce a collision response, that is, to calculate the forces acting on the objects that collide. In this work a very small penetration between two objects is allowed and a separation force is caused by that

[11] An OBB is a rectangular bounding box whose orientation is arbitrary and the resulting hierarchy is called OBBTree.

penetration. This force tries to prevent further penetration and to separate the colliding bodies. The following rules to compute this force are taken into account:

- If object *A* does not penetrate any other object, the force acting on the object *A* is zero;
- If object *A* and *B* penetrate, then forces must be created to act on the objects *A* and *B.* These forces are applied at the point of contact on each object;
- The magnitude of the force is proportional to the penetration depth of *A* and *B*. This depth is the minimum (translational) distance required to separate two colliding objects.
- The direction of the forces is the normal vector to the surface in each object.

The penetration depth is incorporated into a penalty-based formulation to enforce the non-penetration constraint between two deformable objects. As already mentioned in sections 2.2.1 and 2.3.1, penalty method has been widely used because of its simplicity and ease of implementation. Essentially, this method models contacts by placing a spring at each contact point, between the two contacting bodies (see Figure 3.9). Interpenetration is allowed between the objects at a contact point, and the amount of interpenetration is used to introduce restoring or "penalty" force that acts between the objects, pushing them apart.

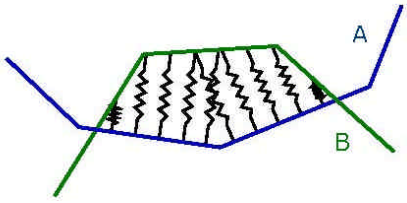


Figure 3.9: Springs inserted between the objects

We do not introduce the spring into the model as a body, but as a force that acts on the objects it attaches. A spring always pulls or pushes along its own length, thus yielding a collinear force pair [Barzel 92]. These forces are based in the Newton's third law ("every action has an equal and opposite reaction") being thus consistent with many natural sources, such as elastic springs, gravitational attraction, etc.

The approaches for estimating penetration depth and penalty force (or string force) are discussed bellow.

3.5.1 Penetration Depth Estimation

As you can see Figure 3.10, the penetration depth *d* is length of the shortest displacement that can cause the separation of the objects *A* and *B*. Assume that the objects *A* and *B* collide at time *t* and that the separation force in direction *d* is applied to the object *B* at the point $\mathbf{P_{BA}}$; it pushes out the object *B* from *A*. The opposite force $-\bar{F}$ in direction *–d* is applied to the object *A* at the point $\mathbf{P_{AB}}$, and pushes *A* away from *B*.

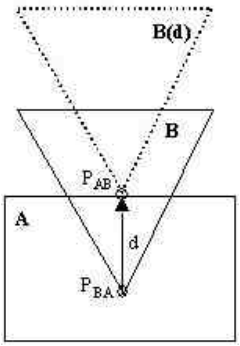


Figure 3.10: Collision geometry

As already mentioned (section 3.3), for each object of a scene a triangulated surface is created. These surfaces are used by collision detection libraries: V-Collide and RAPID. V-Collide determines whether two objects have points in common and then RAPID return a list of triangle pairs in contact. However, these libraries do not provide distance between objects. Thus, we need to compute an approximate penetration depth before calculating the non-penetration force.

We have a list of pairs of faces (f_i, f_j) of each object i and j involved in the collision. Imagine the face f_j represented in Figure 3.11 as being the face in which a vertex **P** of the face f_i collided. For calculating penetration depth d, that here is a distance of the face f_i to face f_j, we project the point **P** onto the plane generated by the face f_j. Point **P'** is the projection of this point on the plane. Thus, we can calculate the distance between **P'** and **P**, and we obtain an approximate value for penetration depth.

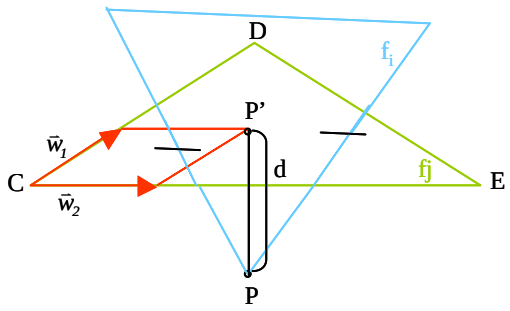


Figure 3.11: Face f_j

The projection plane was defined from the following equations

$$\vec{w}_1 = \frac{\overrightarrow{CD}}{||\overrightarrow{CD}||} \cdot ||\overrightarrow{CP}|| \cdot (\overrightarrow{CP} \cdot \overrightarrow{CD})$$
$$\vec{w}_2 = \frac{\overrightarrow{CE}}{||\overrightarrow{CE}||} \cdot ||\overrightarrow{CP}|| \cdot (\overrightarrow{CP} \cdot \overrightarrow{CE}). \quad (3.3)$$

Then, using the Eqs. (3.3), we find the projection of the point $\mathbf{P'_{BA}}$ in this plane.

$$P' = \vec{w}_1 + \vec{w}_2 + C, \quad (3.4)$$

and the distance between these points is calculated from

$$d = P - P' \tag{3.5}$$

Now suppose the situation where two or three vertices of one face are penetrating the other object. In this case, we have to calculate the penetration distance for both them and choose the greatest between the calculated distances to be the penetration depth.

For the sphere-to-sphere collision, penetration distance is calculated as the difference between distance of the central positions and the sum of the radius of the two involved spheres.

3.5.2 Penalty Force

Penalty method (section 2.3.1) is based on a dependency between the non-penetration force and the penetration depth. The penalty force model considered here assumes that non-penetration force $\vec{F}_{np}$ depends on penetration depth *d* as follow

$$\vec{F}_{np} = \begin{cases} -kd\vec{n}, & if\ d > 0 \\ 0, & if\ d \le 0 \end{cases} \tag{3.6}$$

where *k* is a positive constant (called penalty coefficient) and $\vec{n}$ is the non-penetration force direction. Physically *k* corresponds to a stiff spring, temporarily placed between objects during the collision. In this work the value *k* is calculated using following equation

$$k = m * K \tag{3.7}$$

where *m* is lightest object mass and *K* is a arbitrary positive constant that was chosen from the analysis of the object behavior over the course of a simulation.

Using the Eqs. (3.6) and (3.7), we calculate $\vec{F}_{np}$ for each pair of colliding face. Thus, we write the non-penetration force for the face f_i as

$$\vec{F}_{np_i} = -kd\hat{n}_j \tag{3.8}$$

where $\hat{n}_j$ is the normal vector to the face f_j. For the face f_j, the direction of the $\vec{F}_{npj}$ will be in the direction of the normal vector to the face f_i as shown in the equation

$$\vec{F}_{np_j} = -kd\hat{n}_i. \tag{3.9}$$

Given those equations we propose the following algorithm:

Algorithm to calculate Non-Penetration Force to Mesh Collision

```
compute the spring constant k from Eq. 3.7.
For each pair of contacting face (fi, fj)
  compute the penetration depth according to section 3.5.1
  compute the non-penetration force for the face fi using the Eq. 3.8
  compute the non-penetration force for the face fj using the Eq. 3.9
end
```

Figure 3.12: Algorithm to calculate non-penetration force to mesh collision

For the sphere-to-sphere collision, non-penetration force is calculated as described in the following algorithm:

Algorithm to calculate Non-Penetration Force to Spheres Collision

For each pair of contacting sphere (s_i, s_j)
  compute the spring constant *k* from the elasticity of the molecule material.
  compute the penetration depth according to section 3.5.1
  compute the non-penetration force for the sphere s_i using the Eq. 3.8
  compute the non-penetration force for the sphere s_j using the Eq. 3.9
end

Figure 3.13: Algorithm to calculate non-penetration force to sphere-to-sphere collision

Afterwards, all non-penetration forces are passed as input to the deformation model that will calculate itself the correct deformation and motion of the objects in virtual environment.

4 Demo Application

4.1 Introduction

In this chapter, we show the main results of this work. We describe a demo application where virtual worlds can be specified and simulated, and objects can move and deform according to non-penetration forces presented in the chapter 3. Furthermore, we show some experiments realized.

4.2 Description

4.2.1 Architecture

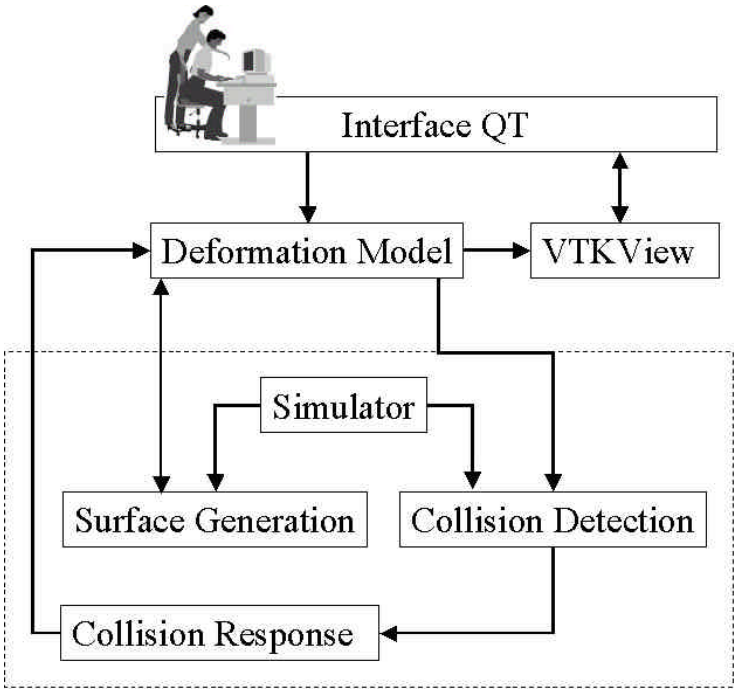


Figure 4.1: Application architecture

Demo application architecture is shown in Figure 4.1. It consists of the following modules:

- ***Interface QT***. A graphical user interface (GUI) was created using QT Non-Commercial version 2.3.0. QT is a multi-platform C++ GUI application framework from Trolltech. This framework is fully object-oriented, easily extensible and allows true component programming [Trolltech 01]. This module is the link between the user and the application. It handles events generated by the user and calls the deformation

model and/or the VTK View module, which take the appropriate action to react to the user. This module receives a signal from the VTK View module to update the image on the 3D window.

- ***Deformation Model***. This module is external to this project; it is just used here. It manages the objects motion and deformation. It is controlled by the simulator, which requests temporal changes in the model, and its state can be read by the VTK View module for visualization purposes. In addition, it keeps a relationship with the collision detection module – which inspects objects positions – and with the collision response module – which send it new non-penetration forces.

- ***VTK View***. This module shows the scene in the 3D window taking into account the information received from the deformation model and the interface module. This visualization is done using VTK (*Visualization ToolKit*). VTK is an open source, freely available C++ class library that supports 3D graphics and visualization [Kitware 01].

- ***Simulator.*** This module simulates a scene. It is responsible by the control of the simulation time and events that can be of two types: user interactions and calls for methods of the modules: surface generation, deformation model and collision detection.

- ***Surface Generation.*** The generation of an implicit surface on the top of the deformable objects is performed by this module (section 3.3).

- ***Collision Detection.*** The task of this module is to detect collisions between objects of a scene and create a list of pairs of colliding faces. Each of these pairs is send to the response force module. Depending of the type of collision detection method defined in interface, this module uses different routines. These routines are described in section 3.4.

- ***Response Force.*** This module handles the effects of the collision between deformable objects. It receives a list of pairs of colliding faces of the collision detection module and calculates a reaction force for each pair of faces using the penalty method (section 3.5). Theses forces are sent to the deformation model.

4.2.2 Interface

The application interface has been designed to provide some necessary tools to evaluate the behavior of the objects in a virtual environment (see Figure 4.2). The various components of the interface and their interactions in the context of the whole application are discussed below.

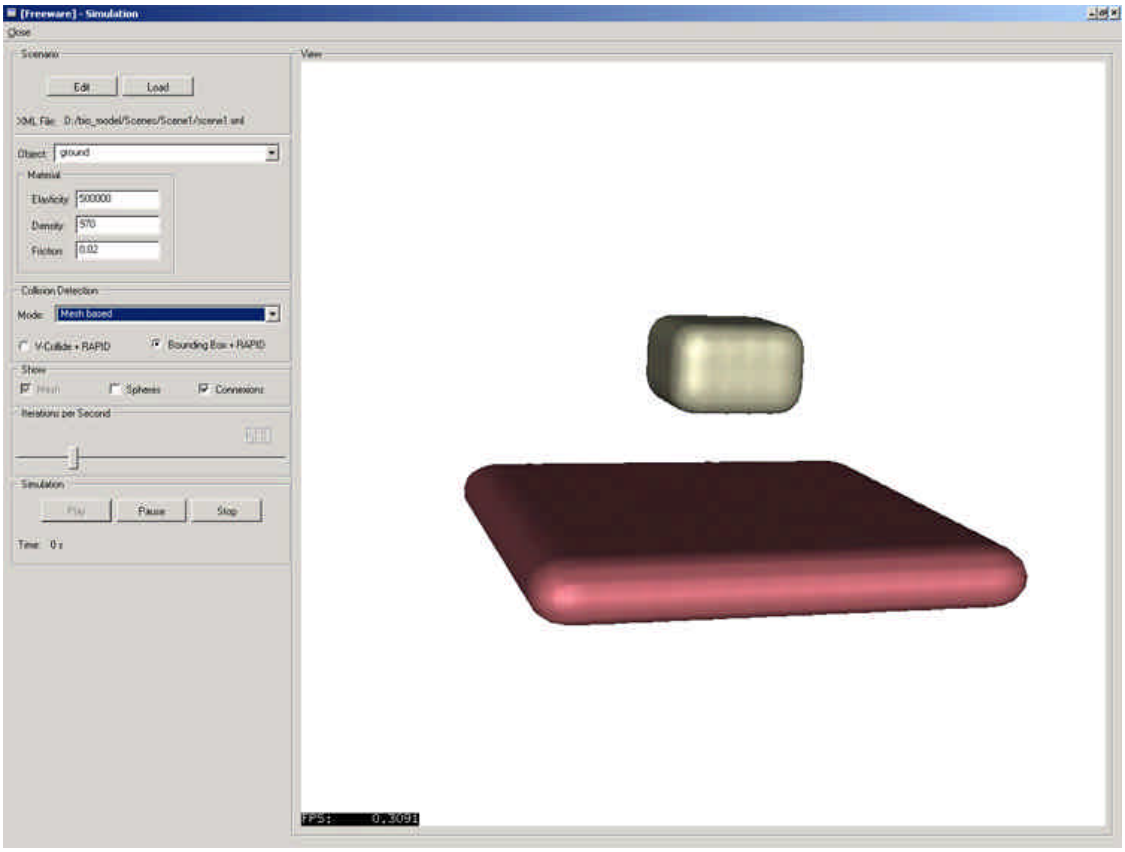


Figure 4.2: Example of the main window

- ***Scene loading***

The user specifies its virtual environment by means of a XML[12] file (see an example in Appendix D). This file defines each virtual object as a set of molecules. Each molecule has its position, radius and proprieties of its material (such as elasticity, friction constant and damping constant). This file also defines some simulation parameters as the number of iterations per second and duration of the simulation. All these information about the virtual environment are loaded into the deformation model and then the user can run the simulation.

The application provides also a XML Editor to the user to create/edit a XML file (Figure 4.3).

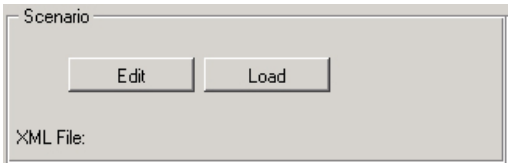


Figure 4.3: Scene loading

[12] Extensible Markup Language

- ***Virtual objects editing***

Once the deformation model is loaded, the user can access the information of each object of the virtual environment and change the following material proprieties: elasticity, density and friction constant (see Figure 4.4). These changes can be realized during the simulation.

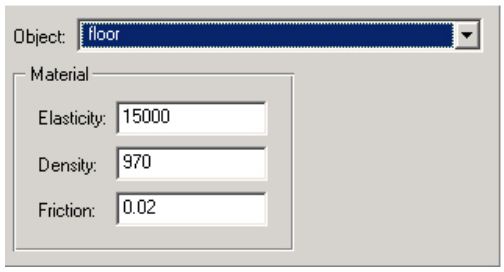


Figure 4.4: Virtual objects editing

- ***Collision detection mode***

The user can choose the type of collision detection that will be used by the collision detection module. As we see in Figure 4.5, there are two methods: spheres-based (section 3.4.1) and mesh-based (section 3.4.2). The former method is a trivial sphere-to-sphere test, and the later uses RAPID library to report the colliding faces between two objects. However, to identify which among all objects are in contact, the user can choose either V-Collide library or Bounding Box. The collision detection mode can be changed during the simulation.

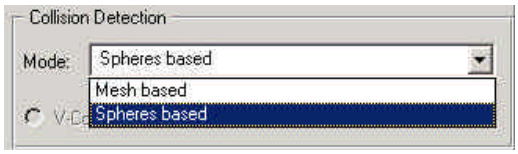


Figure 4.5: Spheres-based collision detection

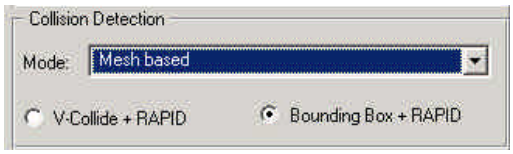


Figure 4.6: Mesh-based collision detection

- ***Scene visualization***

A scene is visualized in a 3D window where the objects can be seen in different modes according to the following options: mesh, spheres and connexions (see Figure 4.7 and Figure 4.8). It is important salient that collision detection mode determines the visualization option. If collision detection mode is mesh-based, an implicit surface will be generated to detect collisions. These options can be modified on the course of the simulation.

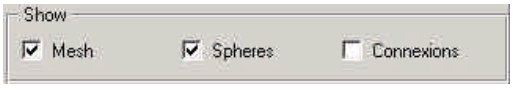


Figure 4.7: Scene visualization

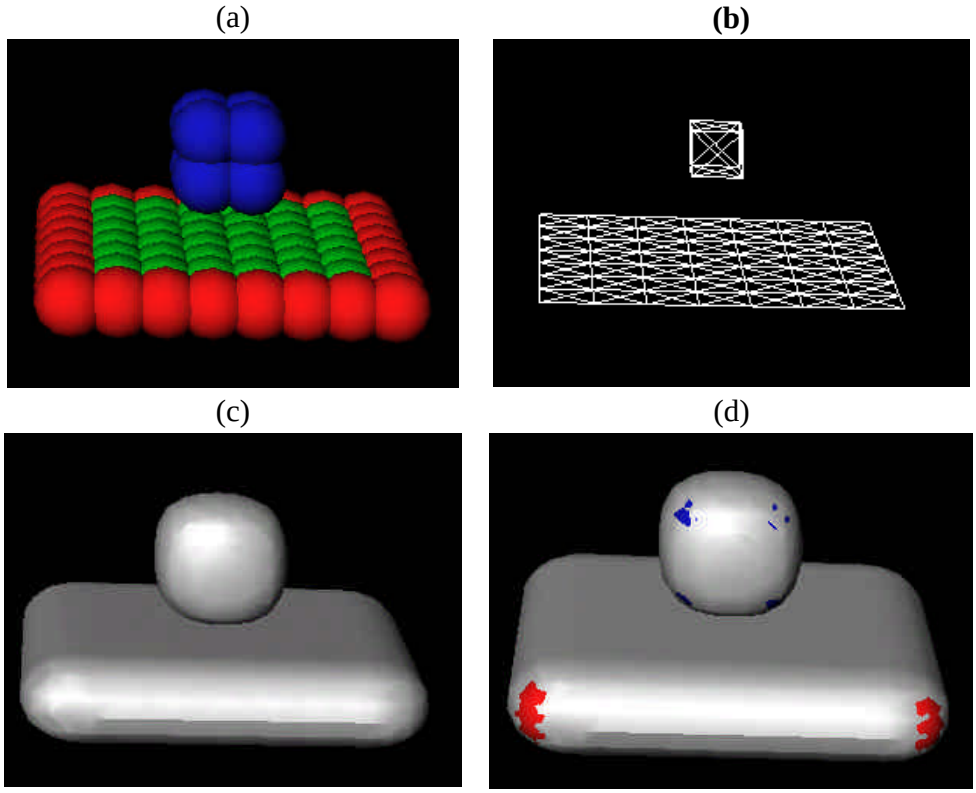


Figure 4.8: Show: (a) Spheres (b) Connexions (c) Mesh (d) Mesh and Spheres

- ***Iterations per second***

The size of the time-step used by numerical integration can be redimensioned on the fly by means of this slider bar that defines the number of iterations to be performed per second of simulation.

Figure 4.9: Iterations per second

- ***Simulation***

Once the scene has been loaded, we can simulate it. Figure 4.10 shows the buttons used to control the simulation.

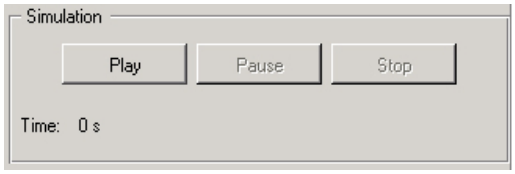
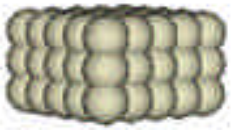


Figure 4.10: Simulation

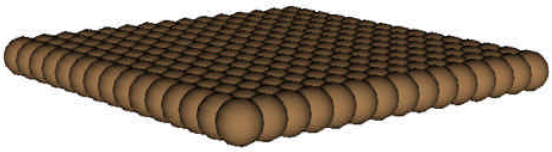
4.3 Experiments

The experiments of sections 4.3.1 and 4.3.2 present the simulation of a box (Figure 4.11) that is pulled towards the ground (Figure 4.12) by a gravitational force. The goal of these experiments is to analyze the behavior of the resulting contact force between these objects.



- 75 molecules;
- 426 connections;
- Density = 2500 kg/m^3;
- Elasticity = 5000 N/m^2;

Figure 4.11: Box



- 225 molecules;
- 812 connections;
- Density = 970 kg/m^3;
- Elasticity = 500000 N/m^2;

Figure 4.12: Ground

4.3.1 Scene without Implicit Surface

The volumes of the objects of this scene are represented by the volume of the spherical molecules that compose them. Thus, such objects are not covered by a surface mesh and collision detection is performed based on spheres. Figure 4.13, shows a sequence of reference key-frames collected during the simulation.

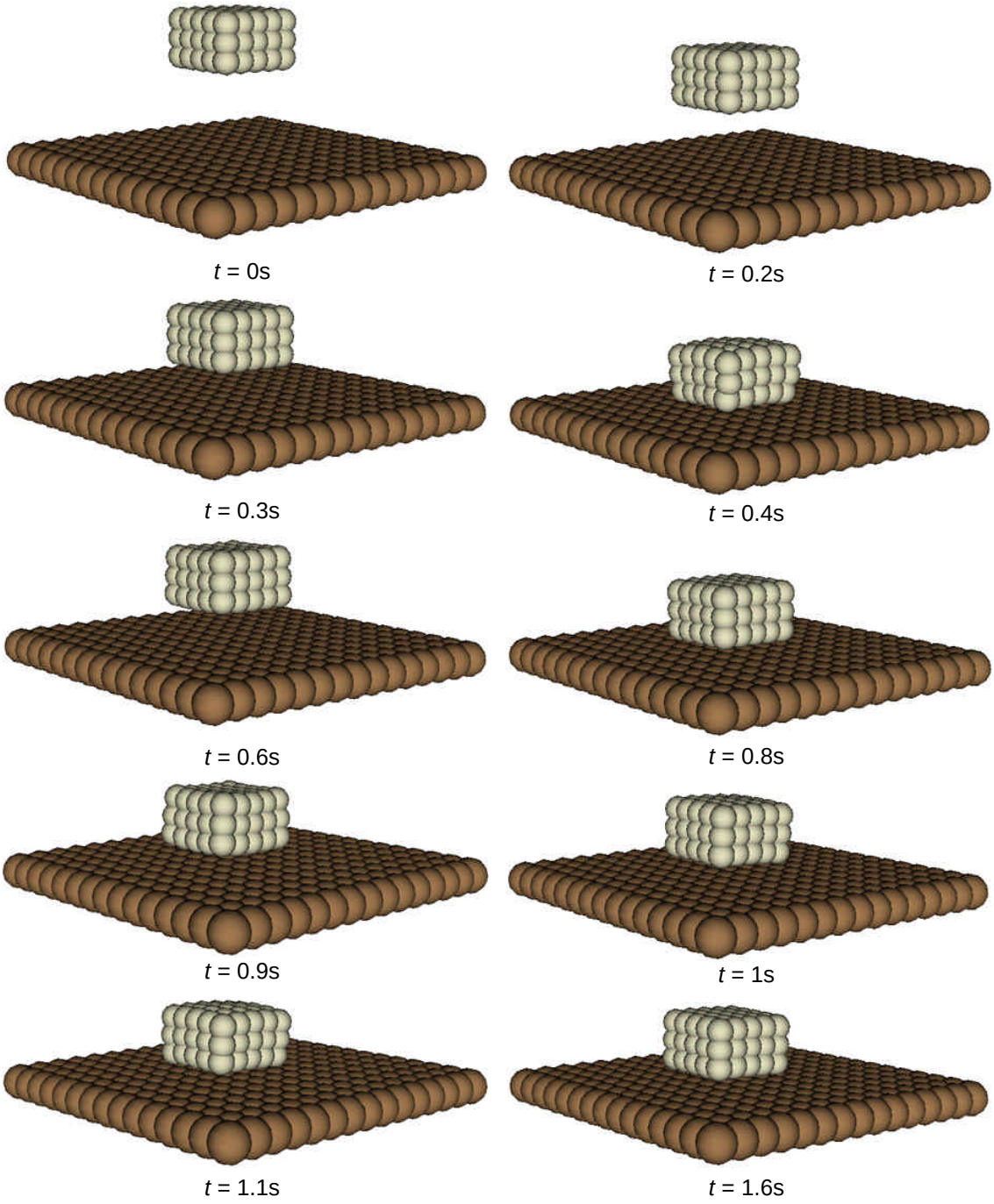


Figure 4.13: Scene without implicit surface - Reference key-frames

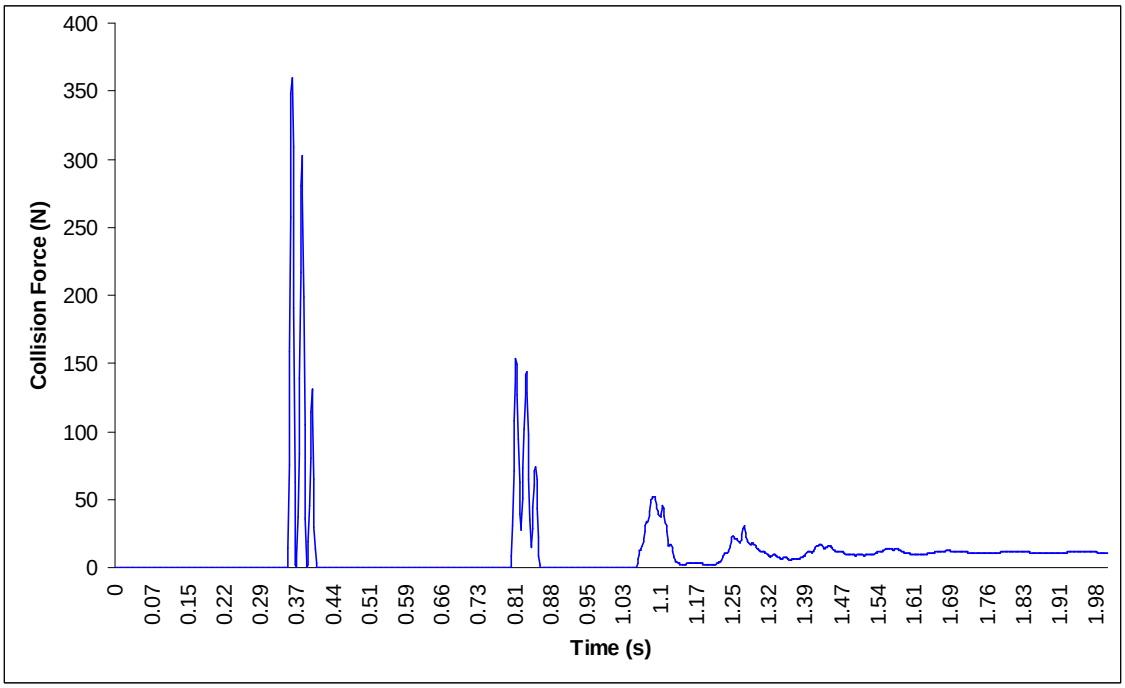


Figure 4.14: Scene without implicit surface - Collision force evolution

The graph below shows the minimum y-coordinate of the box object and the maximum y-coordinate of the ground on the course of the simulation. Once the surfaces of the objects are not smooth, spheres find equilibrium position in the ditches between spheres of the other object, and consequently the maximum and minimum y-coordinates overlap in the graph. That is why the box curve passes bellow the ground one, while on the images we do not see such penetration.

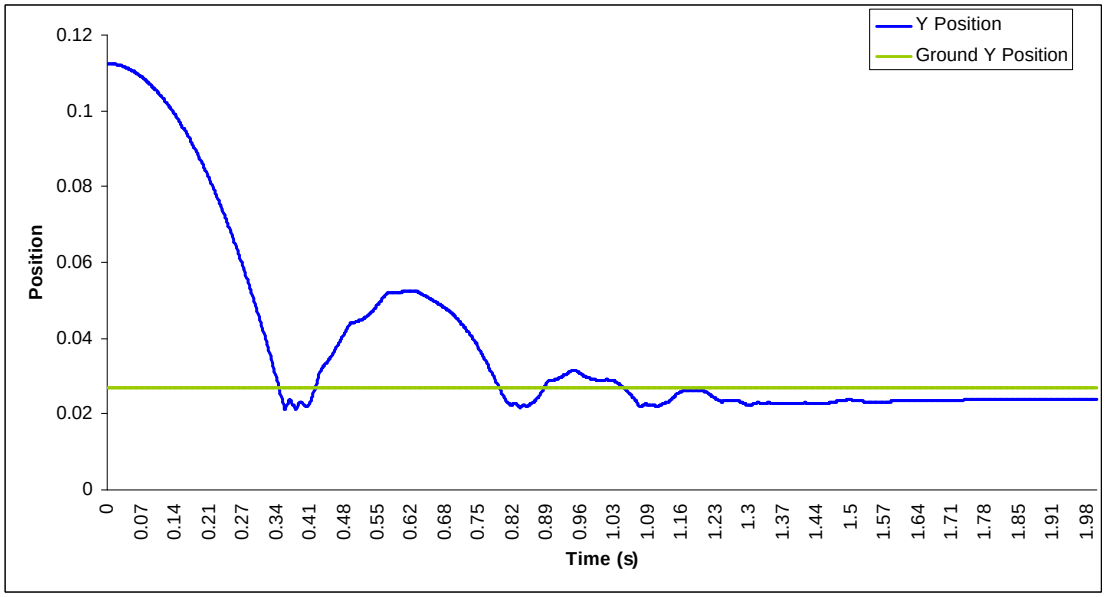


Figure 4.15: Scene without implicit surface – y-coordinate position

We modify the density and elasticity of the box object in order to analyze the influence of this modification on the behavior of the collision force. The results of this experiment are verified in the graphs below.

- *Object's Density Modifying*

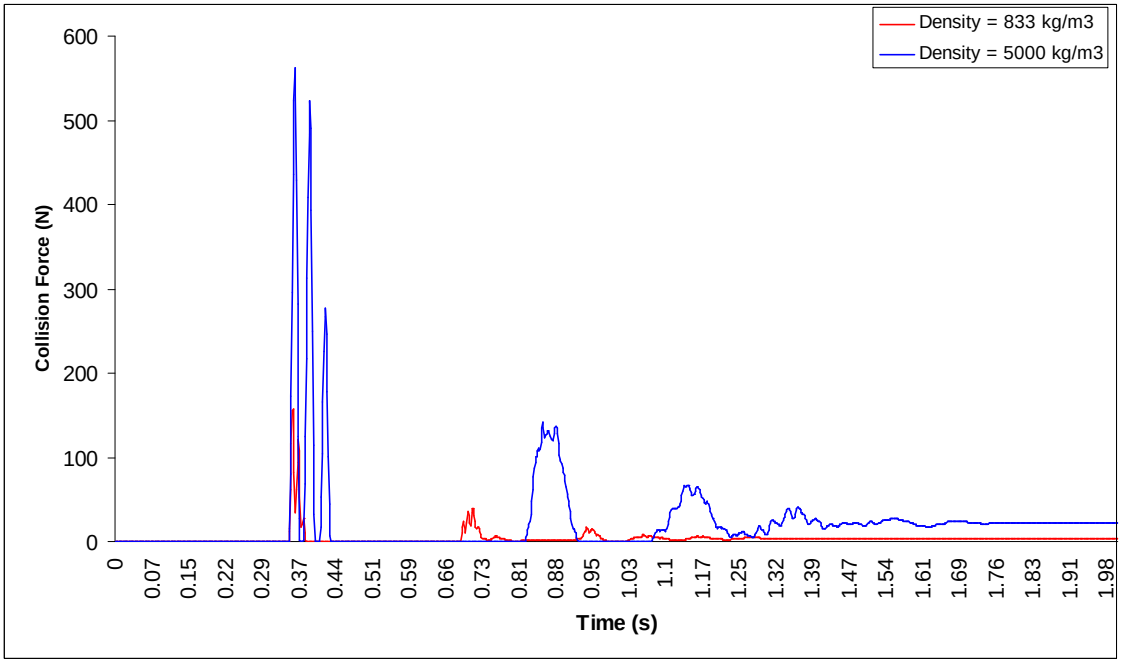


Figure 4.16: Collision force evolution varying the density of the box object (spheres collision)

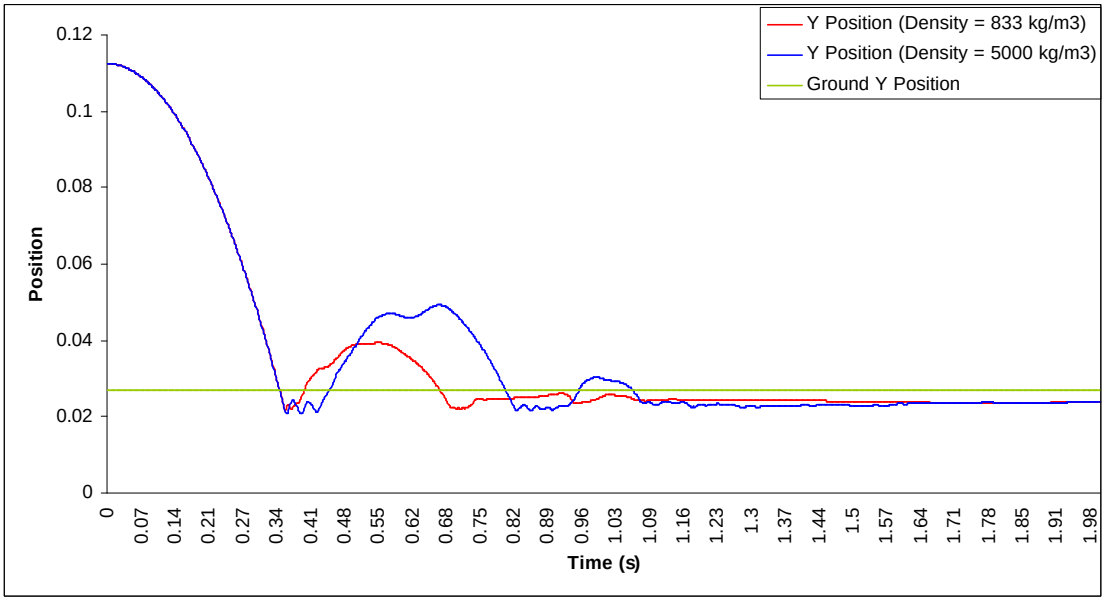


Figure 4.17: Evolution of the y-coordinate position varying the density of the box object (spheres collision)

- *Object's Elasticity Modifying*

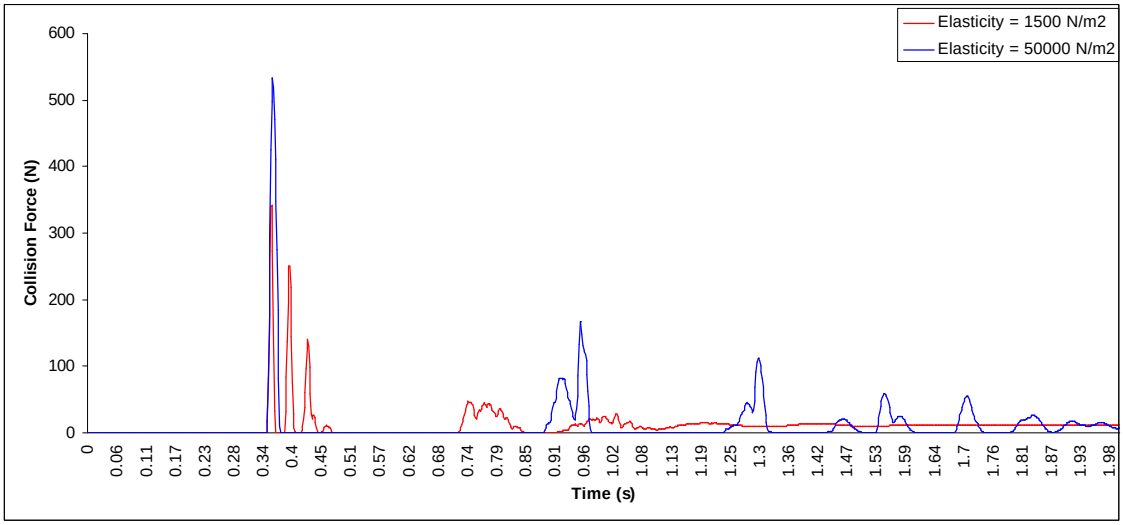


Figure 4.18: Collision force evolution varying the elasticity of the box object (spheres collision)

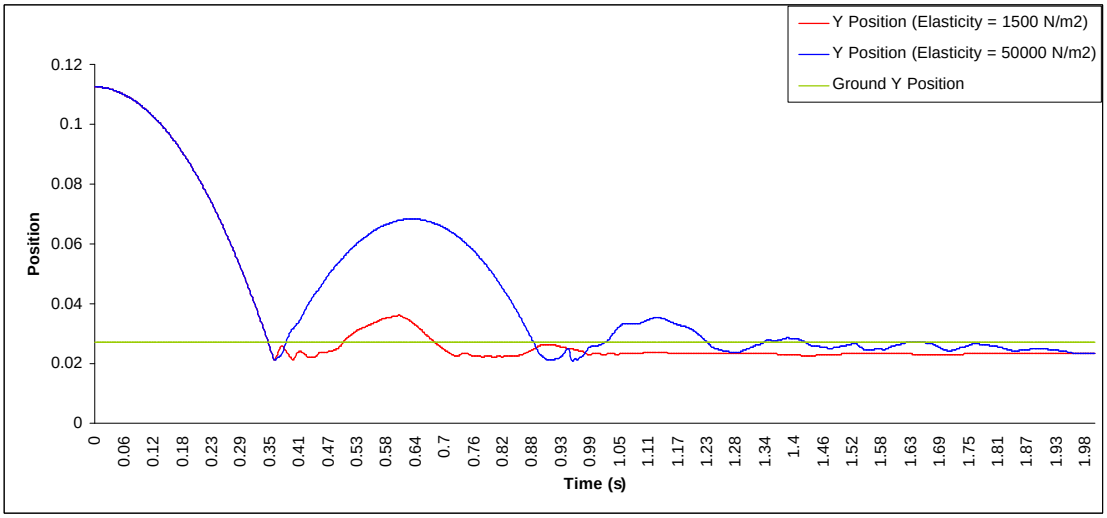


Figure 4.19: Evolution of the y-coordinate position varying the elasticity of the box object (spheres collision)

4.3.2 Scene with Implicit Surface

In this experiment, an implicit surface is generated on the top of each object of the scene. The box surface is composed of 2820 triangles and the ground surface of 9940 triangles. Besides, the collision detection is done using the V-Collide and RAPID libraries.

Figure 4.20 shows snapshots from a simulation sequence where the gravitational force pulls a box downwards.

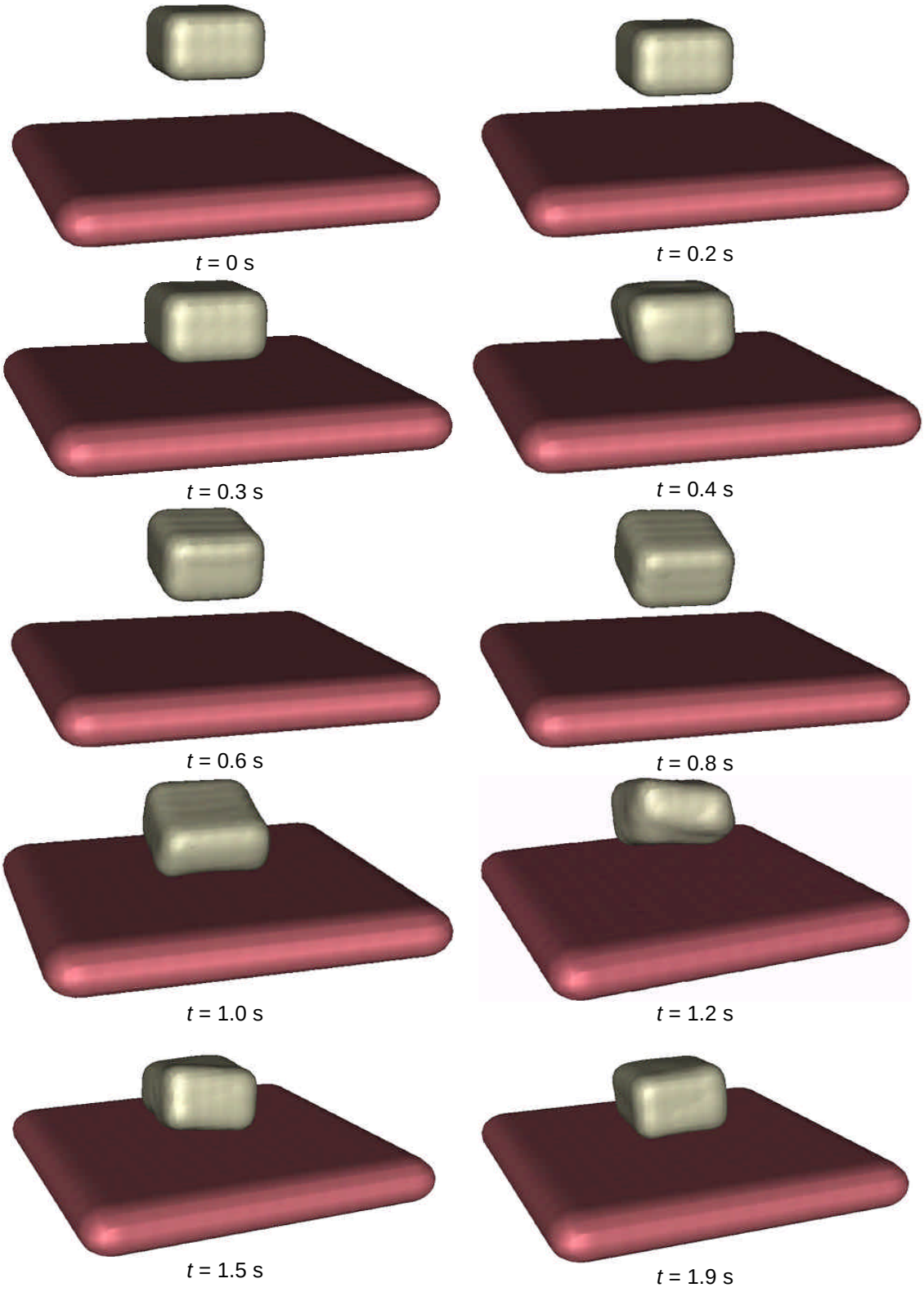


Figure 4.20: Scene with implicit surface - Reference key-frames

The graph below shows collision force evolution during the simulation described by Figure 4.20.

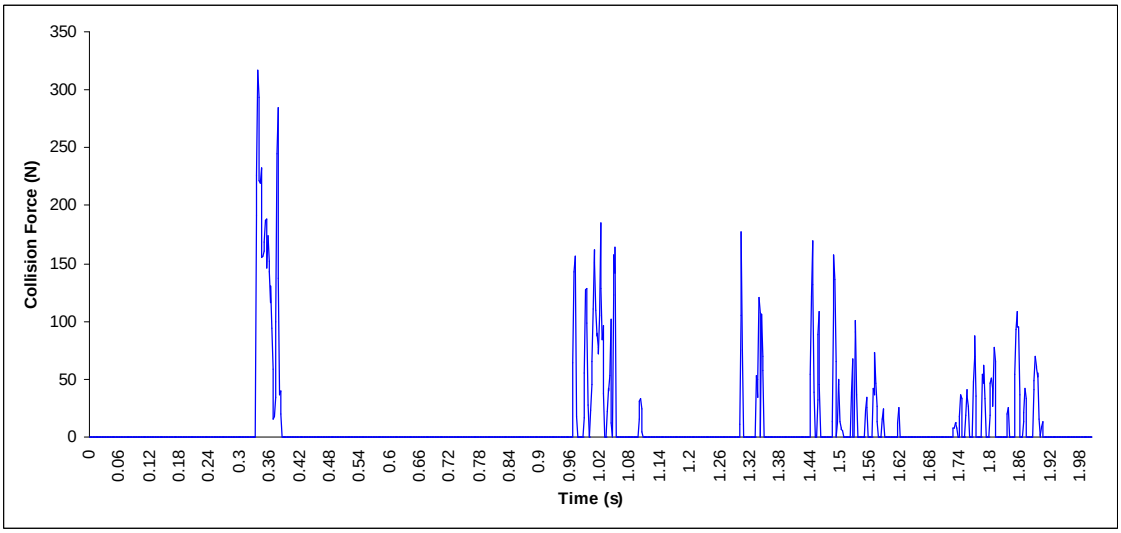


Figure 4.21: Scene with implicit surface - Collision force evolution

In figure 4.22 the graph presents the variation of the minimum y-coordinate of the box and the maximum y-coordinate of the ground during the simulation. This coordinate remains constant in the ground object because it is fixed. However, the y-coordinate of the box object varies according to gravitational force that acts in it. In this graph we can verify that when a collision occurs the y-coordinate of the box briefly crosses y-coordinate of the ground. This represents the amount of penetration between the objects and is used to calculate collision force.

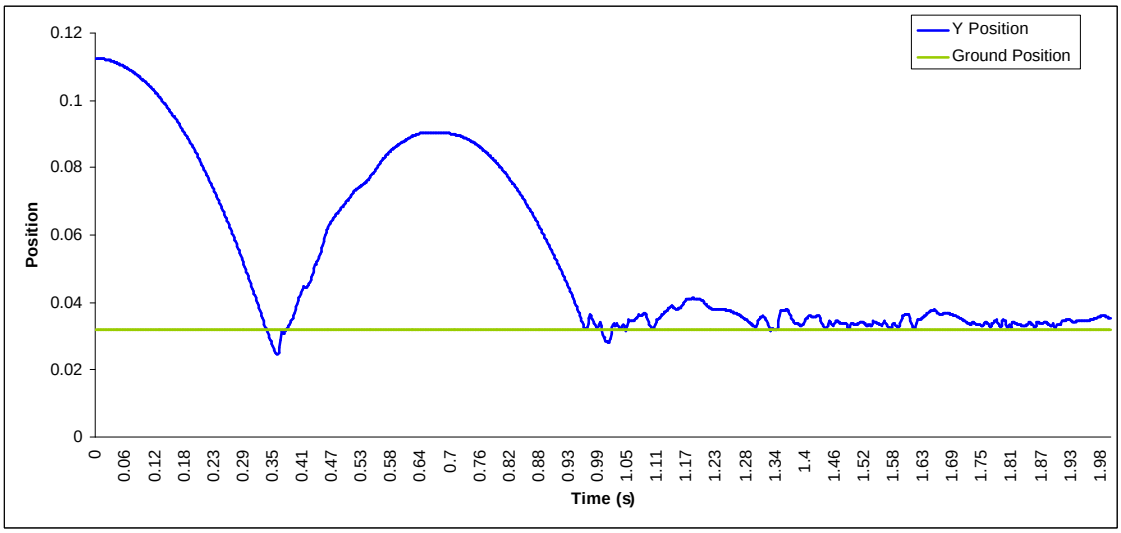


Figure 4.22: Scene with implicit surface - y-coordinate position

As in the experiment presented in section 4.3.1, we modify also the density and elasticity of the box object and plot the results in the graphs below.

- *Object's Density Modifying*

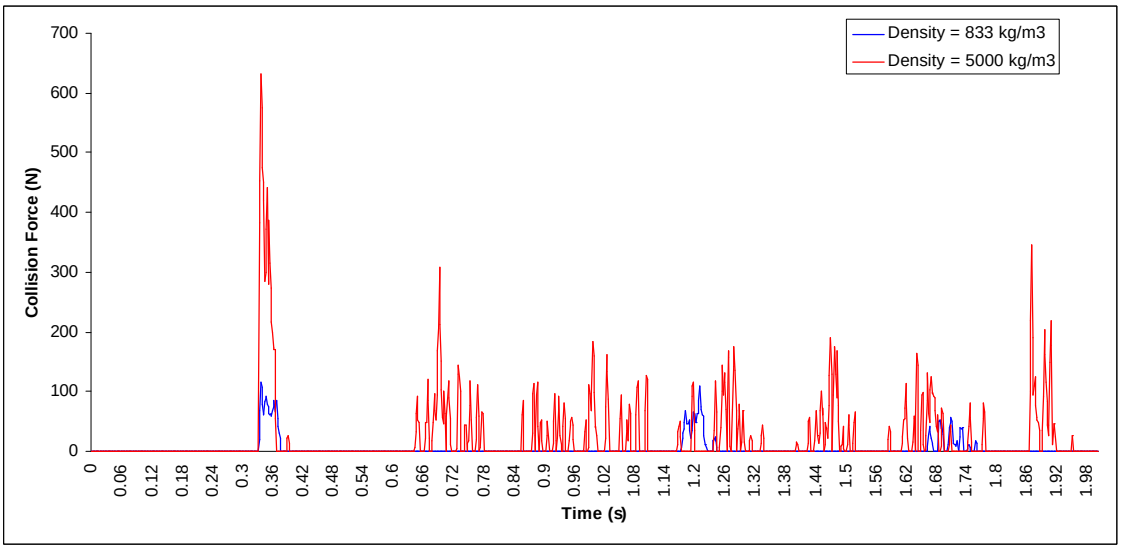


Figure 4.23: Collision force evolution varying the density of the box object (mesh collision)

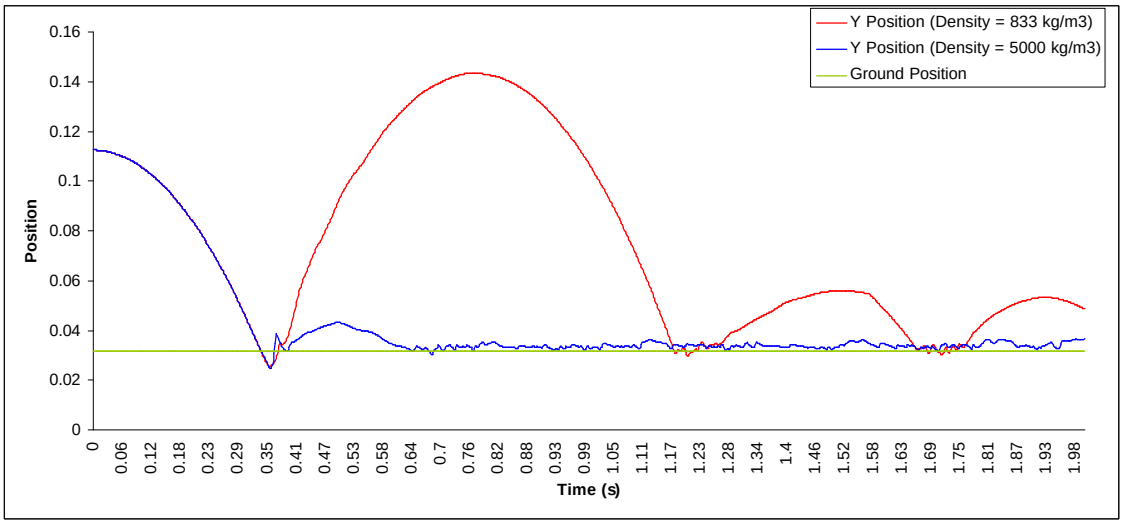


Figure 4.24: Evolution of the y-coordinate position varying the elasticity of the box object (mesh collision)

- *Object's Elasticity Modifying*

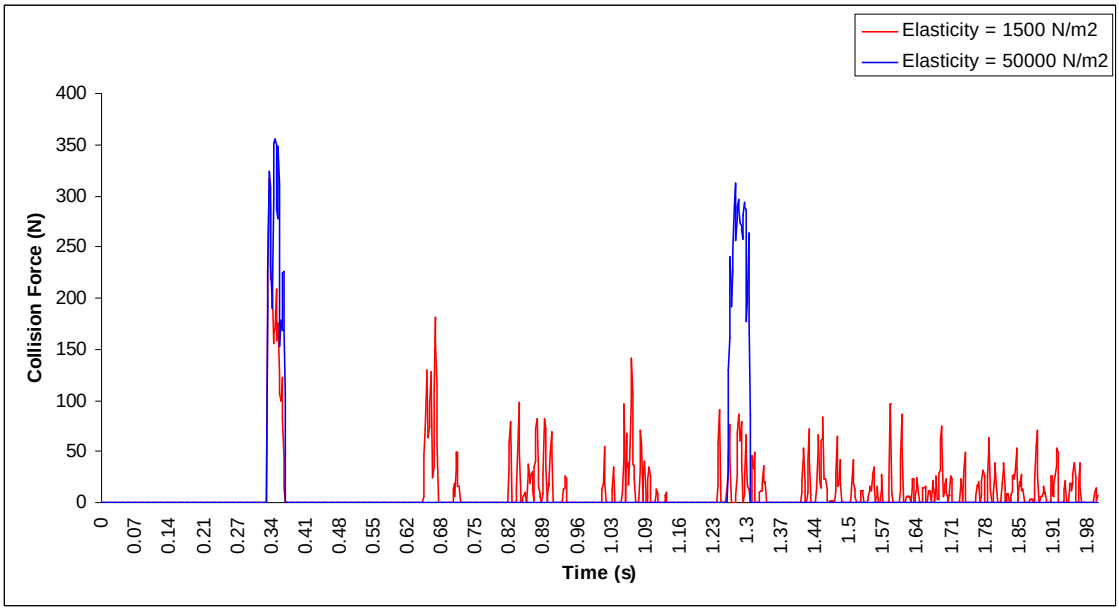


Figure 4.25: Collision force evolution varying the elasticity of the box object (mesh collision)

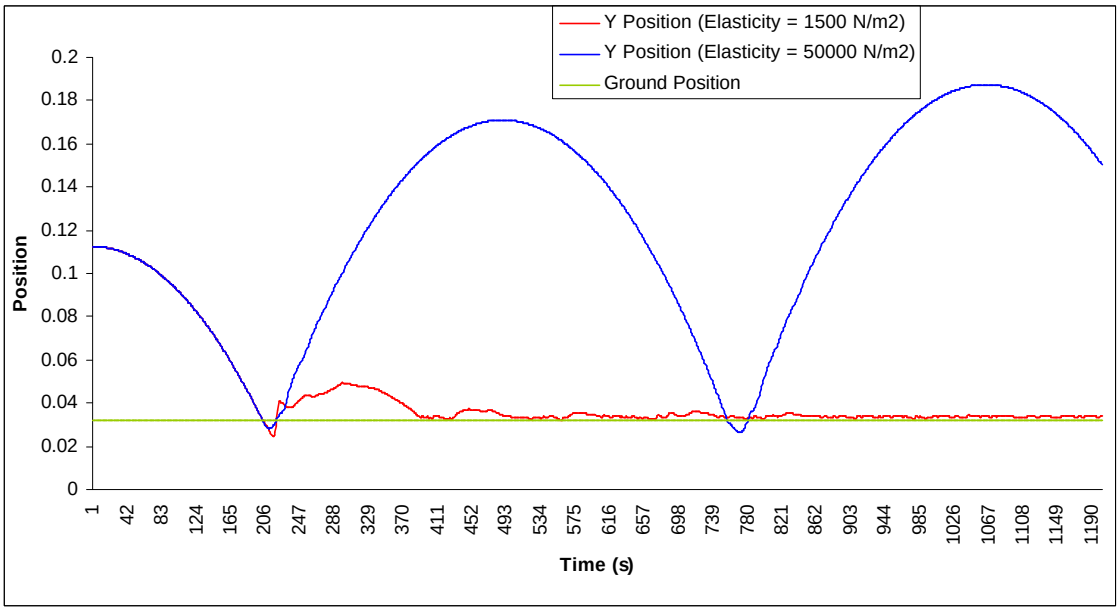


Figure 4.26: Evolution of the y-coordinate position varying the elasticity of the box object (mesh collision)

4.3.3 Performance

In this section we analyze the performance of our implementation. The scene (see Appendix E) is composed of 11 objects where 9 objects fall at the same time due to the gravitational force, 1 object is fixed and 1 object has the fixed board and the non-fixed center (where the gravitational force acts). The characteristics of each object are described in Figure 4.27, the simulation parameters in Table 4.1. The equipment used to perform these experiments was a PC Dual Intel Xeon 1.7Ghz, 1Gb Ram and a Wildcat III graphics card (300Mhz, 64Mb Ram). The application was developed using Microsoft Visual C++ version 6.0 on Windows 2000.

Table 4.1: Simulation Parameters

Parameters	Values
Iterations per second	1000
Simulation duration	5.0s
Refresh rate of the view	100 times per second of simulation
Surface generation	10 times per second of simulation
Collision detection	at each step of simulation

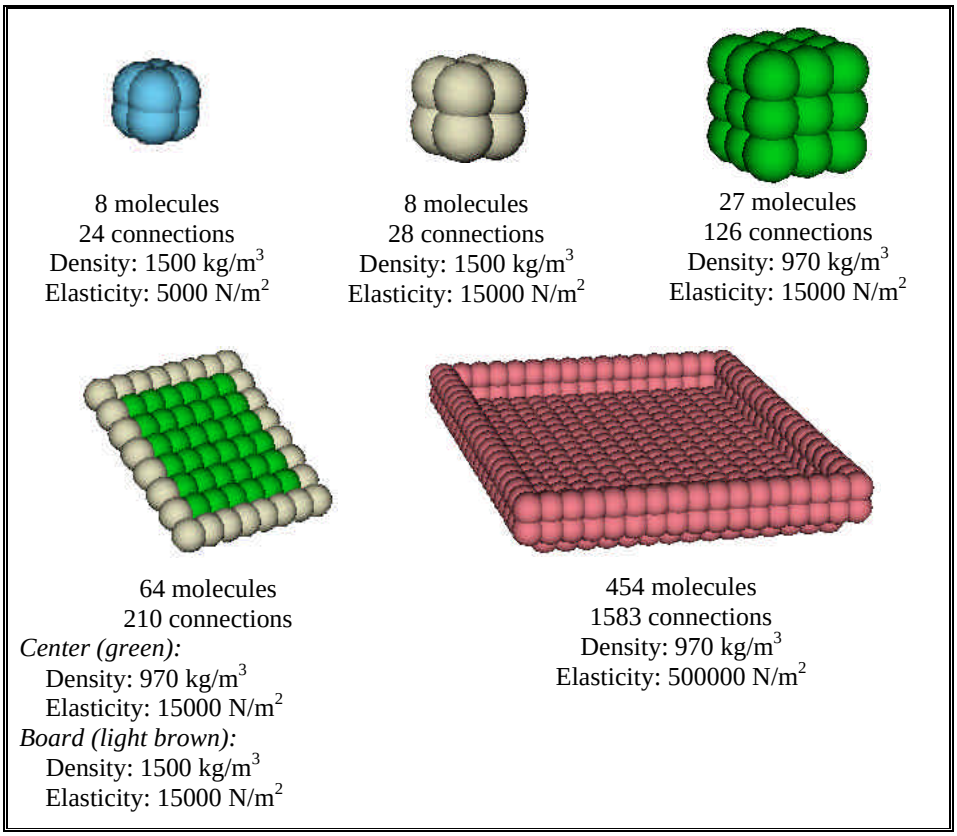


Figure 4.27: Performance test - objects of the scene

Table 4.2: Computation time

Computation time	Surface generation	Collision detection	Visualization	Total
Collision detection using Bounding Box and RAPID	521.441000s (8min 41s)	1303.260000s (21min 43s)	101.519000s (1min 41s)	1997.109000s (33min 28s)
Collision detection using V-Collide and RAPID	538.932000s (8min 58s)	1937.989000s (32min 29s)	103.483000s (1min 43s)	2657.531000s (44min 29s)
Spheres collision	0.000000s	29.622000s	346.002000s (5min 45s)	446.563000s (7min 44s)

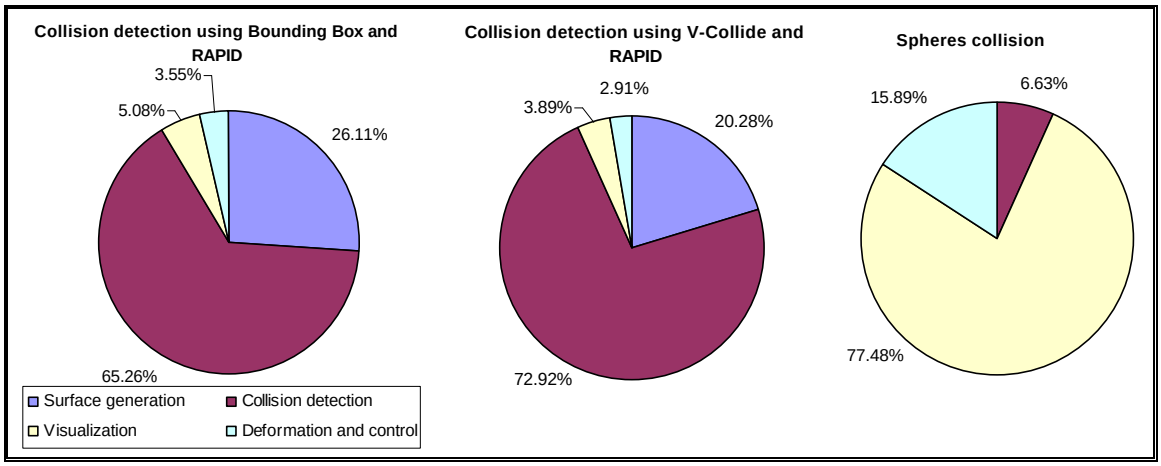


Figure 4.28: Performance

5 Conclusion

The goal of this work was to implement a collection of functions to compute response forces due to collision between simple deformable objects. These functions were implemented based in the penalty method and integrated into a deformation model being developed at VRLab.

Our implementation generates an implicit surface to cover the deformable objects, detect collisions between them, and provide the deformation engine with response forces derived from collisions.

From the performed experiments, we figured out that:

- Collision response forces have been appropriated when the displacements of the virtual objects are small. Nevertheless, for large displacements, the calculation of these forces demands very stiff springs to avoid unwanted interpenetrations, which can cause numerical instabilities. So, the choice of spring constant interacts with the choice of integration time step, and then influences the cost of the simulation.
  One possible solution is an extension to the spring-damper penalty method where relative velocity between the objects at the moment the collision is also considered on determining penalty springs stiffness.
- The method presented in this work to detect collisions using V-Collide is the major bottleneck of simulation. In our experiments, around 69% of the computation time was spent by this process. It confirms that V-Collide is not suitable to detect collisions between deformable objects. On the other hand, the sphere-to-sphere collision method is very fast, but the objects surfaces are not smooth which could cause contacts to be non-realistic. So, other approaches must be investigated in order to find one that better adapts to the type of objects we are dealing with.
- Implicit surface generation is also a considerable time-consuming process, mainly when number of spheres increase. Thus, an idea is to generate the surface only once at the beginning of the simulation, and deform it according to positions of the underlying spheres.

Bibliography

[Aubel 02] A. Aubel. *Anatomically-based human body deformations*. PhD Thesis. EPFL, Lausanne, 2002.

[Baraff 89] D. Baraff. Analytical Methods for Dynamic Simulation of Non-Penetrating Rigid Bodies. In *Computer Graphics*(Proc. SIGGRAPH), Vol. 23, No. 3, pp. 223-232. ACM, July 1989.

[Baraff 92] D. Baraff. Dynamic Simulation of Non-penetrating Flexible Bodies. In *Computer Graphics* (Proc. SIGGRAPH), Vol. 26, No. 2, pp. 303-308. July 1992.

[Baraff 93] D. Baraff. Non-penetrating Rigid Body Simulation. In *EUROGRAPHICS'93*. Barcelona, September 6-10, 1993.

[Baraff 94] D. Baraff. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. In *Computer Graphics* (Proc. SIGGRAPH), Annual Conference Series, pp. 23-34, 1994.

[Baraff 98] D. Baraff and A. Witkin. *Physically Based Modelling*. Siggraph '98 Course Notes. 1998.

[Barzel 92] R. Barzel. *Physically-based Modeling for Computer Graphics*. ACM Press, 1992. 334p.

[Bourke 97] P. Bourke. *Polygonising a scalar field.* 1997. Available at: http://astronomy.swin.edu.au/~pbourke/modelling/polygonise/

[COME 02] CO-ME. http://co-me.ch. 2002

[Cottle 92] R.W. Cottle, J.S. Pang and R.E. Stone. *The Linear Complementary Problem.* Boston: Academic Press, 1996.

[Desbrun 99] M. Desbrun, P. Schröder and A. Barr. Interactive Animation of Structured Deformable Objects. In *Graphics Interface*, pp. 1-8. 1999.

[Faure 96] F. Faure. An Energy-Based Method for Contact Force Computation. In *Proceedings of EUROGRAPHICS'96.* Computer Graphics Forum, Vol. 15, No. 3, pp. 357-366, 1996.

[Gamma 02] Gamma Group. *Collision Detection*. Department of Computer Science, University of North Carolina. 2002. Available at: http://www.cs.unc.edu/~geom/collide/index.shtml

[Gottschalk 96] S. Gottschalk, M. C. Lin and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *Computer Graphics* (Proc. SIGGRAPH), Annual Conference Series, Vol. 30, ACM SIGGRAPH, pp. 171-180. 1996

[Hahn 88] J. K. Hahn. Realistic Animation of Rigid Bodies. In *Computer Graphics* (Proc. SIGGRAPH), Vol. 22, No. 4, pp 299-308. August 1988.

[Hudson 97] T.C. Hudson et al. V-Collide: Accelerated Collision Detection for VRML. In *Proceedings of Second Symposium on the Virtual Reality Modeling Language (VRML'97)*, pp. 119-125. New York: ACM Press, 1997

[Jansson 00] J. Jansson and J.S.M Vergeest. A General Mechanics Model for Systems of Deformable Solids. In *Proceedings International Symposium On Tools and Methods for Competitive Engineering (TMCE 2000)*, pp. 361-372. Delft: Delft University Press, 2000.

[Joukhadar 98] A. Joukhadar, A. Deguet and C. Laugier. A collision model for deformable bodies. In *IEEE International Conference Robotics and Automation*, Vol. 2, pp. 982-988. 1998.

[Kitware 01] Kitware, Inc. *The visualization ToolKit*. 2001 Available at: http://public.kitware.com/VTK/

[McKenna 90] M. McKenna and D. Zeltzer. Dynamic simulation of autonomous legged locomotion. In *Computer Graphics* (Proc. SIGGRAPH), Vol. 24, pp. 29-38. ACM, August 1990.

[Mirtich 94] B. Mirtich and J. Canny. Impulse-based Dynamic Simulation. In *Proc. of Workshop on Algorithmic Foundations of Robotics*. February 1994.

[Mirtich 95] B. Mirtich and J. Canny. Impulse-based Simulation of Rigid Bodies. In *Proc. of 1995 Symposium on Interactive 3D Graphics*, pp. 181-188. April 1995.

[Mirtich 95a] B. Mirtich. Hybrid Simulation: Combining Constraints and Impulses. In *Proc. of First Workshop on Simulation and Interaction in Virtual Environments*, July 1995.

[Mirtich 96] B. Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD Thesis, University of California, Berkeley, December 1996.

[Moore 88] M. Moore and J. Wilhelms. Collision Detection and Response for Computer Animation. In *Computer Graphics*, Vol. 22, No. 4, pp. 289-298. August 1988.

[Nave 00] C. R. Nave. *Hyperphysics.* Department of Physics and Astronomy, Georgia State University, Atlanta, Georgia, 2000. Available at: http://hyperphysics.phy-astr.gsu.edu/hbase/hframe.html.

[Pang 96] J.S. Pang and J.C. Trinkle. Complementarity Formulations and Existence of Solutions of Dynamic Multi-Rigid-Body Contact Problems with Coulomb Friction. *Mathematical Programming.* 73: pp. 199-226. 1996.

[Physics 01] *The Physics Classroom*, 2001. Available at: http://www.glenbrook.k12.il.us/gbssci/phys/Class/newtlaws/newtltoc.html

[Platt 88] J.C. Platt and A. H. Barr. Constraint Methods for Flexible Models. In *Computer Graphics* (Proc. SIGGRAPH), Vol. 22, No. 4, pp. 279-288. ACM, August 1988.

[Popovic 00] J. Popovic et al. Interactive Manipulation of Rigid Body Simulations. In *Computer Graphics* (Proc. SIGGRAPH), Annual Conference Series, pp.209-217. ACM SIGGRAPH, 2000.

[Tallec 94] P. Le Tallec. Numerical Methods for Nonlinear Three-dimensional Elasticity. In P.G. Ciarlet and J.L. Lions. *Handbook of Numerical Analysis*. Elsevier Science B.V., Vol. III, chapter VI, pp. 565-578. 1994.

[Terzopoulos 87] D.Terzopoulos et al. Elastically deformable models. In *Computer Graphics* (Proc. SIGGRAPH), Vol. 21, pp. 205-214. ACM, August 1987.

[Trolltech 01] Trolltech. *On-line reference documentation*. 2001. Available at:

http://doc.trolltech.com/2.3/index.html

[Turk 02] G. Turk and J. F. O'Brien. Modelling with Implicit Surfaces that Interpolate. In *ACM Transactions on Graphics*, Vol. 1, No. 4, pp. 855-873. ACM Press, October 2002.

[Witkin 87] A. Witkin, K. Fleischer and A. Barr. Energy Constraints on Parametrized Models. In *Computer Graphics* (Proc. SIGGRAPH), Vol. 21, No. 4, pp. 225-232. ACM, August 1987.

[Young 96] H.D. Young and R.A. Freedman. *University Physics*. Addison-Wesley Publishing Company, Inc. 1996.

[Zhang 96] P. Zhang. *Physically Realistic Simulation of Rigid Bodies*. Master thesis, Department of Computer Science, Tulane University. 1996. Available at: http://www.eecs.tulane.edu/www/Zhang/

Appendix A: Newton’s Law of Motion

The effects of forces on objects are described by Newton's three laws. A force may be defined as any influence, which tends to change the motion of an object, that is, a force is that which causes an object to accelerate.

- *Newton’s First Law*

  An object moving with constant velocity will continue to move with constant velocity until acted upon by a net, external force.

Newton's First Law states that an object will remain at rest or in uniform motion in a straight line unless acted upon by an external force. It may be seen as a statement about inertia, that objects will remain in their state of motion unless a force acts to change the motion [Young 96].

- *Newton’s Second Law*

  If a net external force acts on a body, the body accelerates. The direction of acceleration is the same as the direction of the net force. The net force vector is equal to the mass of the body times the acceleration of the body.

$$\sum \vec{F} = m\vec{a}$$

The Newton’s second law is used to describe the causes of motion along with Newton’s first law and Newton’s third law. Besides this law has certain limitations, but is extremely useful for the solution of standard problems inducing the effects of friction [Nave 00].

- *Newton’s Third Law*

  If a body A exerts a force on body B (an “action”), then body B exerts a force on body A (a “reaction”). These two forces have the same magnitude but are opposite in direction. These two forces act on different bodies.

  In symbols,

$$\vec{F}_{A\,on\,B} = -\vec{F}_{B\,on\,A}$$

For example, when you kick a soccer ball, the forward force that your foot exerts on the ball launches it into its trajectory, but you also feel the force that ball exerts on your foot. These action and reaction forces are contact forces that are present only when

the two bodies are touching. But it is important remember this law also applies to force that do not require physical contact.

Appendix B: V-Collide User's Manual[13]

Introduction

The V-Collide collision detection library performs efficient and exact collision detection between triangulated polygonal models. It uses a 2-level hierarchical approach: the top level eliminates from consideration pairs of objects that are not close to each other, while the bottom level performs exact collision detection down to the level of the triangles themselves.

The basic steps involved in using this library are creating objects, adding sets of triangles to these objects, choosing which pairs of objects should be tested for collisions, setting the positions of the objects, performing the collision test, and getting back reports of the test results. Based on these results and any other parameters of the simulation/interaction, the objects may be moved and the collisions tested again, etc.

V-Collide is written in C++, but it provides a C interface as well.

Table B.1: C++ Command reference - #include <VCollide.h>

int VCollide::NewObject (int *id);	Create a new object and prepare it for adding triangles.
int VCollide::AddTri (double v1[3], double v2[3], double v3[3]);	Add triangles to the current object (only valid between NewObject() and EndObject()).
int VCollide::EndObject (void)	Finish adding triangles to the current object and build the hierarchical collision detection structures for the object.
int VCollide::DeleteObject (int id);	Delete an object.
int VCollide::ActivateObject (int id);	Turn on collision detection for an object.
int VCollide::DeactivateObject (int id);	Turn off collision detection for an object.
int VCollide::ActivatePair (int id1, int id2);	Turn on collision detection between a specific pair of objects.
int VCollide::DeactivatePair (int id1, int id2);	Turn off collision detection between a specific pair of objects.
int VCollide::UpdateTrans (int id, double trans[4][4]);	Update the transformation applied to an object.Note that we consider only the change in position of the object. Scaling is not supported.
int VCollide::Collide (void);	Compute collisions.
int VCollide::Report (int size, VCReportType *vcrep);	Does not return VC_OK or VC_ERR, but rather the number of collisions that occurred. If size is nonzero, copies up to size collision reports into the array colrep.

[13] Extraided and adaptaded from Hudson's documentation about V-Collide: A. Pattekar; J. Cohen; T. Hudson; S. Gottschalk; M. Lin; D. Manocha. **V-COLLIDE USER'S MANUAL - Release 1.1.** Department of Computer Science, University of North Carolina.

Table B.2: C command reference - #include <VCol.h>

void * vcOpen (void);	Creates and returns a valid handle to a new collision detection engine.
void vcClose (void *vc_handle);	Shuts down a collision detection engine.
int vcNewObject (void *vc_handle, int *id);	Create a new object and prepare it for adding triangles.
int vcAddTri (void *vc_handle, double v1[3], double v2[3], double v3[3]);	Add triangles to the current object (only valid between begin_object() and end_object()).
int vcEndObject (void *vc_handle)	Finish adding triangles to the current object and build the hierarchical collision detection structures for the object.
int vcDeleteObject (void *vc_handle, int id);	Delete an object.
int vcActivateObject (void *vc_handle, int id);	Turn on collision detection for an object.
int vcDeactivateObect (void *vc_handle, int id);	Turn off collision detection for an object.
int vcActivatePair (void *vc_handle, int id1, int id2);	Turn on collision detection between a specific pair of objects.
int vcDeactivatePair (void *vc_handle, int id1, int id2);	Turn off collision detection between a specific pair of objects.
int vcUpdateTrans (void *vc_handle, int id, double trans[4][4]);	Update the transformation applied to an object.Note that we consider only the change in position of the object. Scaling is not supported.
int vcCollide (void *vc_handle);	Compute collisions.
int vcReport (void *vc_handle, int size, col_report_type * colrep);	Does not return VC_OK or VC_ERR, but rather the number of collisions that occurred. If size is nonzero, copies up to size collision reports into the array colrep.

Creating objects

To create an object, first call NewObject(), which will set up an empty object and provide an integer ID for that object.

For every triangle in the object, call AddTri() with the coordinates of its three vertices. If the object has faces with more than three vertices, you will first need to triangulate them.

When you are done adding triangles, call EndObject(). This tells the library to build its data structures for the object.

The only library call which may appear between NewObject() and EndObject() is AddTri(). Objects can be deleted with DeleteObject().

You can create or delete objects at any time during the simulation, but building the data structures for these objects has some run-time overhead.

Managing activation state

V-Collide's activation state determines which objects will be tested for collisions. The activation state has two components -- a per-object component and a pairwise component (for each possible pair of objects). These two components are managed independently, so for a pair of objects to be tested for collision, not only must the pair be active, but each of the two objects must be active as well.

When an object is created, it is active by default. In addition, all pairs of objects that include this new object are also active.

The pairwise component of the activation state is managed using the ActivatePair() and DeactivatePair() calls, while the per-object component is managed using the ActivateObject() and DeactivateObject() calls. Because these components are managed independently, modifying an object activation state does not affect the activation state of an object's pairs.

Moving objects

To move an object, call UpdateTrans(). This function takes as arguments the ID of the object and a new transformation for it, expressed as a 4x4 matrix, formulated to be multiplied to the left of a column vector during the transformation. This matrix should be a rigid-body transformation -- rotation and translation. When an object is first created, it has the identity matrix as its transformation.

Performing the collision test

When all the objects' transformations have been modified as necessary and the activation state is properly set, call collide() to perform the collision testing for the current time step.

Getting reports

Calling Report() will return the number of collisions that occurred in the most recent test. Report() can also be passed a buffer and a buffer size limit, and the buffer will be filled with collision reports (of type VCReportType, defined in VCollide.[H]).

Return values

Most operations called on a V-Collide engine return an integer success code: VC_OK on success, or an appropriate error code on failure. These codes are defined in VCollide.[h]. Return values should be checked regularly to make sure that the application is performing correctly.

Data files

V-Collide has no native data file format, so you are free to use to format of your choice. The example code reads from several different file types.

Appendix C: RAPID User's Manual[14]

Performing a Collision Query

A model thus specified can have a placement in its environment, or in "world space". The model's placement in world space is defined as the placement of the model's coordinate axes within world space, which are specified as a rotation, *R*, followed by a translation, *T*. Given the placement of a model with *R* and *T*, we can determine the location in world space of a vertex of the model, given the vertex's coordinates in model space:

$$x_w = Rx_m + T$$

where x_m is a point in the model coordinate system, and x_w are the coordinates of the same point, but with respect to the world coordinate system.

The basic function of RAPID is to indicate whether two objects are in contact in world space. Suppose model m_1 has orientation R_1 and position T_1 in world space, while model m_2 has orientation R_2 and position T_2. Then the function call to rapid which asks whether the two models are touching is,

```
int
Collide (double R1[3][3], double T1[3][3]. RAPID_model *m1,
         double R2[3][3], double T2[3][3], RAPID_model *m2,
         int flag);.
```

This function returns RAPID_OK, which is 0, on success. A nonzero value indicates that the call failed, and the value is itself the error code. At present, the only error RAPID_Collide() can return is RAPID_ ERR_ COLLIDE_ OUT_ OF_ MEMORY.

After calling this function, the number of pairwise intersecting triangles can be found in the global variable RAPID_ num_ contacts. So if this variable is 0, the models were not touching. If it nonzero, they were touching.

To find out which triangles were among the contact pairs, the client must look in the global array RAPID_contacts[] and the first RAPID_num_contacts elements are valid data. This is an array of contact pair structures:

```
struct collision_pair{
        int id1;
        int id2;
};

struct collision_pair *RAPID_contact;
```

[14] Extraided and adaptaded from Gottschalk's documentation about RAPID: S. Gottschalk. **RAPID User Manual Version 2.0.** Department of Computer Science, University of North Carolina, Chapel Hill, N.C. 4/25/97.

Each contact pair structure corresponds to a unique pair of overlapping triangles, indicated by the id1 and id2. So, for example, after calling Collide() as shown above, if find that RAPID_num_contacts has been set to 43 then we could look at, say, RAPID_contact[20].id2 to retrieve from the 21'st contact pair the triangle from model *m2*

The global variables remain valid until Collide() is called again.

Note that a given triangle id may appear multiple times in the contact pair list once for each triangle it touches in the opposing model. But a given contact pair will appear only once in the list.

Theoretically it is possible for each of *n* triangles in one model to touch each of *m* triangles in another model, result in a list containing *m*n* contact pairs. If *m* and *n* are in the hundred thousands or millions then the list could be quite long. If RAPID is unable to allocate space for the contact list (or any other required structure) then Collide() will return RAPID_ ERR_ COLLIDE_ OUT_ OF_ MEMORY instead of RAPID_ OK.

Building a model

So, how did RAPID acquire the models in the first place, and how do the triangles get their id numbers?

The client tells RAPID the shape of an object by allocating a RAPID_model object, and adding the model's triangles to it. The following sequence of calls creates a pyramid model, consisting of six triangles. Notice that the square base of the pyramid must be built as two triangles.

```
static double p0[3] =  { 0.0,  0.0, 1.0 };    //top of pyramid
static double p1[3] =  {-0.5, -0.5, 0.0 };    //SW corner
static double p2[3] =  { 0.5, -0.5, 0.0 };    //SE corner
static double p3[3] =  { 0.5,  0.5, 0.0 };    //NE corner
static double p4[3] =  {-0.5,  0.5, 0.0 };   //NW corner

RAPID_model *m =  new RAPID_model;
m->BeginModel();
m->AddTri(p1, p2, p0, 0 );  //south face
m->AddTri(p2, p3, p0, 1 );  //east face
m->AddTri(p3, p4, p0, 2 );  //north face
m->AddTri(p4, p1, p0, 3 );  //west face
m->AddTri(p1, p4, p2, 4 );  //bottom face
m->AddTri(p2, p4, p3, 5 );  //bottom face
m->EndModel();
```

Notice that each triangle is given an id number, as we add it to RAPID's object. When RAPID reports contacts, these are the id numbers that get put into the contact_pair structures.

The BeginModel() tells RAPID to prepare the object *m* for the addition of triangles. Each subsequent AddTri() adds a triangle to the object *m* stores a copy of the triangle in *m*. When EndModel() is called, RAPID knows you won't be adding any more triangles, and it then performs any necessary preprocessing.

Any of these three procedures may attempt to allocate additional space for the model. If the allocation fails, then the procedure will return RAPID_ ERR_ COLLIDE_ OUT_ OF_ MEMORY. Otherwise, a successful procedure call will return RAPID_OK.

Note: It is acceptable to overlap the BeginModel() and EndModel() pairs for different objects as in the example below:

```
RAPID_model *m1 = new RAPID_model;
RAPID_model *m2 = new RAPID_model;
m1->BeginModel();
m1->AddTri(…);
m2->BeginModel();
m2->AddTri(…);
m1->AddTri(…);
m1->EndModel();
m2->AddTri(…);
m2->EndModel();
```

The RAPID model object can be destroyed with the usual C++ syntax,

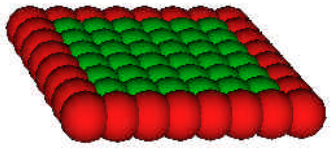
```
delete m1;
delete m2;
```

To reload a model into a RAPID_model object, the BeginModel() call can be invoked again, which frees all associated storage.

Once EndModel() completes, the model is said to have been processed. Collide() returns RAPID_ERR_UNPROCESSED_MODEL when passed unprocessed models.

The usual build sequence for RAPID_model objects is BeginModel() repeated AddTri() followed by one EndModel() and this entire sequence can be repeated. However, any sequence of calls is safe - the system will not be corrupted. Calling EndModel() when no triangles have been added yet will leave.

Appendix D: Example of a XML File



```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE come SYSTEM "comescene.dtd">

<come>
  <scene gX="0.0" gY="-1.4" gZ="0.0">
    <simulation fps="600" duration="30.0"/>
    <patient name="John" gender="M" weight="70" height="1.80"
      age="25">
    <materials>
      <material description="bord"   R="1.0" G="0.1" B="0.1"
        damping_const="0.2" density="970.0"  youngs_modulus="15000.0" medium_density="1.0"/>
      <material description="center" R="0.0" G="0.8" B="0.1" damping_const="0.2" density="970.0"
              youngs_modulus="15000.0" medium_density="1.0"/>
     <material description="object" R="0.1" G="0.1" B="0.8" damping_const="0.2" density="833.0"
             youngs_modulus="15000.0" medium_density="1.0"/>
    </materials>
    <molecule_organ type="cartilage" description= "floor">
       <molecule x="-0.0525" y="-0.05" z="-0.0525" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0525" y="-0.05" z="-0.0375" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0525" y="-0.05" z="-0.0225" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0525" y="-0.05" z="-0.0075" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0525" y="-0.05" z="0.0075" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0525" y="-0.05" z="0.0225" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0525" y="-0.05" z="0.0375" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0525" y="-0.05" z="0.0525" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0375" y="-0.05" z="-0.0525" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0375" y="-0.05" z="-0.0375" material="center" friction_const="0.02"
              radius="0.012" />
      <molecule x="-0.0375" y="-0.05" z="-0.0225" material="center" friction_const="0.02"
              radius="0.012" />
      <molecule x="-0.0375" y="-0.05" z="-0.0075" material="center" friction_const="0.02"
              radius="0.012" />
      <molecule x="-0.0375" y="-0.05" z="0.0075" material="center" friction_const="0.02"
              radius="0.012" />
      <molecule x="-0.0375" y="-0.05" z="0.0225" material="center" friction_const="0.02"
              radius="0.012" />
      <molecule x="-0.0375" y="-0.05" z="0.0375" material="center" friction_const="0.02"
              radius="0.012" />
      <molecule x="-0.0375" y="-0.05" z="0.0525" material="bord" friction_const="0.02"
              radius="0.012" fixed="true"/>
      <molecule x="-0.0225" y="-0.05" z="-0.0525" material="bord" friction_const="0.02"
```

radius="0.012" fixed="true"/>
<molecule x="-0.0225" y="-0.05" z="-0.0375" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0225" y="-0.05" z="-0.0225" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0225" y="-0.05" z="-0.0075" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0225" y="-0.05" z="0.0075" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0225" y="-0.05" z="0.0225" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0225" y="-0.05" z="0.0375" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0225" y="-0.05" z="0.0525" material="bord" friction_const="0.02"
radius="0.012" fixed="true"/>
<molecule x="-0.0075" y="-0.05" z="-0.0525" material="bord" friction_const="0.02"
radius="0.012" fixed="true"/>
<molecule x="-0.0075" y="-0.05" z="-0.0375" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0075" y="-0.05" z="-0.0225" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0075" y="-0.05" z="-0.0075" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0075" y="-0.05" z="0.0075" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0075" y="-0.05" z="0.0225" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0075" y="-0.05" z="0.0375" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="-0.0075" y="-0.05" z="0.0525" material="bord" friction_const="0.02"
radius="0.012" fixed="true"/>
<molecule x="0.0075" y="-0.05" z="-0.0525" material="bord" friction_const="0.02"
radius="0.012" fixed="true"/>
<molecule x="0.0075" y="-0.05" z="-0.0375" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="0.0075" y="-0.05" z="-0.0225" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="0.0075" y="-0.05" z="-0.0075" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="0.0075" y="-0.05" z="0.0075" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="0.0075" y="-0.05" z="0.0225" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="0.0075" y="-0.05" z="0.0375" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="0.0075" y="-0.05" z="0.0525" material="bord" friction_const="0.02"
radius="0.012" fixed="true"/>
<molecule x="0.0225" y="-0.05" z="-0.0525" material="bord" friction_const="0.02"
radius="0.012" fixed="true"/>
<molecule x="0.0225" y="-0.05" z="-0.0375" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="0.0225" y="-0.05" z="-0.0225" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="0.0225" y="-0.05" z="-0.0075" material="center" friction_const="0.02"
radius="0.012" />
<molecule x="0.0225" y="-0.05" z="0.0075" material="center" friction_const="0.02"

```
                radius="0.012" />
        <molecule x="0.0225" y="-0.05" z="0.0225" material="center" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0225" y="-0.05" z="0.0375" material="center" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0225" y="-0.05" z="0.0525" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0375" y="-0.05" z="-0.0525" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0375" y="-0.05" z="-0.0375" material="center" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0375" y="-0.05" z="-0.0225" material="center" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0375" y="-0.05" z="-0.0075" material="center" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0375" y="-0.05" z="0.0075" material="center" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0375" y="-0.05" z="0.0225" material="center" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0375" y="-0.05" z="0.0375" material="center" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0375" y="-0.05" z="0.0525" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0525" y="-0.05" z="-0.0525" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0525" y="-0.05" z="-0.0375" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0525" y="-0.05" z="-0.0225" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0525" y="-0.05" z="-0.0075" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0525" y="-0.05" z="0.0075" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0525" y="-0.05" z="0.0225" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0525" y="-0.05" z="0.0375" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
        <molecule x="0.0525" y="-0.05" z="0.0525" material="bord" friction_const="0.02"
                radius="0.012" fixed="true"/>
 </molecule_organ>
<molecule_organ type="cartilage" description= "cube">
        <molecule x="-0.0075" y="0.03" z="-0.0075" material="object" friction_const="0.02"
                radius="0.012" />
        <molecule x="-0.0075" y="0.03" z="0.0075" material="object" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0075" y="0.03" z="-0.0075" material="object" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0075" y="0.03" z="0.0075" material="object" friction_const="0.02"
                radius="0.012" />
        <molecule x="-0.0075" y="0.045" z="-0.0075" material="object" friction_const="0.02"
                radius="0.012" />
        <molecule x="-0.0075" y="0.045" z="0.0075" material="object" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0075" y="0.045" z="-0.0075" material="object" friction_const="0.02"
                radius="0.012" />
        <molecule x="0.0075" y="0.045" z="0.0075" material="object" friction_const="0.02"
```

```
                    radius="0.012" >
                    <force x="0.0" y="0.0" z="0.0" time="0.0"/>
                    <force x="0.0" y="0.0" z="0.0" time="6.0"/>
                    <force x="0.0" y="0.07" z="0.0" time="6.5"/>
                    <force x="0.0" y="0.07" z="0.0" time="7.0"/>
                    <force x="0.0" y="0.0" z="0.0" time="7.2"/>
                    <force x="0.0" y="0.0" z="0.0" time="10.0"/>
          </molecule>
        </molecule_organ>
      </patient>
    </scene>
</come>
```

Appendix E: Scene of the performance test

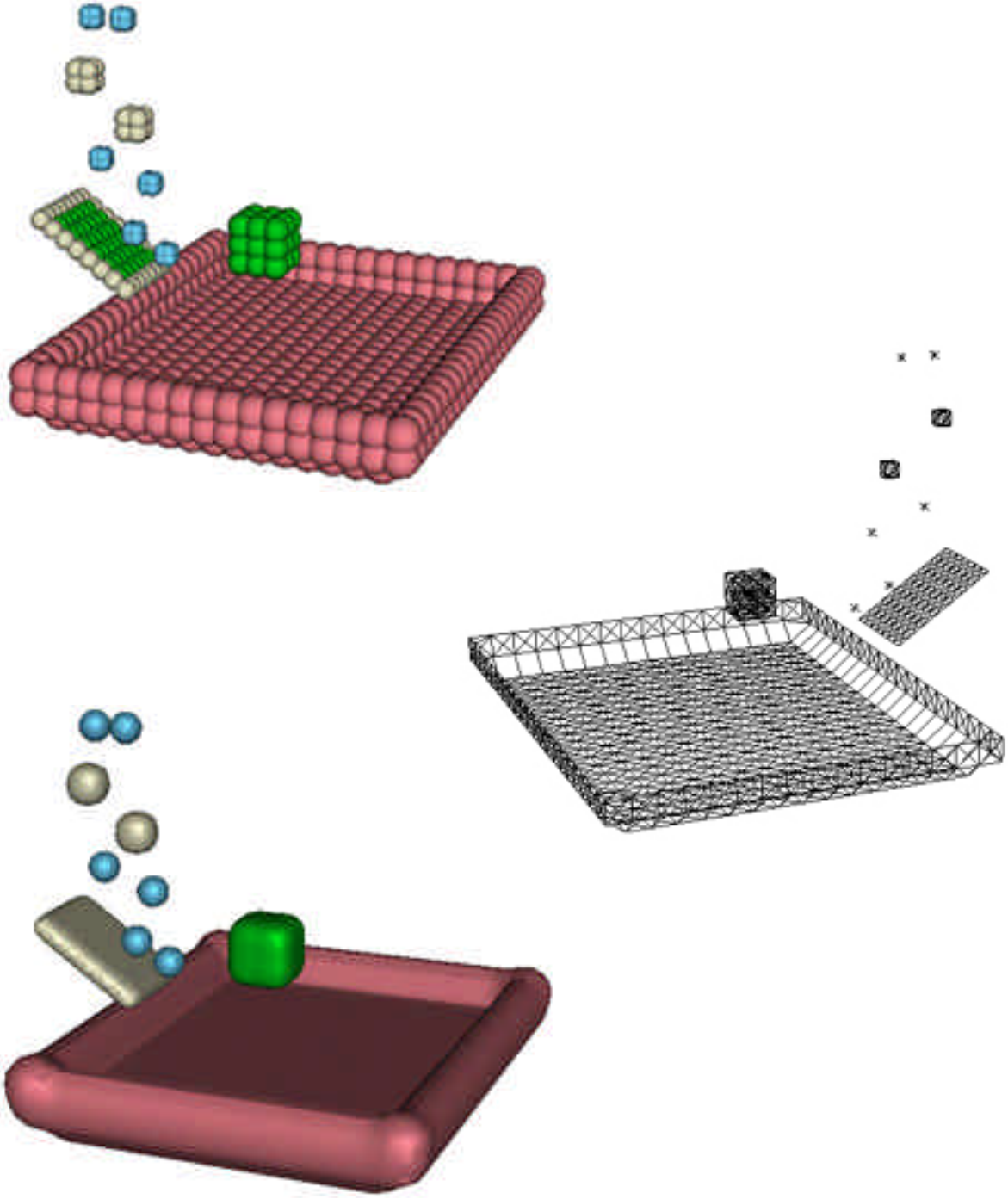


Figure E.1: Scene of the performance test