Real-Time Physically-Based Deformable Objects Simulation

Thesis · September 2013

DOI: 10.13140/2.1.2047.5847

1 author:

Lazaro Lesmes

1 PUBLICATION **0** CITATIONS

SEE PROFILE

Centro de Investigación en Matemáticas A.C.

Real-Time Physically-Based Deformable Objects Simulation

Tesis

para obtener el grado de

Maestro en Ciencias con Especialidad en
Computación y Matemáticas Industriales

Presenta:

Lazaro Emilio Lesmes León

Director de Tesis:

Dra. Claudia E. Esteves Jaramillo

September 04, 2013 Guanajuato, Gto., México

Abstract

This thesis presents an approach to real-time physically-based simulation of deformable objects. The work described here, is focused on deformation of volumetric solids represented as tetrahedral meshes. A damper mass-spring system has been implemented using OpenGL API and GLSL. The implementation developed reaches speed of interactive rates solving numerical simulations with huge meshes of hundreds of thousands elements due to the exploitation of latest features presented in current GPUs. Several numerical integration schemes are exposed and implemented. The use of some specific techniques to the mathematical model allowing numerically stable simulation over complex geometric meshes using explicit methods is described.

To illustrate the behaviour and performance, a graphics computer application was developed using modern graphics hardware. Results and computational time of various sample meshes are shown.

Resumen

En esta tesis se presenta un enfoque a la simulación en tiempo real de objetos deformables basada en leyes de la física. El trabajo realizado se centra en la deformación de sólidos volumétricos representados como mallas de tetraedros. Se ha implementado un sistema masa-resorte amortiguado usando OpenGL y GLSL. La aplicación desarrollada alcanza velocidades que permiten resolver simulaciones numéricas con grandes mallas de cientos de miles de elementos en tiempo interactivo, gracias a la explotación de las funcionalidades que brindan GPUs actuales. Se exponen varios esquemas de integración numérica y el uso de algunas técnicas específicas para alcanzar simulaciones numéricamente estables usando métodos explícitos sobre mallas geometricamente complejas.

Para mostrar los resultados y el rendimiento de la investigación realizada, se desarrolló una aplicación gráfica utilizando hardware gráfico moderno. Se exponen las tablas de los tiempos de cómputo y resultados obtenidos para varias mallas.

To my family

Contents

List of Figures

1 Introduction

Physically-based deformable object simulation has been an active research topic in the computer graphics (CG) community for almost three decades now. The main goal is usually to get eye-believable animations. To reach this goal, many physically-based models for deformable objects simulation have emerged. Numerous application have been developed for modeling objects composed of real materials: e.g. solids deformation and fracture, ropes, clothes, muscles, fluid and gases, melting objects, granular materials, etc. Also, recent applications demand deformable objects simulation in real-time, e.g virtual reality environments, video games, clinical surgery simulators, etc. This demand can be fulfilled because the current technology brings up the possibility to implement algorithms with huge computational complexity, and get the results in milliseconds. Therefore, the real-time simulation of physically-based deformable objects is a modern research topic.

1.1 Motivation

As summer project of 2012, I implemented a motion planning algorithm to move a deformable robot through an environment of rigid obstacles based on RRT[1] ([LaValle and Kuffner Jr. 2001]). In this project, the robot is a geometrically-based deformable object which is enforced by a FFD[2] ([Sederberg and Parry 1986]): object vertices position are computed interpolating established control points position. Then, looking for more realistic results, we decided to try physically-based deformable objects simulation. For our final application, a very efficient implementation

[1] Rapidly-exploring Random Tree, http://msl.cs.uiuc.edu/rrt/ .
[2] Free Form Deformation.

is required because it will be executed inside a sampling-based motion planner(which can imply that the simulation has to run a few thousand time). It is for this reason that we chose a model based on a *mass-spring system* for physically-based deformable objects, which is extensively used in CG because of its low computational cost.

1.2 Contribution

The real-time simulation of volumetric deformable solids represented as tetrahedral meshes is the main concern of this work. For achieving this, an implementation of a *mass-spring system* with volume preserving behaviour, employing GLSL[3], is proposed. A new algorithm was designed, a detailed description of its implementation, is given as well as the related qualities of OpenGL (labelled as a graphic library) for general-purpose computing on GPU (GPGPU). A novel GPU-based[4] solution for physically-based deformable object simulation, employing the latest OpenGL API features, was developed; and it is described in this work. There is, to our knowledge only one other work ([Georgii et al. 2005]) that implements *mass-spring system*, with volume preserving tetrahedral meshes, using GLSL; so, this is a little explored approach.

1.3 Organization

The main thread of this document is the description of the solution here proposed for real-time deformable objects simulation; but all issues dealt to achieve it are also discussed. The model implemented in this work for deformable objects modeling appears in the technical bibliography as *mass-spring system* or as *particles system*[5]. Because of that, in the following sections the terms *point masses* and *particles* are uses interchangeably.

This thesis is divided in six chapters following the current introduction. *Chapter 2* gives an overview of many physically-based deformable object models that have been used in CG, with a deeper view on *mass-spring system*. Also, there is a section of previous works related to volumetric deformable solids simulation. The mathematical formulation behind the implemented model is described in *Chapter 3*; as well as the equations that drive the dynamics of the system, the numerical integration schemes considered and the algorithm designed are listed. Then, the computational tools employed and the implementation of the algorithm of *Chapter 3*, over specific hardware to achieve a real-time application, are the contents of *Chapter 4 and 5* respectively. Next, the results from the implementation are presented and discussed in *Chapter 6*. Finally, in the last chapter, the conclusions of this research are done and the considerations for future work are marked.

[3] OpenGL Shading Language.
[4] Graphics Processing Unit.
[5] Also used to fluids and gases simulation.

2

Physically-Based Deformable Objects

The simulation of physically-based deformable objects in CG has been frequently focused more on getting animations that are believable to the eyes of the observer, rather than on the accuracy and correctness of the underlying models. In some applications, such as video games or clinical surgeries simulators, this is further enhanced by the need of achieving user interactive rates, which makes the use of simplified physical models a requirement. Fortunately, the rapid growth of graphics hardware capabilities has allowed to increase the complexity of the underlying models while decreasing simulation times from several hours to a few milliseconds: Algorithms that were proposed in the literature a few decades ago, that were prohibitive for real-time applications can be implemented in modern graphics processors to be used at interactive rates.

The beginning of research in the area of deformable objects in CG is usually marked with Lasseter's discussion on *squashing and stretching* [Lasseter 1987] together with Terzopoulos *et al.* seminal paper on elastically deformable models [Terzopoulos et al. 1987]. After these works, many researchers have tackled the problem of physically-based deformable objects simulation in CG community. Many of these works have been summarized in the surveys of [Gibson and Mirtich 1997; Nealen et al. 2006]. In the first survey [Gibson and Mirtich 1997], the authors describe a few purely geometrically-based algorithms but focus mainly on physically-based approaches and their applications. Nearly a decade later, Nielsen *et al.* [Nealen et al. 2006] proposed a classification of physically-based models for object deformations in two main groups: Lagrangian and Eulerian methods.
The survey classifies some of the methods present in the literature into these groups:

- Lagrangian Mesh Based Methods
  - Continuum Mechanics Based Methods
    - Finite Element Method (e.g Figure 2.1a, [Georgii 2008])
    - Finite Differences Method (e.g Figure 2.1b,[Terzopoulos et al. 1987])
    - Finite Volume Method (e.g Figure 2.1c,[Teran et al. 2003])
  - Mass-Spring Systems (e.g Figure 2.1d,[Teschner et al. 2004])
- Lagrangian Mesh Free Methods
  - Loosely Coupled Particle Systems (e.g Figure 2.1e, [Bell et al. 2005])
  - Smoothed Particle Hydrodynamics (SPH) (e.g Figure 2.1f, [Müller et al. 2003])
  - Mesh Free Methods for the solution of PDEs (e.g Figure 2.1g, [Pauly et al. 2005])
- Reduced Deformation Models and Modal Analysis (e.g Figure 2.1h, [Barbič and James 2005])
- Eulerian and Semi-Lagrangian Methods
  - Methods for Fluids and Gases Simulation (e.g Figure 2.1i, [Feldman et al. 2005])
  - Methods for Melting Objects Simulation (e.g Figures 2.1j, 2.1k and 2.1l, [Carlson et al. 2002])

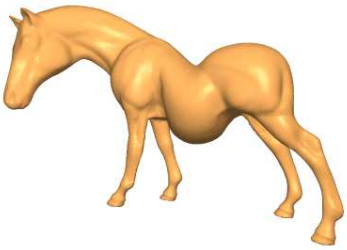
References to many interesting research about each one of previous categories and subcategories are given in [Nealen et al. 2006]. Special highlight is given to the works of [Barbič and James 2005] where samples of complex geometric objects under large deformations are shown at interactive rates and of in [Georgii 2008] where, based on the implicit finite element method with a multigrid approach, 200000 elements can be simulated at rate a rate of 10 times per second.

Additionally, in [Nealen et al. 2006] it is pointed out that since the survey of [Gibson and Mirtich 1997], significant contributions have been made in key areas of physically-based deformable models in CG, e.g. object modeling, fracture, cloth animation, stable fluid simulation, time integration strategies, discretization and numerical solution of partial differential equations (PDEs), modal analysis, space-time adaptivity, multiresolution modeling and real-time simulation.

2.1 Mass-Spring Systems

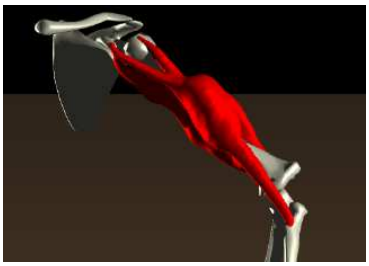
A detail review about mass-spring systems is presented in this section as it was the method chosen for modeling the deformable objects described and implemented in this thesis.

As its name implies, a mass-spring system consists in the representation of the target object as a set of point masses connected by massless springs in a specific lattice structure (see Figure 2.2). In [Erleben et al. 2005], a mass-spring system is defined as essentially a Particles System with a set of predefined springs between pairs of particles. The only function of springs is to produce internal forces to handle the behaviour of the deformable object. Neither collisions nor crossings between the springs are considered. Depending on the design of the network of springs

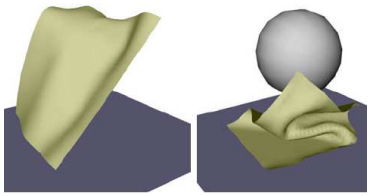


(a) Deformed tetrahedral horse.

(b) Persian carpet falling.



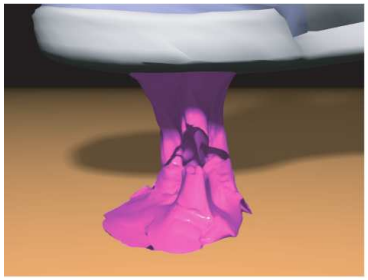
(c) Skeletal muscle contraction.



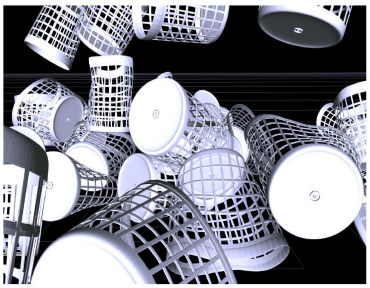
(d) Deformable cloth model.

(e) Flow through an hourglass.

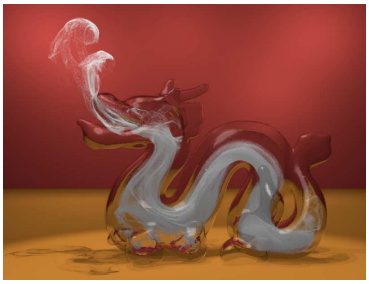
(f) Pouring water into a glass.



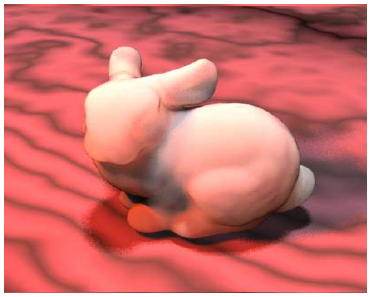
(g) Highly plastic deformations and ductile fracture.



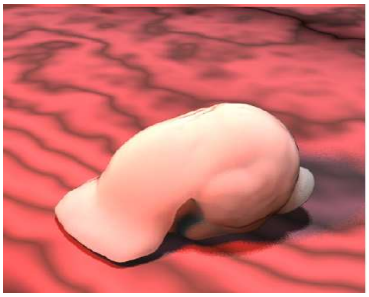
(h) Multibody dynamics with large deformations.



(i) Smoke on a tetrahedral mesh inside.



(j) Melting bunny sequence.



(k)

(l)

Figure 2.1: Physically-Based Deformable Models in CG.

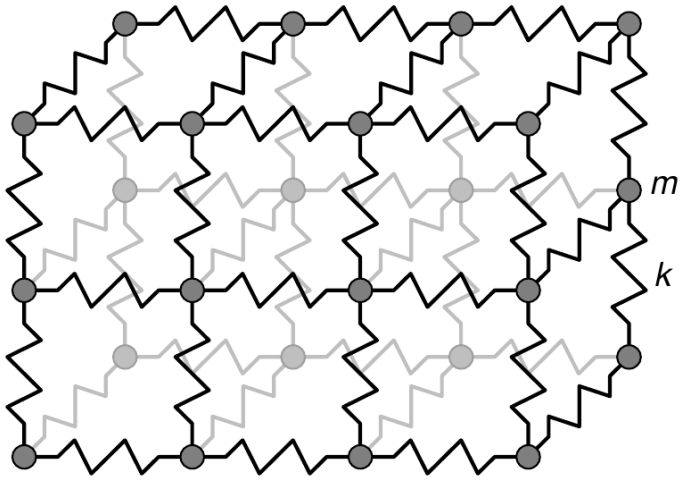


Figure 2.2: A portion of a mass-spring model. Springs connecting point masses exert forces on neighboring points when a mass is displaced from its rest positions [Gibson and Mirtich 1997]

.

that is formed by joining the point masses, objects with several properties can be modelled, e.g. objects with shearing and/or bending resistance, objects that preserve area and/or volume, etc. Some examples are described in Section 2.1.1.

A mass-spring system is therefore a dynamic system governed by some particular law of motion. In Section 3.1 Newton's and Lagrange's laws of motion, which have been used in the works discussed in Section 2.1.1, will be exposed in detail. The state of the system, analogous to *phase space* in [Baraff and Witkin 2003], at a given time t consists of the three dimensional vectors position x_i and velocity v_i of all point masses ($i = 1 \dots n$). Moreover, other relevant quantities for the motion of a particle are: *mass and force*. The force acting on each point mass is the composition of external forces such as gravity, friction, collision response, user interaction, etc. and internal forces due to the springs connection of neighbouring particles. In the mathematical model considered, theses forces are derived from the potential energy of the system, (see Section 3.3). In practical applications, any other quantities such as: color, shape, etc.; can be attached to each particle for rendering or visualization purposes.

2.1.1 Related Work

A lot of works for modeling deformable objects have exploited the mass-spring system, because it is intuitive, easy to implement and computationally cheaper than other models. For instance, a mass-spring system is computationally cheaper than a Finite Element System as the later has to be discretized and then assembled, whereas the former is already discrete.

A well-known introduction to physically-based modeling in CG, and in particular to the mass-spring modeling system, is that of [Baraff and Witkin 2003]. This is a short but excellent

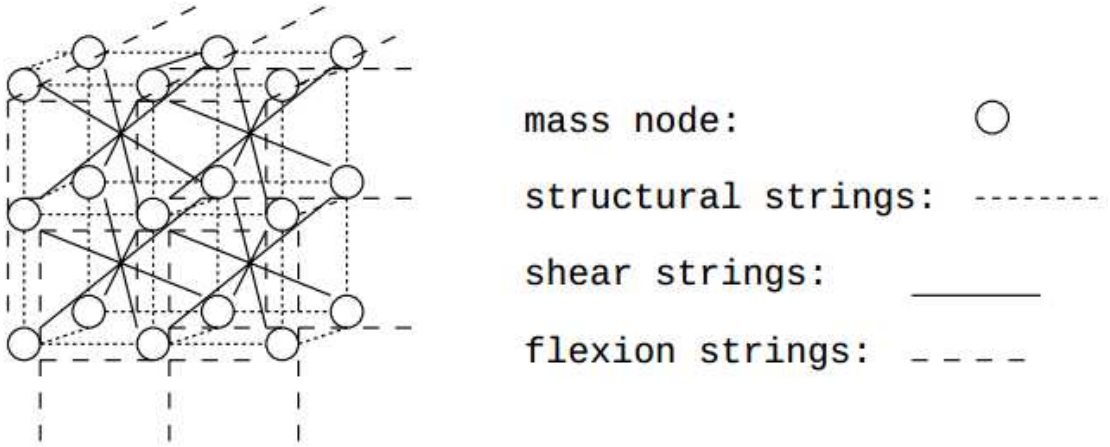


Figure 2.3: A Volumetric Mass-Spring Lattice [Chen et al. 1998]

course that covers this topic in great details from the beginning. Another, well-known reference is the book "Physics-based Animation" by Kenny Erbelen et al.[Erleben et al. 2005], very comprehensive for beginners and full of tips such as: pseudo codes and explicit formulas, really useful for implementation as well as a complete survey on related techniques and methods for physically-based modeling. A more recent work focused on the implementation of the mass-spring deformable model on GPU has been presented in [Vassilev and Rousev 2008] where an algorithm and several data structures are proposed taking advantage of parallel processors. Previously described works are helpful to understand and implement a general mass-spring system but do not focus on the problem of simulating volumetric deformable solids, which include plastic deformations, fractures, melting, etc. The remainder of this section is focused on elastic deformable objects as they are the main subject of study of our work.

In [Chen et al. 1998], the authors state that most of the previous work on mass-spring models were limited to 2D modeling or 3D rigid object modeling. The elastic model they propose, is a voxel-based mesh of $l \times m \times n$ point masses, regularly spaced in a lattice(Figure 2.3). Three different types of springs are defined linking specific point masses in order to achieve particular behaviors: (1) *Structural Springs,* that act in opposition to pure compression and stretching; (2) *Shear Springs,* that are constrained under shear stress and (3) *Flexion Springs,* that are resistant to bending. An animation sequence of a falling and bouncing chair using the model is shown.

Tzvetomir Vassilev and Bernhard Spanlang presented a mass-spring model for real-time simulation of volume-preserving deformable solids in [Vassilev and Spanlang 2002]. The model consist in surfaces of triangular meshes that conform volumetric objects, each vertex is a point mass and each edge a spring. The authors present a new type of spring called *"support spring",* with the intention of simulating matter inside objects and perform a volume-preserving behavior, saving the cost of explicit volume computations during the simulation. These springs connect

each point mass in the triangulated body surface with a virtual point in the body center. The main feature of support springs, is that their response depend not only on its own elongation but also, in a collective way, on the state of the whole system. The authors argue that their algorithm is linear with respect to the number of triangles of the deformable solid.

In [Matyka and Ollila 2003], Maciej Matyka and Mark Ollila simulated a pressure model of soft body based on simple laws of physics using a mass-spring system. Their model also consist in objects with surfaces of triangular meshes, these objects are virtually filled of an ideal gas. Implementing thermodynamics laws, the authors claim that were obtained high quality results in real-time. The additional step of their algorithm is the computation of pressure over all faces for calculating the corresponding force over vertices. To achieve a fast volume computation, required for the calculation of pressure, bounding objects of simple geometry, such as boxes, spheres and ellipsoids, are used.

A more general and robust mass-spring system was developed by Matthias Teschner et al. in [Teschner et al. 2004], this paper was the main reference for this thesis. Their model applies to both, volumetric objects of tetrahedral meshes and objects with surfaces of triangular meshes, and it considers elastic and plastic deformation. A system of potential energies is defined considering specific constraints such as distance between vertices, area of the triangles, volume of the tetrahedra, etc. These energies are minimal when the body is at its restring state. Then, the forces over a point mass are computed as the derivatives of the energies with respect to its position. Each potential energy type (one type of each kind of constraint) defined can influence a few of the neighboring particles, e.g. the energy of distance preservation affects the two particles at the spring extremes, the energy of area preservation affects the three particles in a triangle and the energy of volume preservation affects the four particles in a tetrahedron. Depending on the kind of object to be modeled, the effect of one potential energy compared with the effect induced by the other could be insignificant, e.g. when volumetric objects are modeled, area preservation energy is not considered, only energies of distance and volume preservation have considerable effects. Several geometrically complex meshes are tested and shown. A table with computation time of eight explicit numeric integration methods is analysed.

A method to accelerate computation of surgical simulation and modeling complex organs was presented by Jesper Mosegaard et al. in [Mosegaard et al. 2005]. It is pointed as the first proposal of a GPU-based mass-spring system by Marilena Maule et al. in [Maule et al. 2010]. Their model is represented as 3D grid built from medical datasets. All point masses are encoded in a 2D texture and neighbors information are reached through texture coordinates. Each vertex is always connected to a fixed amount neighbors, eighteen in this case, which can be a drawback in more general systems, e.g. with smaller valences. As results for this work, interesting sample of surgical simulation on a pig heart consisting of 42.745 particles in a regular grid reconstructed from a CT data set is shown, and the speed-up of GPU implementation over

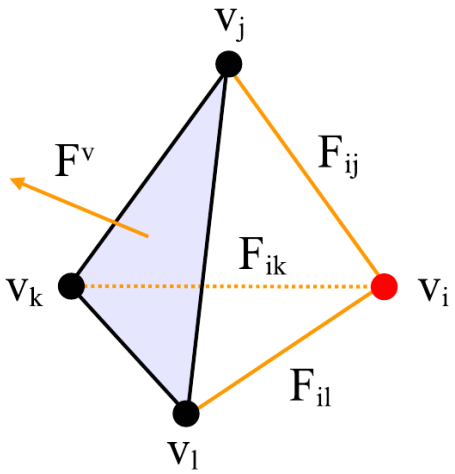


Figure 2.4: Illustration of volume preserving forces. The vertex v_i, forces are calculated with respect to the stretching of the springs ij, ik, il. The volume force F^v at vertex v_i in the direction of the opposite face normal ensures that each element resists changes of its initial volume. [Georgii et al. 2005]

CPU implementation with an increasing number of nodes is tabulated.

Georgii and R.Westermann analyse different implementations of mass-spring systems for interactive simulations of deformable surfaces on GPU in [Georgii and Westermann 2005]. Two different simulation algorithms implementing scattering[1] and gathering[2] operations on the GPU are compared with respect to performance and numerical accuracy. Some memory access patterns and GPU specific issues to be considered in mass-spring systems simulation are discussed. This work is extended in [Georgii et al. 2005] where volumetric deformable objects represented with tetrahedral meshes are simulated. The authors state that to further increase the physical realism of their simulation, volume preservation behavior was added. The forces on tetrahedral point masses derived from the volume-preservation action are linear with respect to the difference in volume and act in the direction of the normal vector of the face opposite to the respective vertex (see Figure 2.4). In their GPU implementation, particles are codified in a 2D texture (here called *vertex texture*). Then, the whole mesh is represented as a sequence of textures where each tetrahedron adjacent to its respective vertex in the texture is encoded in a 2D texture. To avoid memory overhead, a procedure to store vertexes with high valences in textures is described. A more comprehensive description and different algorithms about this research were published in Georgii's Ph.D. thesis [Georgii 2008].

A novel GPU-based implementation of physically-based deformation of tetrahedral meshes using a mass-spring approach was presented in [Tejada and Ertl 2005]. The main contribution of this work is the implementation of an implicit numerical solver on a GPU, but no volume-preservation forces where considered in their model. They affirm that fast and stable deformation

[1] Memory write operation from a computed address: e.g the C code $a[i] = x$ is a scatter operation.
[2] A gather operation is an indirect memory read operation, such as $x = a[i]$.

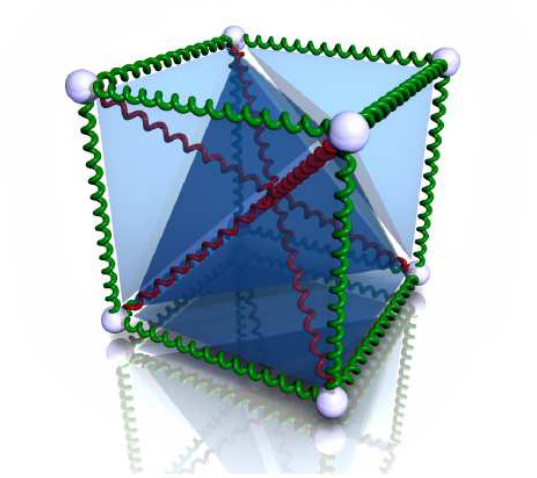


Figure 2.5: The volume constraint with its five tetrahedrons, eight particles, and 16 springs. The red springs can have different spring constants from the green ones.[Diziol et al. 2009]

of tetrahedral meshes with larger time-steps simulation were achieved. Performance comparisons of implicit and explicit numerical integration schemes on GPU are exposed. The mesh topology is represented with two 2D textures, only neighborhood information is stored, the vertex incidence on tetrahedral elements is not considered. Additional 2D textures are used to hold the vertex positions, velocities and external forces.

Min Hong et al. present a new method for fast volume preservation of a mass-spring system to achieve efficient deformable object simulation in [Min Hong and Welch 2006]. This method does not use a volume discretization and its authors argue that the simulated behavior is comparable to a finite-element method-based model at a fraction of the computational cost. This approach is independent from the geometric structure of an object.

Another method for volume preservation of surface models is that of [Arnab and Raja 2008]. The authors develop a surface mass-spring model with shape-preserving springs for deformable volume simulation. These springs act as surface springs but their length is zero. They work as virtual anchors from the interior of the object for preserving shape at equilibrium state. To compute volume displacement during simulation, a derivation of volume calculation employed in [Min Hong and Welch 2006] is implemented. An approach that estimates springs stiffness values from real material properties is proposed. A performance comparison of four different approaches that vary the stiffness setting and mass distribution is given.

Raphael Diziol et al. present a method for simulating volume conserving deformable bodies using an impulse-based approach in [Diziol et al. 2009]. Their model uses tetrahedral meshes where each tetrahedron is enforced with volume constraints in order to ensure total volume-preservation. From an arbitrary triangle mesh, a specific volume discretization is done, then each cell consisting of five tetrahedrons gets assigned a volume constraint. The volume constraint

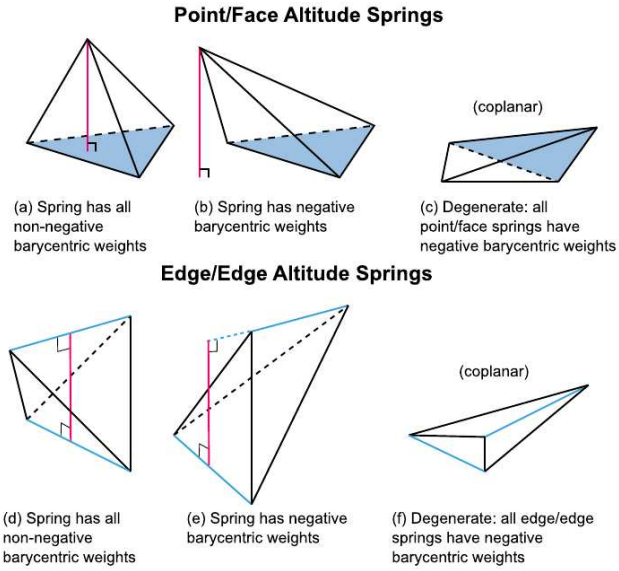


Figure 2.6: Varying examples of point/face and edge/edge altitude springs. Examples (a) and (d) represent ideal cases for the two types of altitude springs, whereas (b) and (e) show altitude springs of each type with negative barycentric weights. (c) shows all point/face altitude springs having negative weights, but there is still one edge/edge altitude spring that has non-negative weights. (f) shows all edge/edge altitude springs having negative weights, but one point/face altitude spring has non-negative weights. [Selle et al. 2008]

consists of a volume preserving tetrahedrons helped by springs which are introduced as external forces (see Figure 2.5). The authors argue that the impulse-based simulation enforces the constraints iteratively until all constraints are satisfied, and also that their algorithm is easy to implement and ensures exact volume conservation at each simulation step.

A mass-spring model for hair simulation is presented by Andrew Selle et al. in [Selle et al. 2008]. This work is similar to the others cited here because the hair full geometry is simulated using a tetrahedral mesh. To simulate volume preservation behavior, a new kind of spring is defined: *Altitude Spring* (see Figure 2.6). This is placed between each particle of the tetrahedron and a virtual node is projected onto the plane of the opposite face, or between two edges in a mutually orthogonal direction to the two lines containing the edges. The connection is used in certain cases of highly stretched or degenerate tetrahedron. Other kinds of springs are introduced to achieve more natural hair behavior, e.g. curly or straight hair.

2.2 Real-Time Applications in Computer Graphics

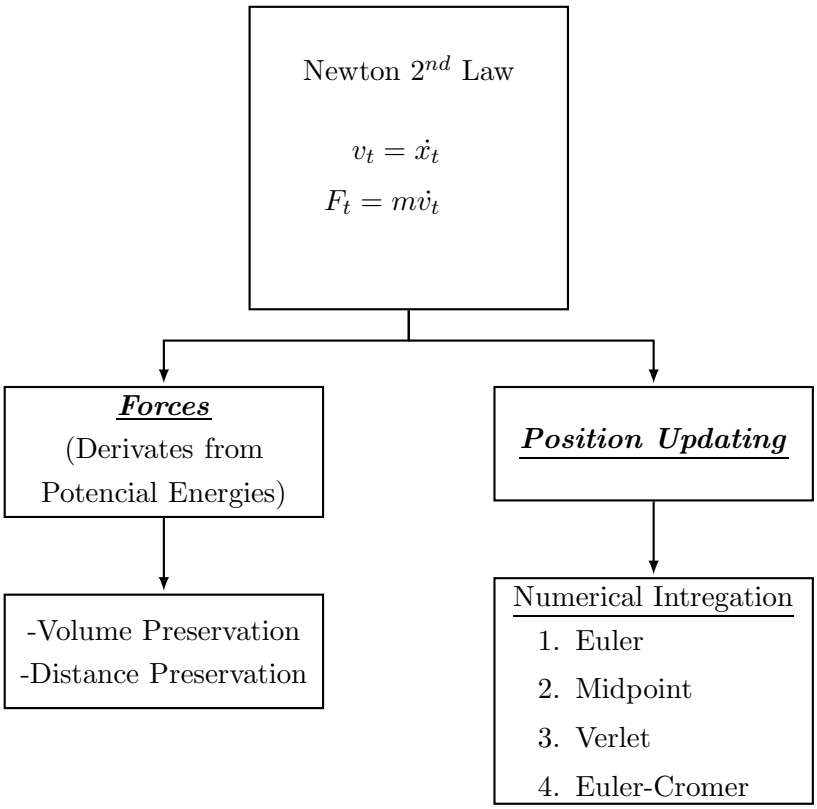
Real-Time Applications in Computer Graphics are about perception. Applications that display events that the user perceives they occur at speed they would in the real life, are eligible. According to [Akenine-Möller et al. 2008], an application displaying 15 fps (frames per second) is certainly real-time; the user focuses on action and reaction. From about 72 fps and up,

differences in the display rate are effectively undetectable. On the other hand, in the case of physics-based simulation, Teschner et al. indicate that an application runs in real time when the ratio of numerical integration time step and the computing time in compute one simulation step is one [Teschner et al. 2004]. They also propose this ratio as performance measure between numerical integration schemes. In this thesis results considering both criteria will be given.

3

Mathematical Model and Simulation

The overall approach of our approximation for modeling physically-based deformable objects is depicted in the following diagram:



3.1 Laws of Motion

The dynamic behavior of a mass-spring system is determined by a given law of motion. In all the references cited in *Related Work* Section 2.1.1 either the Newton's (Equation 3.2) or the Lagrange's (Equation 3.1) Laws of motion are used for modeling the particles movement:

$$m_i\ddot{x}_i + c\dot{x}_i + \sum F_i^{internal} = F_i^{external} \tag{3.1}$$

$$m_i\ddot{x}_i = \mathbf{F_i}, \tag{3.2}$$

where m_i is the mass of particle i, which is considered a constant scalar in this formulation; c is a damping constant; $F_i^{internal}$ is the resulting vector of all internal forces, $F_i^{external}$ is the resulting vector of all external forces and $\mathbf{F_i}$ is the resulting vector of all forces, over the point mass i (details in section 3.3); $\dot{x}_i$ and $\ddot{x}_i$ are the first and second derivatives of particle position with respect to time, which denote velocity and acceleration, respectively.

3.2 Potential Energies

To model deformation on objects, an energy-based system such as that is exposed in [Teschner et al. 2004; Erleben et al. 2005] is considered. In this thesis the formulation developed in [Teschner et al. 2004] is followed. Given constraints of the form $C(p_0, \ldots, p_{n-1})$, which are scalar functions depending on the positions of point masses p_i, an associated potential energy is defined as:

$$E(p_0, ..., p_{n-1}) = \frac{1}{2}kC^2, \tag{3.3}$$

with k denoting a stiffness coefficient that has to be defined for each type of potential energy. The overall potential energy derived from previous constraints is interpreted as a deformation energy of the object and it is zero if the object is not deformed, otherwise it is larger than zero.

The first type of potential energy described E_D, is associated with the constraint $C_D = \| p_j - p_i \| - D_0$ where p_i, p_j are the current positions of particles i, j respectively and D_0 $(with\ D_0 \neq 0)$ is the initial distance or resting distance between them:

$$E_D(p_i, p_j) = \frac{1}{2}k_D \left(\frac{\| p_j - p_i \| - D_0}{D_0} \right)^2. \tag{3.4}$$

This energy influences all pairs of particles connected by tetrahedral edges.

The second type of potential energy E_V represents the energy based on the difference of the

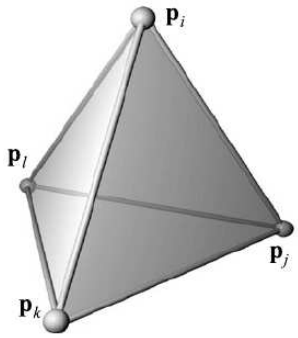


Figure 3.1: A tetrahedron with four mass points is the basic volumetric primitive of deformable model implemented. In this simple example, six distance-preserving forces between all pairs of points, e.g. $F_D(p_i, p_j)$, and the volume-preserving force $F_V(p_i, p_j, p_k, p_l)$ are considered.[Teschner et al. 2004]

current volume of a tetrahedron and its initial volume V_0 $(with\ V_0 \neq 0)$:

$$E_V(p_i, p_j, p_k, p_l) = \frac{k_V}{2}\left(\frac{\frac{1}{6}|(p_j - p_i)\cdot((p_k - p_i)\times(p_l - p_i))| - V_0}{V_0}\right)^2. \tag{3.5}$$

with p_i through l being the positions of the particles. The four particles that build a tetrahedron are influenced by this energy.

Another type of potential energy commonly considered in energy-based systems implementation is the potential energy based on the difference of areas between the current surface of the triangle and its initial area E_A. Teschner et al. pointed out that the influence of E_A is negligible compared with that of E_D and E_V for volumetric deformable solids simulation [Teschner et al. 2004]. For this reason E_A is not considered in this work.

3.3 Forces

Forces that act over each point mass in the system are classified as external or internal forces. On the one hand, external forces are those arise outside the system, e.g. gravitational force, air resistance force, etc. Each one has specific characteristics depending on its own nature, e.g. global earth gravity acts independently on each particle, it is equal to m_{ig}, where $\mathbf{g}$ is a constant vector in the direction of the center of the earth and with a magnitude equal to the gravitational constant (9.8 m/s^2 is a commonly used value). On the other hand, in the current formulation, internal forces are forces produce due to potential energies defined in Equation 3.3. Forces at particle p_i are computed as the derivatives of the energies with respect to the particle position:

$$F^i(p_0, ..., p_{n-1}) = -\frac{\partial}{\partial p_i}E = -kC\frac{\partial C}{\partial p_i}, \tag{3.6}$$

where k is the same value defined in Equation 3.3 and C are the constraints described above. Internal forces conserve linear and angular momentum of the object, as it is discussed in [Teschner et al. 2004].

Consequently, the forces derived from potential energy in Equation 3.4 that act over connected particles $x_i,\ x_j$, can be written as:

$$\begin{aligned} F_D(x_i) &= -k_D C_D \frac{\partial C_D}{\partial x_i} = -k_D(\parallel x_i - x_j \parallel - L)\frac{x_i - x_j}{\parallel x_i - x_j \parallel} \\ F_D(x_j) &= -k_D C_D \frac{\partial C_D}{\partial x_j} = k_D(\parallel x_i - x_j \parallel - L)\frac{x_i - x_j}{\parallel x_i - x_j \parallel}. \end{aligned} \tag{3.7}$$

Because of the behavior they cause to the deformable object, these forces are called *Distance Preservation Forces*; they coincide with the forces from Hooke's Spring Law in [Erleben et al. 2005].

Another kind of internal forces computed in our approach are the so called Volume Preservation Forces Fv, which can mimic bending forces. Given a tetrahedron composed of point masses $p_i,\ p_j,\ p_k,\ p_l$ (see Figure 3.1); edges defined as the vectors $e_1 = p_j - p_i$, $e_2 = p_k - p_i$, $e_3 = p_l - p_i$, and constraints from potential energy in Equation 3.5 written as $C_V(p_i, p_j, p_k, p_l) = \frac{1}{6}e_1(e_2 \times e_3) - V_0$, these forces over the four particles can be defined as:

$$\begin{aligned} F_V(x_i) &= k_V C_V(e_2 - e_1) \times (e_3 - e_1) \\ F_V(x_j) &= k_V C_V(e_3 \times e_2) \\ F_V(x_k) &= k_V C_V(e_1 \times e_3) \\ F_V(x_l) &= k_V C_V(e_2 \times e_1). \end{aligned} \tag{3.8}$$

Analogously to previous displayed forces, these forces are called *Volume Preservation Forces*; they can mimic bending forces. These forces are equivalent to those used to obtain the volume preservation behavior in [Georgii et al. 2005].

3.3.1 Damping Forces

"A dissipative force is one for which energy of the system decreases when motion takes places" [Eberly and Shoemake 2004]. In this sense, Damping Forces are dissipative forces that oppose the movement of particles in the system. They act like friction forces in the direction contrary to the velocity but they should not slow down the movement of the whole system. Equation 3.6 applying a damping effect can be written as:

$$F^i(p_0, ..., p_{n-1}; v_0, ..., v_{n-1}) = (-kC - k_d \sum_{0 \le j < n} \frac{\partial C}{\partial p_j} v_j)\frac{\partial C}{\partial p_i}, \tag{3.9}$$

with v_i denoting the velocity of a mass point and k_d denoting the damping coefficient. Then, the *distance preservation forces* from Equations 3.7 result the same as the Hooke's Law for damped springs in [Erleben et al. 2005]:

$$F_D(x_i) = -F_D(x_j) = -\left[k_D\left(\parallel x_i - x_j \parallel - L\right) + k_d \frac{(x_i - x_j)\cdot(v_i - v_j)}{\parallel x_i - x_j \parallel}\right] \frac{x_i - x_j}{\parallel x_i - x_j \parallel}, \tag{3.10}$$

where the damping parameter controls how the deformation energy of an object is reduced during dynamics simulations. Damping forces can improve the stability of the system because it is taken to its resting state. In [Teschner et al. 2004] the authors argue that they obtained stable simulations considering damping only within the *distance preservation forces*. Experiments done in this work showed that applying damping to *volume preservation forces* in addition to *distance preservation forces* can help for simulation of geometrically complex volumetric objects, such as those in the Figure 5.1. The implementation done for the above purpose do not follow exactly the Equation 3.9. An approach based on Equation 3.10 has been developed. The relative velocities between the six pairs of the four particles per tetrahedron projected over the *volume preservation forces* direction are considered:

$$F_V(x_i) = \left[k_V C_V - k_d \frac{(3v_i - v_j - v_k - v_l)\cdot((e_2 - e_1)\times(e_3 - e_1))}{\parallel (e_2 - e_1)\times(e_3 - e_1) \parallel}\right] \frac{(e_2 - e_1)\times(e_3 - e_1)}{\parallel (e_2 - e_1)\times(e_3 - e_1) \parallel}, \tag{3.11}$$

analogously, forces on particles j, k, l are computed. This formulation got better results(see results in Section 6) than the one in Equation 3.9 and than the one which considered only damping within *distance preservation forces* .

3.3.2 Deformation Limits

If a real spring is forced to an extremely large deformation it does not recover its original length. This shows that springs have deformation limits. In physically-based simulations, to neglected this limits can cause a problem of stiffness[1]. In order to achieve stable simulations a position-based strain limiting technique has been implemented. This is a geometric-based solution to simulate deformation limits. The procedure is to correct the positions of connected particles if the distance between them is greater than a fixed proportion from their initial separation:

$$\begin{aligned} & if \quad \parallel x_i - x_j \parallel \; > \; \alpha L \;\; then \\ \hat{x_i} \;&=\; x_i + \frac{m_j}{m_i + m_j}\left(\parallel x_i - x_j \parallel - \alpha L\right)\frac{x_i - x_j}{\parallel x_i - x_j \parallel} \\ \hat{x_j} \;&=\; x_j - \frac{m_i}{m_i + m_j}\left(\parallel x_i - x_j \parallel - \alpha L\right)\frac{x_i - x_j}{\parallel x_i - x_j \parallel}. \end{aligned} \tag{3.12}$$

[1]These type of problems arise when the differential equation associated to the problem have two or more functional terms that have widely different scales[Eberly and Shoemake 2004].

where L is the resting distance between the particles i, j. The position correction is done over the spring direction. The above formulation tries to include physical behavior moving more the particles with the smallest mass. The linear and angular momentum of the object is preserved because only internal forces are affected.

3.4 Mass Distribution

The physic properties of the particles that compose a volumetric object determine the object behavior during physically-based simulations. Therefore, in order to achieve realistic animation, the total mass of a virtual model has to replicate the total mass of the real object. A principle for homogenize volumetric solids is that, to bigger volume correspond greater mass; consequently, a mass distribution based of the model volume discretization has been implemented. Point masses that are common to many tetrahedron have more mass than others. The mass of a particle i is computed as:

$$m_i = \frac{\frac{\sum_{j \in i} v_j}{n_i}}{\sum_{t=0}^{T} \frac{\sum_{j \in i} v_j}{n_i}} M, \tag{3.13}$$

where $\frac{\sum_{j \in i} v_j}{n_i}$ is the average volume of all tetrahedra which have the particle i as node; the denominator of Equation 3.13 is the total sum of the average volume corresponding to each particle; M is the object mass. The above mass distribution allows stable simulations of deformable objects with irregular topology or non-uniform shape.

3.5 Numerical Integration

To simulate the dynamic behavior of the deformable model developed in this work, Newton's law of motion Equation 3.2 is computed for each particle. Newton's law of motion is an ordinary differential equation (ODE) of second order, but can be rewritten as a system of two coupled equations of first order:

$$\begin{aligned} v_t &= \dot{x_t} \\ F_t &= m\dot{v_t}, \end{aligned} \tag{3.14}$$

where F_t is the total force acting on a particle and x_t, v_t are functions that describe the motion of a particle, representing the particles position and velocity respectively. These are the quantities required for physically-based animation. The sub-index t_i denotes a discretization of the above functions with respect to time, e.g. x_{t0}, v_{t0} are the initial values. Given the initial values, Equations 3.14 belongs to a class of problems called *initial values problems*.

Several explicit numerical integration methods (EM) were implemented in this work to solve numerically the raised problem. Although, EM are in general only conditionally stable[2], they fit better than implicit ones for data-parallel computing[3] because no linear equations system have to be solved and therefore no data transfer is needed. Beside, parallelism is useful because a real-time application is desired, EM were chosen. Stable simulations are obtained combining some techniques described above (damping forces, strain limiting and mass distribution) with a appropriate time step (h) for the EM.

In this work four EM where implemented in order to compare their performance. These are (1)Euler,(2)Midpoint,(3)Verlet and (4)Euler-Cromer. The first method implemented is Euler's Method. The numerical integration step from t to $t+h$ for updating position and velocity, is written as:

$$\begin{aligned} x_{t+h} &= x_t + h\dot{x_t} &= x_t + hv_t \\ v_{t+h} &= v_t + h\dot{v_t} &= v_t + h\frac{F_t}{m}. \end{aligned} \tag{3.15}$$

The position update depends on velocity; the force update generally depends on the position and on the velocity. This method is not accurate ($O(h)$) and h must be close to zero to achieve numerical stability.

Another implemented method is the Midpoint Method or Second-Order Runge-Kutta. Position and Velocity updates are given as:

$$\begin{aligned} v_{t+\frac{h}{2}} &= v_t + \frac{h}{2}\frac{F_t}{m} \\ F_{t+\frac{h}{2}} &= f(x_t + \frac{h}{2}v_t, v_{t+\frac{h}{2}}) \\ x_{t+h} &= x_t + hv_{t+\frac{h}{2}} \\ v_{t+h} &= v_t + hF_{t+\frac{h}{2}}, \end{aligned} \tag{3.16}$$

This method has $O(h^2)$ accuracy but additional operations have to be done to compute $v_{t+\frac{h}{2}}$ and $F_{t+\frac{h}{2}}$.

The third one implemented method is Verlet Method. An important difference with the previous methods is that Verlet's method computes the new position value without any explicit velocity information. The position at time $t+h$ is calculated using position at time t, position at time $t-h$ and the total force at time t:

[2]EM requires small time steps to not increase the error in the result (Sometimes, are impractical time steps).
[3]A programming paradigm in which the same analysis code is applied simultaneously to different data elements.

$$\begin{aligned} x_{t+h} &= 2x_t - x_{t-h} + h^2\frac{F_t}{m} \\ v_{t+h} &= \frac{x_{t+h} - x_{t-h}}{2h}. \end{aligned} \tag{3.17}$$

The accuracy of position computation is $O(h^4)$ and for velocity is $O(h^2)$. This high accuracy allows to use a larger time step in comparison with other EM.

The last method implemented is the Euler-Cromer's Method, which is a semi-implicit method. The velocity is computed as in Euler's Method but for computing the new position the updated velocity information is used:

$$\begin{aligned} x_{t+h} &= x_t + hv_{t+h} \\ v_{t+h} &= v_t + h\frac{F_t}{m}. \end{aligned} \tag{3.18}$$

The obtained accuracy is as low as that of the Euler's Method, but the semi-implicit scheme gets more numerical stability.

Results and comparison between the above methods are given in Section 6. Also, specific considerations for their GPU implementation with OpenGL is detailed in Section 5.

3.6 Algorithm

A new algorithm for mass-spring system modeling was developed in this thesis. In general, this algorithm can be seen as a combination of the *element-centric*[4] and *edge-centric*[5] approach proposed in [Georgii 2008] for mass-spring system implementation on GPU. If only an *element-centric* approach is employed, the computation of *Distance Preservation Forces* is performed more than once over common edges to several tetrahedrons. This affects the accumulated force of point masses joined by theses edges. Moreover, applying only the *edge-centric* approach (only with the tetrahedra edges) volume preservation behaviour can not be achieved.

The CPU implementation of the Algorithm 3.6 is straightforward, but in the case of GPU implementation several data structures have to be designed and specific memory issues have to be considered. Section 5 gives a detailed explanation of these implementations. The complexity of the Algorithm 3.6 is $(O(v + e + t))$ where v is the number of vertices, e is the number of edges and t is the number of tethahedra of the object mesh. The pseudo-code listed is a general algorithm for mass-spring system simulation of volumetric deformable solids. However, some

[4]Every tetrahedron maintains references to its four points, spring stiffness and rest volume.
[5]Every edge maintains references to both incident points, spring stiffness and rest length.

Algorithm 3.1: Mass-spring system simulation

Input: A tetrahedral mesh of the object

repeat
  foreach *mass point i in mesh* **do**
    initialize total force accumulator F_i of vertex x_i.
  end
  foreach *tetrahedron t in mesh* **do**
    compute volume-preservation force over each mass point in t.(Equations on 3.8).
    add the computed force to each vertex total force accumulator.
  end
  foreach *edge e in mesh* **do**
    compute distance-preservation force over both mass point in e.(Equations on 3.7).
    add the computed force to both vertex total force accumulator.
  end
  foreach *mass point i in mesh* **do**
    update vertex position x_i, using any of EM described in Section 3.5.
  end
until *simulation ends*

modifications had to be done in specific cases, e.g. if the Midpoint Method is used for position update computation, another two cycles have to be added to compute the volume and distance preservation forces.

4

Computational Tools

With the aim of getting a real-time application, a data-parallel programming paradigm is employed to implement the algorithm proposed in this thesis for mass-spring system simulation. Some of the most recent features of graphics hardware have been exploited, a few of which will be described in this chapter.

4.1 GPU Architecture and Functionality

Early generations of graphics processors were intended only for rendering tasks such as: illumination, shading and texturing of triangles, etc. Special-purpose fixed-function engines (e.g OpenGL) were implemented in hardware making it impossible to use these chips in non rendering applications. Since then, the hardware of GPUs have had a huge developed[1].

As Kirk and Hwu pinpoint in their book [Kirk and Hwu 2010] (Figure 4.1), a modern GPU *is organized into an array of highly threaded streaming multiprocessors(SMs), two SMs form a building block. Also, each SM in Figure 4.1 has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with up to* 4 *gigabytes of graphics double data rate (GDDR) DRAM, referred to as global memory in Figure 4.1. These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold video images, and texture information for three-dimensional (3D) rendering, but for computing they*

[1] The GPUs computational power is growing 2-3 times faster than Moore's Law [Georgii 2008].

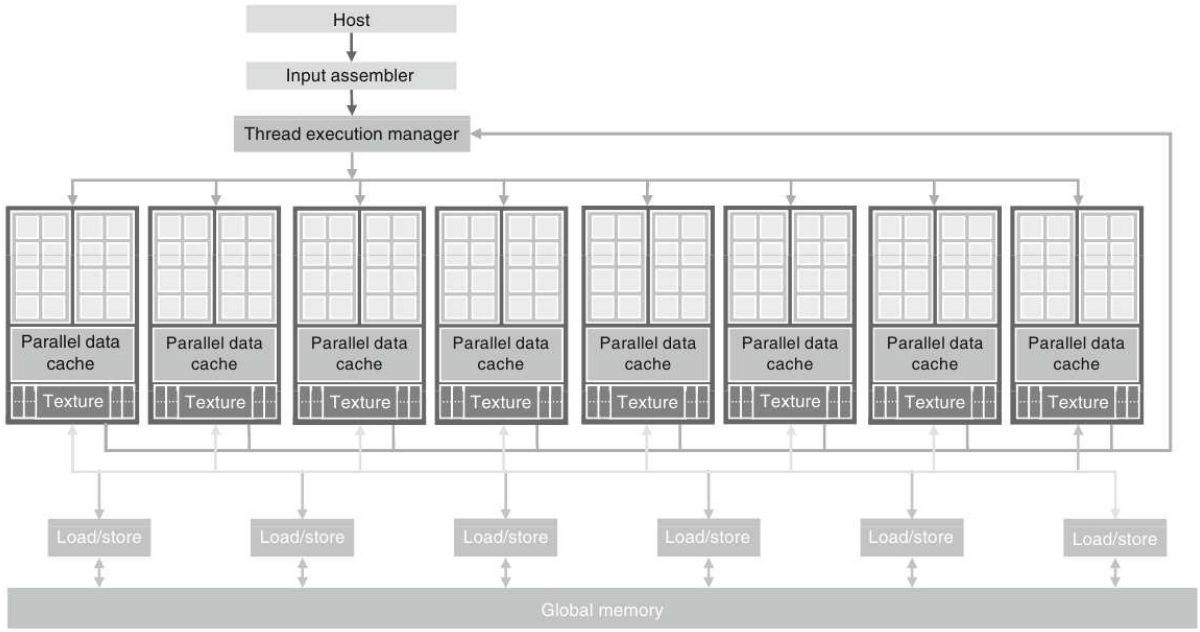


Figure 4.1: Architecture of a Modern GPU. [Kirk and Hwu 2010]

function as very-high-bandwidth.

The current design of GPU and its computational power has allowed its use for the implementation of general-purpose computing applications (labelled as GPGPU for the scientific community). Recently, several Application Programming Interfaces (APIs), with their own programming languages, have been created to develop GPGPU applications, e.g. CUDA and OpenCL. Also, APIs which were originally designed only for CG applications, such as Direct3D, OpenGL, etc., have included features to bring on GPGPU to their users, e.g. Compute Shaders (similar to Cuda or OpenCL kernels) introduction in OpenGL 4.3. Modern GPUs support both single and double precision floating point, and they outperform current CPUs with respect to memory bandwidth and floating point operations (see Figure 4.2).

4.2 OpenGL

The main computational tool used to implement the knowledge gained during this research is OpenGL[2], an API for accessing features in graphics hardware. OpenGL is widely used in the industry, from game to virtual reality and it is supported on a wide variety of platforms: from mobile phones to supercomputers.

In the latest edition of the official OpenGL reference book, also know as the Red Book, Shreiner et at. introduce OpenGL in the following way: *OpenGL is designed as a streamlined,*

[2]More information at http://www.opengl.org .

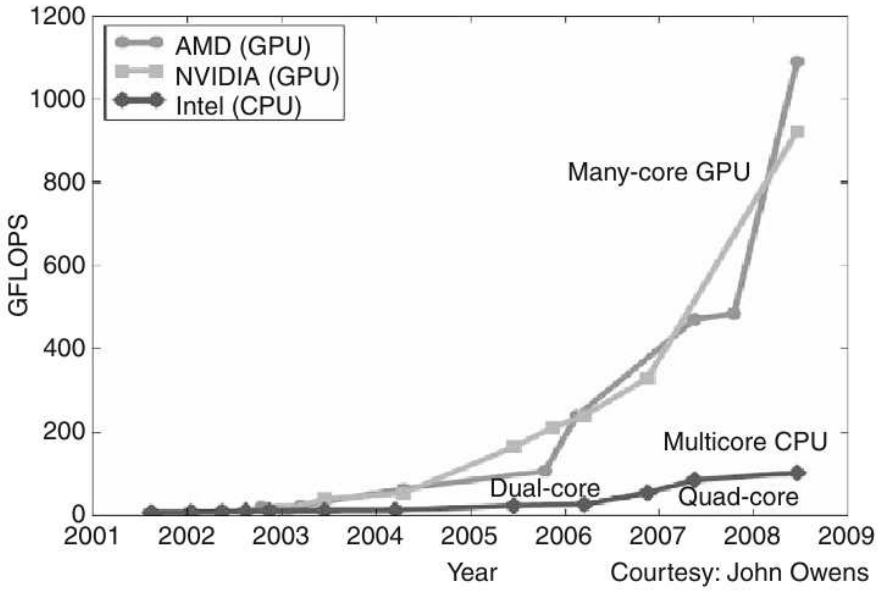


Figure 4.2: Performance between GPUs and CPUs.[Kirk and Hwu 2010]

hardware-independent interface that can be implemented on many different types of graphics hardware systems, or entirely in software (if no graphics hardware is present in the system) independent of a computer's operating or windowing system. As such, OpenGL doesn't include functions for performing windowing tasks or processing user input; instead, your application will need to use the facilities provided by the windowing system where the application will execute. Similarly, OpenGL doesn't provide any functionality for describing models of three-dimensional objects, or operations for reading image files (like JPEG files, for example). Instead, you must construct your three-dimensional objects from a small set of geometric primitives –points, lines, triangles, and patches.[Shreiner et al. 2013].

OpenGL has been in development for almost two decades, since its first release in July 1994. Significant changes have been done from the first versions to the latest ones. These include the replacement of fixed functionalities by programmable ones with the introduction of a specific programming language: the OpenGL Shading Language(GLSL). Generally speaking, an application using the modern OpenGL API can be describe in three steps: (1) data flow from CPU application to GPU; (2) data pass trough a sequence of processing stages in the GPU (Rendering Pipeline); (3) a rendered image is obtained. For the present work OpenGL 4.2 was employed, then many of features described in following sections are dependent on this version.

4.2.1 OpenGL Shading Language(GLSL)

The GLSL is a high-level programming language used to encode the programmable stages of OpenGL Rendering Pipeline, see Figure 4.3. It is a C-like language that has evolved together with OpenGL since its release with OpenGL 2.0 in 2004. Files written in GLSL are loaded to

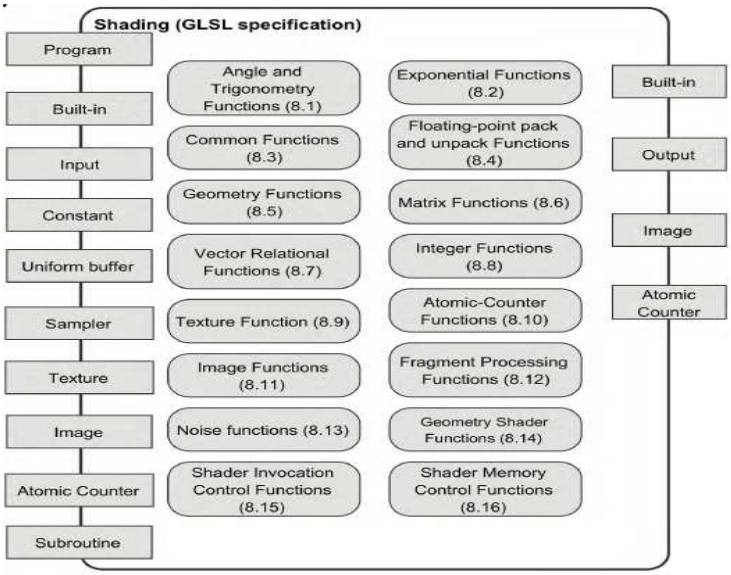


Figure 4.3: OpenGL 4.2 GLSL Specification [Cozzi and Riccio 2012].

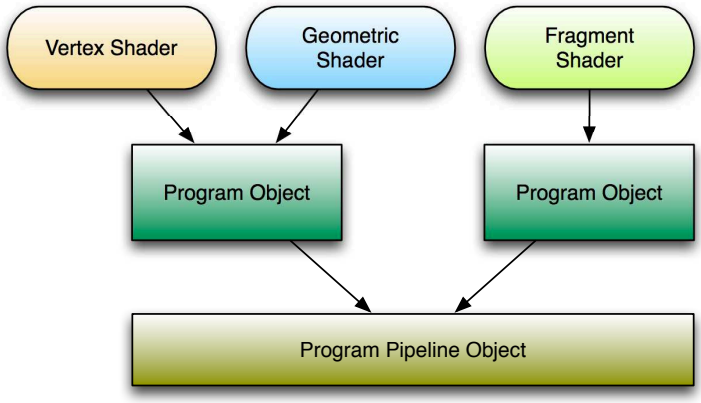


Figure 4.4: Program Pipeline Object Set Up.

OpenGL objects called *shaders*, these represent the programmable stages (mentioned above) and allow the compilation of the GLSL code. OpenGL *shaders objects* are attached to an OpenGL *program object*. It represent the compiled executable code, in GLSL, for one or more *shaders objects* and to link them to check if output of the previous stage corresponds to the input of the next one; e.g if the vertex shader output match with the fragment shader input or other optional shader following it. At the same time, OpenGL provide an object called *program pipeline*, that is a container of *program objects*; it allows to use multiple different *program objects* to set up all the programmable stages of the OpenGL Rendering Pipeline (the sequence of procedures to convert 3D data into a 2D image); with the *program pipeline objects* is possible to reuse and to combine the *shaders objects* in several *program objects*. The diagram in Figure 4.4 illustrates the above description.

With the current capabilities of GPUs, latest releases of GLSL have added several features that allow GPGPU use the rendering pipeline: e.g. now it is possible to read/write to textures

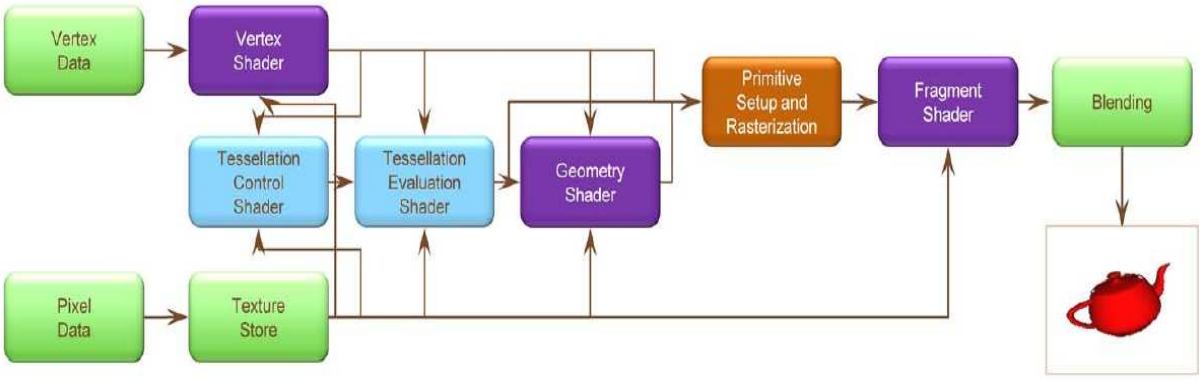


Figure 4.5: The OpenGL 4.X Rendering Pipeline [Angel and Shreiner 2011]

.

and to do atomic arithmetic operations over data through image objects. Because of this, it has been possible in our work, to do physically-based simulation with the same library that is used in the graphical part, avoiding the inclusion of additional computational tools.

4.2.2 Rendering Pipeline

The latest OpenGL Rendering Pipeline is shown in Figure 4.5. It represents an state machine where data provided by the application are processed. Boxes labelled as *Shader* are completely programmable stages using GLSL. First, data are passed from application to GPU (vertices and geometric primitives); then, data are processed through a sequence of shader stages: Vertex Shader, both Tessellation Shader and Geometry Shader; after this, the rasterizer[3] (not programmable yet) generates fragments for any primitive that is inside of the clipping region[4]; finally, a Fragment Shader process each generated fragment. The programmer has the control of which shader stages to employ and what they do. Not all shader stages are required, in general only Vertex and Fragment Shader must be included[5], the other are optional. In the following paragraphs, these stages used in this work are described in detail:

- *Vertex Shader:* For each vertex provided by the application the Vertex Shader is executed (in parallel) to process its respective information. Routines that are commonly done at this stages are: computation of vertex position (it generally implies to use transformation matrices); determining the vertex color using lighting computations; etc. No new vertex can be generated or existing one be removed, but additional information can be passed to next stages using vertex attributes (Section 4.2.1).
- *Geometry Shader:* Processes individual geometric primitives before rasterization, including to create new primitives and to transform or to remove them (e.g. convert triangles to lines). In this stages all vertices of a primitive (and its information from previous stage)

[3]To transform a virtual 3D scene in a 2D image.
[4]Region inside the viewing-volume considered in a CG application.
[5]When rasterizer is disabled, Fragment Shader can be omitted.

are accessible. Applications related with transform the shape of and object, e.g removing geometry based on some criteria, are straightforward in this shader.

- *Fragment Shader:* Determines the fragment final color and depth. In this stage a programmable control over the fragment color based on its screen position is allowed. Also, it is possible to determine which fragment can be drawn or not (fragment discard). Efficient shading techniques are implemented in this shader stage, e.g. deferred shading (only visible vertices are illuminated).

Each one of the above steps runs in parallel on the SPs of the GPU, one per each instance that they process. It means: *Vertex Shaders* run in parallel, one per each vertex stemmed by OpenGL; *Geometric Shaders* run in parallel, one per each geometric primitive specified as input; *Fragment Shaders* run in parallel, one per each fragment resulted from previous step. These features make OpenGL a useful tool for high performance graphics applications.

A lot can be written about OpenGL and GLSL, but it is out to the goal of this thesis. This session is an attempt to introduce the computational tool and the subset of features used in this work.

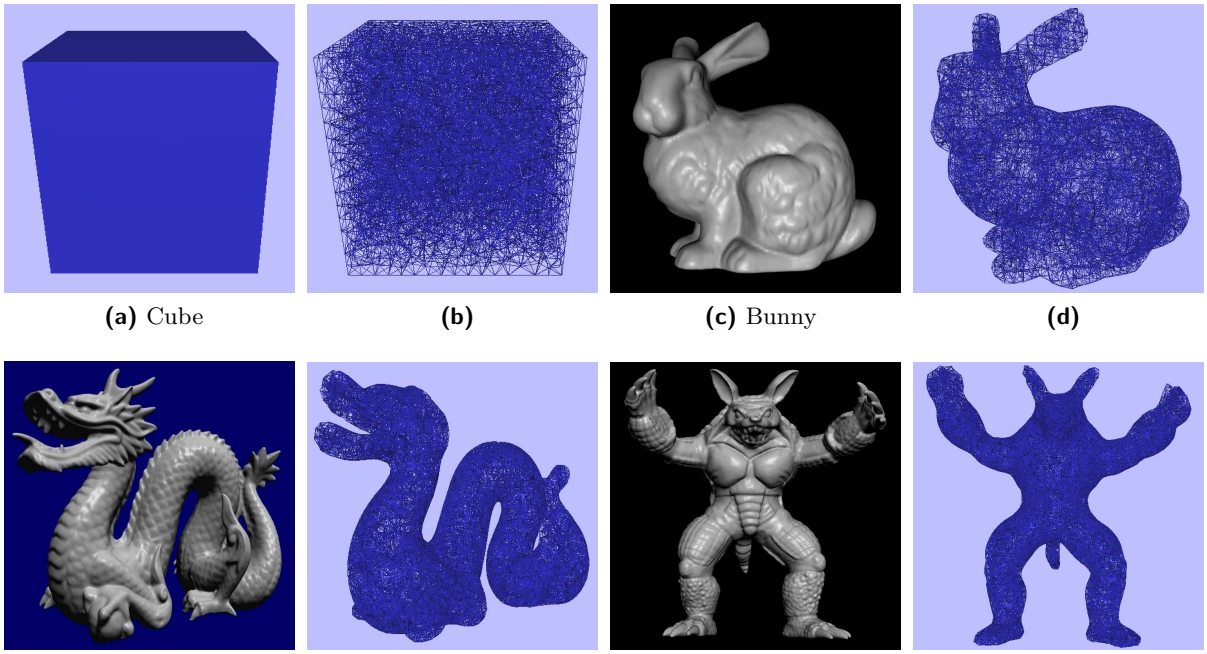
5 Implementation

With OpenGL as the principal computational tool (Section 4.2), in this work a c++ application was implemented. To display the images produced with OpenGL, the open source library *FreeGLUT*[1] was employed for managing the windowing system. For the purpose of comparing computing time, two different implementations of the algorithm 3.6 were done, one using C++ to run in CPU[2] and other using OpenGL Shaders (Section 4.2.1) to run in GPU[3]. In both cases the rendering part was done rather similar using OpenGL. In the following sections, the key issues during the implementation process are described in detail. Also, all the tools that were a useful support for extra tasks, as preparing the input tetrahedral meshes (e.g. holes filling), are mentioned.

5.1 Tetrahedral Meshes

The mass-spring system developed in this work is designed to work with volumetric solids represented as a tetrahedral mesh. The algorithm implemented receives a tetrahedral mesh as input. The mesh data are read from two text files: one lists all vertices, where the first line is the count of vertices and the remaining lines are the cartesian coordinates (x,y,z) of each vertex, one line per vertex. The other lists all tetrahedra, using the same format as the first file: the first line is the count of tetrahedra and the rest lines are four indices referencing four vertices positions in the list of vertices, that represent the nodes of a tetrahedron, also one line per tetrahedron.

[1] http://freeglut.sourceforge.net .
[2] This will be referred as CPU implementation.
[3] This will be referred as GPU implementation.



(a) Cube **(b)** **(c)** Bunny **(d)**

(e) Dragon **(f)** **(g)** Armadillo-Man **(h)**

Figure 5.1: Triangular Meshes and its volumetric discretization in tetrahedra.

Mesh	Vertices	Tetrahedra	Edges
Cube	3041	15484	19706
Bunny	4742	19507	27452
Bunny2	7133	35081	45417
Dragon	25156	102453	143704
Armadillo-Man	33276	145835	197798

Table 5.1: Tetrahedral Meshes Description. Columns $1, 2, 3$ show the numbers of vertices (point masses or particles), tetrahedra and edges respectively. *Bunny2* is *Bunny* refined.

Figure 5.1 shows example input tetrahedral meshes used in this work which are described in the Table 5.1. Tetrahedral meshes are obtained from triangular meshes[4] using the tool: TetGen[5]. However, poor quality meshes (having tetrahedra of zero volume, edges of zero length, holes, etc.) can affect the numerical stability of simulations. While TetGen does not give tetrahedra of zero volume or edges of zero length, for high resolution triangular meshes it can return volumes or lengths in the order 10^{-9} or less. To avoid these problems triangular meshes are scaled; also, the tool MeshLab[6] is employed for hole filling and to get low resolution triangular meshes from high resolution ones. But, still there may remain problems regarding high differences with the scale of the inverses of point masses, tetrahedra volumes and edges lengths when using irregular

[4]Triangular meshes used in this work came from http://www.cc.gatech.edu/projects/large_models .
[5]http://wias-berlin.de/software/tetgen .
[6]http://meshlab.sourceforge.net .

meshes. In order to illustrate the above differences, frequency histograms of the point masses inverse, tetrahedra volumes and edges lengths are presented in Figure 5.2.

As the point masses are set depending on the volume distribution (Section 3.4), to very small tetrahedra correspond very small point masses values, and therefore huge values of the inverses of the point masses: e.g. the Figure 5.2 shows that the dragon range of the values of the inverses of the point masses reaches $1.2e+10$. Huge values of the inverses of the point masses cause unstable numerical simulation because they are factor of the total force over a particle in the position update formulation, see the Section 3.5. Therefore, an artifice used in this work to obtain stable numerical simulations is to truncate the largest values of the inverses of the point masses: first, a mass quantity (kg) is given to distribute it between all particles of a mesh; then, all the values of the inverses of the point masses greater than a prefixed value are truncated to this value. Table 5.2 shows the values employed to produce stable simulations and the results described in Section 6. The meshes with the "worst" histograms in Figure 5.2 have higher percent of values of the inverses of the point masses over the truncation value. The value "$\cdots$" means that no values were truncated. The Sub-figures with caption **Mass_Inverse_Truncated** in the Figure 5.2 represent the values of the inverses of the point masses after they were truncated.

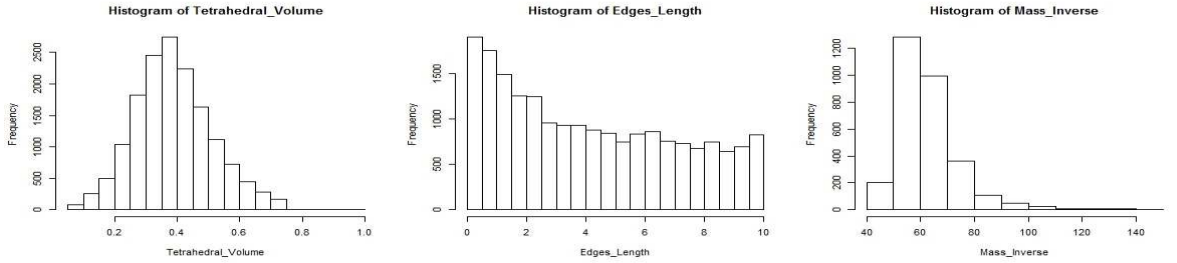
Mesh	Mass	Max Mass Inv.	Trunc. val	Over	Percent
Cube	50	143.04	$\cdots$	0	0
Bunny	75	9739.69	500	55	1.2
Bunny2	100	7667.24	500	93	1.3
Dragon	200	$1.34e+10$	250	16204	64.4
Armadillo-Man	200	$5.59e+09$	2000	13108	39.4

Table 5.2: Mass configuration set up to tetrahedral meshes. The second column is the mass quantity to be distributed between all particles of the object; third column is the maximum mass inverse value after distribution; fourth column is the value at which masses inverse values are truncated; fifth column is the amount of particles that have mass inverse value greater than the truncation value and the last column is what percent from all system particles represents the quantity in column 5.

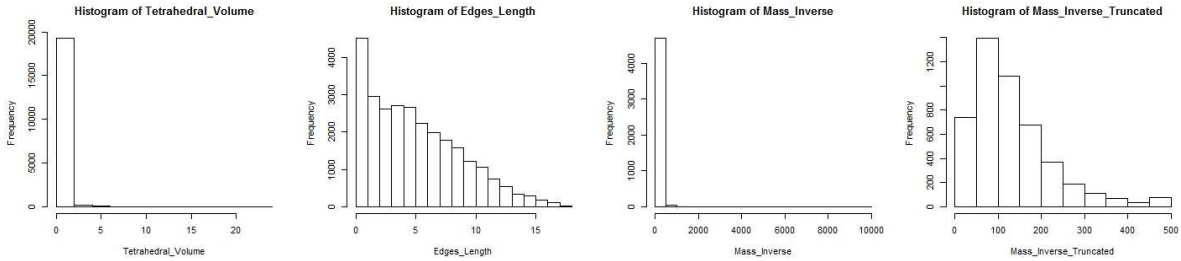
5.2 Memory Management

For the CPU implementation the memory management is done in an ordinary way. Data are read to RAM Memory and they are stored in lineal arrays in order to optimize memory space. This is important because data are always loaded to RAM Memory, even if they are transferred to GPU for GPU implementation. Cases of huge data amount, where the RAM Memory is not enough to store them, are out of the scope of this thesis but are considered as future work. Data are loaded in RAM as:

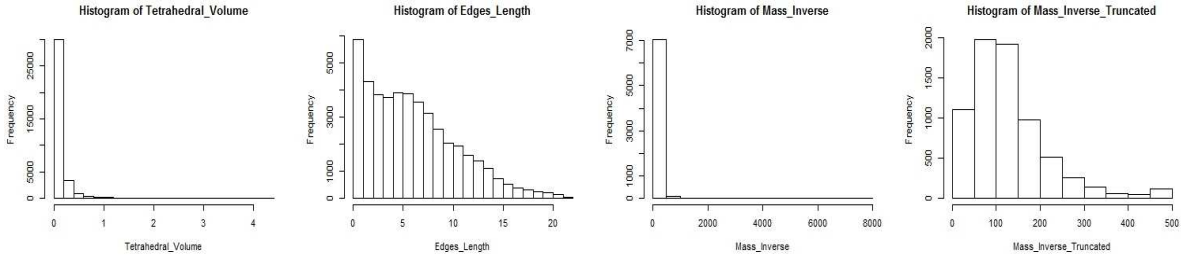
- A floating point array to store vertices coordinates (x, y, z). These are taken as the vertices initial positions and do not change during simulation time.



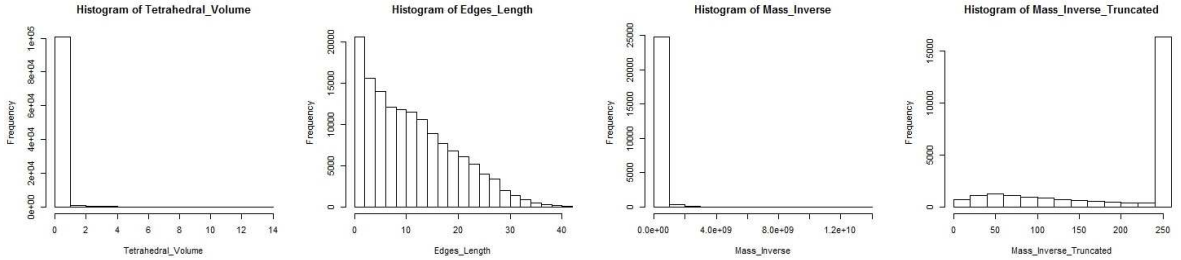
(a) Cube histograms



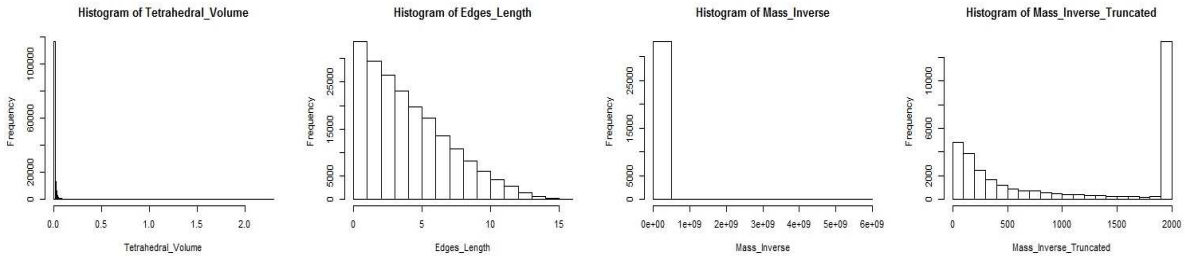
(b) Bunny histograms



(c) Bunny2 histograms



(d) Dragon histograms



(e) Armadillo-Man histograms

Figure 5.2: Meshes Quality Histograms. Less disproportionate histograms indicate higher quality meshes: e.g. cube and bunny.

- An unsigned integer array to store tetrahedral nodes.

these are read from hard drive (Section 5.1), and their dimensions are $3 \times vertices\_count$ and $4 \times tetrahedra\_count$ respectively. Additionally memory is reserved for:

- *edges:* an unsigned integer array of dimension $2 \times edges\_count$ to store only once all mesh edges. An algorithm was designed to get all edges given a tetrahedral mesh (see Algorithm 5.1).
- *mass:* a floating point array of dimension $vertices\_count$ to store the mass of each particle. Actually, the inverse of the mass is saved instead of the mass in order to get better performance on the simulation, avoiding unnecessary computations.
- *distance:* a floating point array of dimension $3 \times vertices\_count$ to store the distance covered for each particle (in x, y, z) since the start of the simulation.
- *velocity:* a floating point array of dimension $3 \times vertices\_count$ to store the velocity (v_x, v_y, v_z) of each particle.
- *force:* a floating point array of dimension $3 \times vertices\_count$ to store the total force, excluding the gravitational force, over each particle. Instead of using gravitational force, the gravitational constant g is employed explicitly as acceleration in each numerical integration step. Thus, a multiplication operation is avoided gaining in performance and in numerical correctness.

Algorithm 5.1: Find all edges from tetrahedral mesh

```
   Input: A tetrahedral mesh of the object
 1 std :: vector < uint >  matrix_edges[vertices_count − 1]
 2 for i = 0 to vertices_count − 1 do
 3 |   matrix_edges[i] = std :: vector < uint > ()
 4 end
 5 foreach tetrahedron t in mesh do
 6 |   sort the four indices in tetrahedron_nodes
 7 |   add indices in tetrahedron_nodes to matrix_edges
 8 end
 9 edges = new uint[edges_count]
10 edge_index = 0
11 for i = 0 to vertices_count − 1 do
12 |   for j = 0 to matrix_edges[i].size() do
13 |   |   edges[edge_index + +] = i
14 |   |   edges[edge_index + +] = matrix_edges[i][j]
15 |   end
16 end
```

In the Algorithm 5.1, *matrix_edges* (line number 1) is a data structure where per each vertex in the mesh, there is a list of its connected vertices. These connected vertices are inserted in order according to their index while browsing through the tetrahedra (function in line number 7). For two connected vertices i, j *with* $i < j$, only the index j is added to the list of vertex i;

so it is guaranteed that the same edge is inserted just once.

For the GPU implementation, memory inside the device is employed to avoid the data interchange between RAM and GPU. This allows a better performance because no time is spent in data transfer; data remain in the GPU during the whole simulation time. The type of memory chosen for storing the information inside the GPU (see in Figure 4.1), is an important issue to consider in the design of a solution because it can affect the application performance: e.g. accessing global memory is slower than accessing local memory. The OpenGL objects used for the GPU implementation have been set with this in mind:

- Vertices coordinates, masses, edges and tetrahedral indices are stored in GPU using *buffer objects* (described below).
- The whole information of distance, velocity and force are stored in GPU using *textures* (described in the following paragraphs).

Buffer Objects are defined in [Shreiner et al. 2013] as just chunks of memory managed by OpenGL server[7]. The usage mode of a buffer object is defined when it is created trough a target parameter. The buffer objects to store the vertices coordinates and masses are targeted as **GL_ARRAY_BUFFER**. So, at run time in the vertices processing step in GPU (see OpenGL Rendering Pipeline in Section 4.2.2), the coordinate and mass associated to each vertex are accessible from local memory by the *Vertex Shader* processing it. Moreover, the buffer objects for storing edges and tetrahedral indices are targeted as **GL_ELEMENT_ARRAY_BUFFER**. These indices represent references to the current information (vertices coordinates, point masses values) which is stored in **GL_ARRAY_BUFFER**, allowing to save memory. As result, at run time in the geometric primitives processing step, the whole information associated with the vertices present in a geometric primitive (an edge or a tetrahedron) is accessible as local memory by the *Geometric Shader*. Furthermore, the buffer objects are set up with the parameter **GL_STATIC_DRAW**, which means that they are populated with data only once and that these data are used many times in the GPU.

Textures are defined in [Shreiner et al. 2013] as large chunks of image data that can be used to paint the surfaces of objects to make them appear more realistic. However, for many more purposes, e.g in this work they are used as data containers with the required information to perform the simulation. The OpenGL version employed in this implementation (4.2) allows the data stored in textures to be accessible as global memory at any step of the rendering pipeline. This is possible through the OpenGL objects called *images*, that are bounded to the textures. This binding can be done at compiling or run time. Formatted data can be stored directly in textures only by specifying a format to the texture. Floating point data of distance and velocity are stored in **GL_RGB32F 2D** texture, a two dimensions texture with three channels of memory that contains 32-bits floating point data. Force data are stored in a **GL_R32F 2D**

[7] In many references, OpenGL API is presented as a client-server architecture where the server is the GPU; and the client is the part that runs in CPU making requests to the GPU.

texture, equivalent to the one describe above with only one channel. OpenGL *images* objects allows us to do atomic operations over data in textures. In this work: atomic read/write to texture that stored position information is done from the Vertex Shader for updating particles position; atomic add to texture that store force information is done from the Geometric Shader for gathering the force over each particle (details in Section 5.4). The atomic add operation is an special case because the standard OpenGL only implements it (at its last version 4.3) over integer or unsigned integer textures. There are two options for doing it over floating point textures: (1) to write a function that encapsulates floating point numbers in integers as the one presented in [Crassin and Green 2012]; (2) to use a specific vendor extension, e.g. in this work **GL_NV_shader_atomic_float** extension from *NVIDIA* is employed. The use of a specific vendor extension gives better performance because it offers a hardware implementation, but it requires the use of a specific hardware: the current implementation is restricted to the use of *NVIDIA* hardware.

5.3 Numerical Simulation

The issues to deal with floating point numbers is a well discussed topic in the area of numerical mathematics. In this work the main issue to overcome was comparing floating point numbers for near equality. To reduce the effect of numerical errors, the distance covered by each particle is considered instead absolute position, for internal force calculation. But even so, numerical errors rise: e.g. for an object in free fall, the distance between connected particles differs a bit from their resting distance after few simulation steps. The numerical error that produces the above difference is: given the floating point numbers A, B, γ the operation $[A-B == (A+\gamma)-(B+\gamma)]$ is not always true, e.g when γ is too small in comparison with A or B. This error can cause unexpected internal forces producing unstable simulations. The following code illustrates the gap with 32 floating point numbers:

```
int a = 1.1f;
return ((a + a + a) - 3.3f);
```

the return value is around $2.38419e-07$. To deal with the issue of comparing floating point numbers for near equality in this work, the advice given by D. Knuth in [Knuth 1997] and enunciated by Chris Lomont[8] is followed: a good floating point routine should claim two floats are nearly equal not if their *absolute* error[9] is bounded by a tolerance, but if if their *relative* error[10] is bounded by some tolerance. So, the following function is used:

```
bool KnuthCompare(float a, float b, float relError)
{
    return ( abs(a-b) <= relError*max(abs(a),abs(b)) );
}
```

[8] http://www.lomont.org/Math/Papers/2005/CompareFloat.pdf .
[9] given to numbers a, b the absolute error is $|a-b|$.
[10] given to numbers a, b the relative error is $\frac{|a-b|}{|a|}$.

using $\leq$ instead $<$ ensures that the function works correctly in the case $a = b = 0.0f$. There is not a specific way to figure out a good value for $relError$. In this work good results have been obtained with $relError = 1.0e - 06$.

5.4 Algorithm Implementation

The CPU implementation of the Algorithm 3.1 is straightforward. The key issue of this implementation is the transfer of data from RAM to GPU for the rendering task. Because of the simulation runs in CPU, computed distance information has to be sent to GPU to visualize the results. Fortunately, for performance purposes, data transfer is needed only in one direction: from RAM to GPU. To achieve this the technique described in [Hrabcak and Masserann 2012] is used (Buffer Respecification (Orphaning)). This consists in calling the function *glBufferData* to fill data in buffer object with a NULL pointer and after recall specifying the data. The following code shows this technique:

```
glBindBuffer( GL_ARRAY_BUFFER , my_buffer_object);
glBufferData( GL_ARRAY_BUFFER , data_size, NULL, GL_STREAM_DRAW);
glBufferData( GL_ARRAY_BUFFER , data_size, mydata_ptr, GL_STREAM_DRAW);
```

For the GPU implementation four OpenGL *program pipeline objects* (Section 4.2.1) were developed, three for the physically-based simulation an one for the rendering. An overall of eight *shaders* were written for the four *program pipeline objects*:

- Program pipeline for distance preserving force computation: vertex and geometric shader.
- Program pipeline for volume preserving force computation: vertex and geometric shader.
- Program pipeline for position updating: vertex shader.
- Program pipeline for rendering: vertex, geometric and fragment shader.

Each simulation step executes three OpenGL rendering commands[11], each one with its respective *program pipeline object* bound:

The first command is for distance preserving force computation: **glDrawElements** with a *buffer object* containing the edges (Section 5.2) bound. This command executes, in parallel, a *vertex shader* per each vertex from the edges list. In this step information of position and mass[12] is read from *buffer objects* and information of distance and velocity is read from *textures* through *image objects*. After, the information is sent to the next step: the *geometric shader*, which is executed, also in parallel, once per edge. Now, in this new step, with the information of both vertices of an edge, distance preserving force is computed (Equation 3.7) and written to the *force texture*, also through *image objects*. With the option **GL_RASTERIZER_DISCARD** enabled, the pipeline is truncated after the *geometric shader* step.

[11] An OpenGL function that triggers the rendering pipeline in the GPU .
[12] Mass information is used to apply strain limiting, Section 3.3.2 .

The second command is for volume preserving force computation (Equation 3.8), it is also **glDrawElements** but now a *buffer object* containing the tetrahedra is bound. This command executes analogously to the previous one described. Also the pipeline is truncated using the option **GL_RASTERIZER_DISCARD**, to avoid the rendering stage.

The third command is to update the point masses position: **glDrawArrays**. This command executes in parallel a *vertex shader* per each vertex of the mesh. In this step the position updating is achieved solving a numerical integration method (Section 3.5). For the Verlet Method: information of distance, previous distance, velocity and force is read from *textures*; then a numerical step is computed (Equation 3.17). Finally, the results are saved to *textures*. This time only the *vertex shader* step is needed, and the pipeline is truncated in a similar way to the previous commands.

5.4.1 Rendering

The implementation of the rendering part is rather similar for the CPU and the GPU simulation. This thesis is focused in physically-based simulation, so not advanced rendering technique was implemented but it is considered as future work. The rendering command called is **glDrawElements** with a *buffer object* containing the tetrahedra bound. In the *vertex shader* information of position is sent to the *geometric shader* that receive primitives of the type **lines_adjacency** and produces **triangle_strip** with four triangles that represent the face of a tetrahedron. In this step the normals of the tetrahedron vertices are computed and sent to the *fragment shader*. In the *fragment shader* the final color of each pixel is determined applying the Phong Illumination Model ([Phong 1975]).

In order to achieve the best performance, the underlying implementation of memory transactions in OpenGL is caching-based. This implies that: e.g. in a simulation step the second command can begin its execution while the information written by the first command in not in the *texture* yet, causing an undefined simulation behaviour. To solve this issue a memory barrier provided by OpenGL is used. The function **glMemoryBarrier** *defines a barrier ordering memory transactions issued before the command relative to those issued after the command. Memory transactions performed by shaders are considered to be issued by the rendering command that invoked the execution of the shader.* The parameter **GL_SHADER_IMAGE_ACCESS_BARRIER_BIT** *specifies that data read from an image variable in shaders executed by commands after the barrier should reflect data written into those images by commands issued before the barrier* [Shreiner et al. 2013]. A memory barrier is placed between each of the above rendering commands.

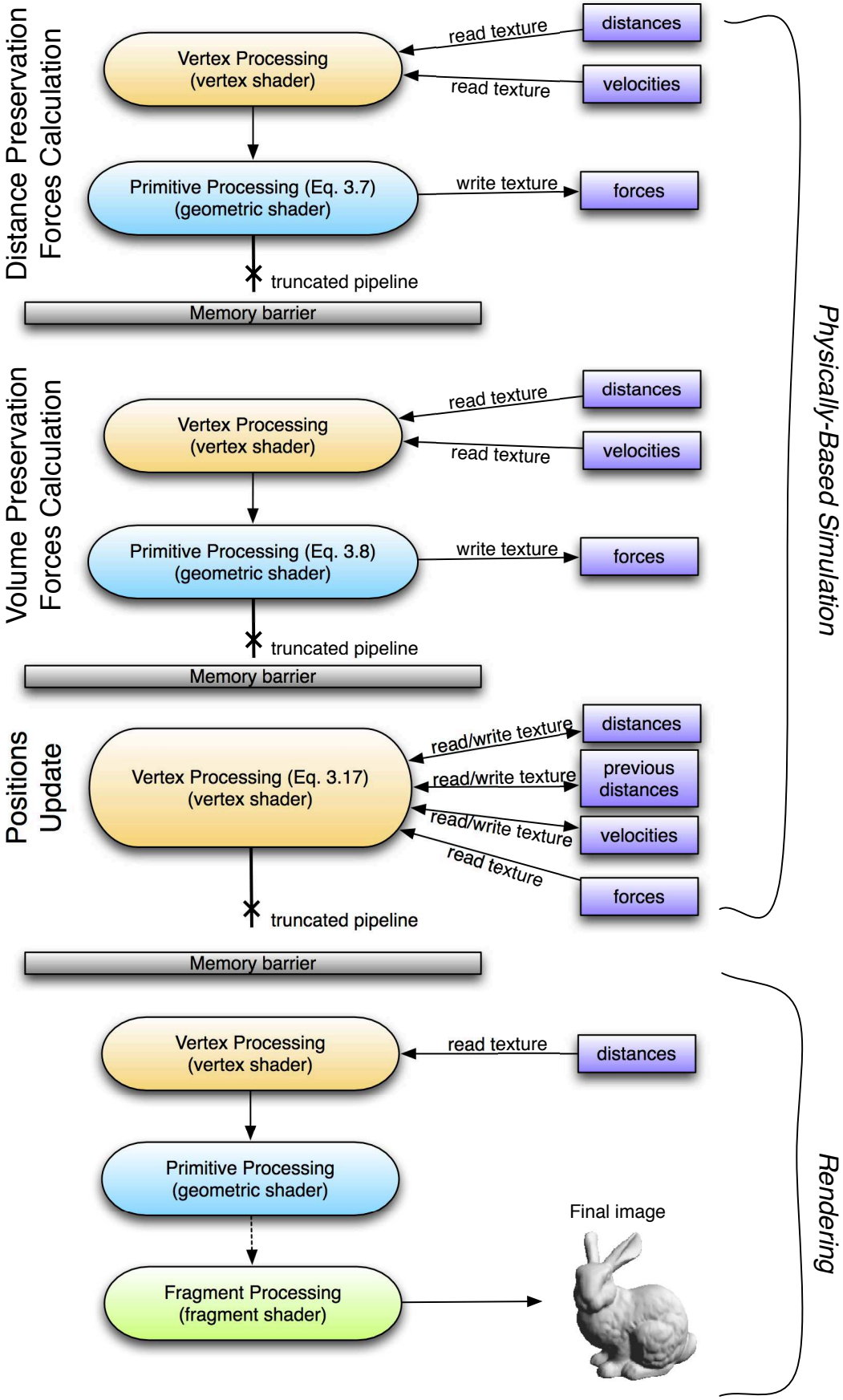


Figure 5.3: Diagram of the GPU implementation described in this section.

6

Results and Discussion

The implementations (CPU and GPU) described in the Section 5 were tested several times with the tetrahedral meshes in Figure 5.1. The experimental proposed to demonstrate the performance of the volumetric deformable solids simulation implemented, is to drop an object in free fall from a given height over a flat floor. For this experiments we have not considered friction forces between the floor and the deformable object. Although this work does not cover collision detection response between rigid and deformable objects or between deformable and deformable objects (neither self-collision of deformable objects), an strategy of collision detection response between individual particles and the plane has been implemented. This strategy is applied between each particle in the mass-spring system and the floor. When a point mass is found to penetrate the floor ($n_f(x_p - x_f) < 0$, where n_f is the normalized floor normal, x_f is the position of a point in the floor plane and x_p is the mass point position), it is projected out the floor through its normal direction and its velocity is cancelled (no bouncing). This strategy is based on the idea introduced in [Provot 1995].

The first results presented in this work show a comparison between the four numerical integration methods exposed in Section 3.5, which were implemented in CPU. The numerical simulations have been done with a cube of 1602 point masses, 7647 tetrahedra and 9997 edges. The constants of distance and volume preservation forces (Equations 3.7 and 3.8) are set to 50 and the total mass[1] to 50kg. Damping coefficients for distance and volume preservation forces (Section 3.3.1) are both 0.1 for *Euler* and *Midpoint* methods; and $0.001, 0.01$ respectively for *Verlet* and *Euler-Cromer* methods. The Table 6.1 shows the results given when the Algorithm

[1]The Mass quantity to be distributed between all object particles.

3.1 is executed using each of the numerical schemas in the update position routine (lines 13-15). Also, the collision detection response is evaluated in this routine. In this work, the numerical stability of each method is evaluated qualitatively according the behaviour of the deformable objects in the simulation.

Method	**CPU**		
	comp. time	**time step**	**ratio**
Euler	9.6	3.0	0.313
Midpoint	14.3	3.5	0.245
Verlet	9.6	6.5	0.677
Euler-Cromer	9.6	6.5	0.677

Table 6.1: Comparison of Numerical Integration Methods. The first two columns show the average of 1000 samples of computing time measured and the maximum time step which get stable numerical simulation for each method (in milliseconds). The last column shows the ratio between the time step and the computing time as a performance measure of the numerical integration schemes.

The *Midpoint* method gets the worse performance and, equal to the *Euler* method, requires small time step to produce stable numerical simulation. The *Verlet* and *Euler-Cromer* methods get better performance. From these results we decided to implement, only *Verlet* and *Euler-Cromer* methods in GPU using GLSL (Section 4.2.1). Several simulations for each tetrahedral mesh in the Figure 5.1 were executed with both methods. Computing time comparison between the CPU and the GPU implementation is the content of the Section 6.1.

6.1 Computing Time

The *Verlet* and *Euler-Cromer* (Section 3.5) methods have some differences: the *Verlet* method has accuracy $O(h^4)$ in position update calculation and the *Euler-Cromer* method $O(h)$; the *Verlet* method does not use velocity information for position update, but the *Euler-Cromer* method use a semi-implicit scheme where position is updated using the velocity at the next step; the *Verlet* method requires the information of previous step position which implies more memory requirements than the *Euler-Cromer* method and, at least, two more accesses to textures in GPU. The *Verlet* method does two multiplications and an add operation more than the *Euler-Cromer* method to perform position update. Considering the above differences, the last two may affect the computing performance. However, do not exist difference between the computing time of these methods (in the scale of milliseconds), even for large meshes such as the Armadillo-Man mesh (Table 5.1). The same computing time is obtained by both methods when using the same tetrahedral mesh, in the CPU and the GPU implementation. Also, visually, the deformable objects have the same behavior in the animations produced (Figure 6.2). These shows the current optimization of arithmetic and memory access operations in CPU and GPU.

The Figure 6.1 shows the computing time comparison for each mesh between the CPU and

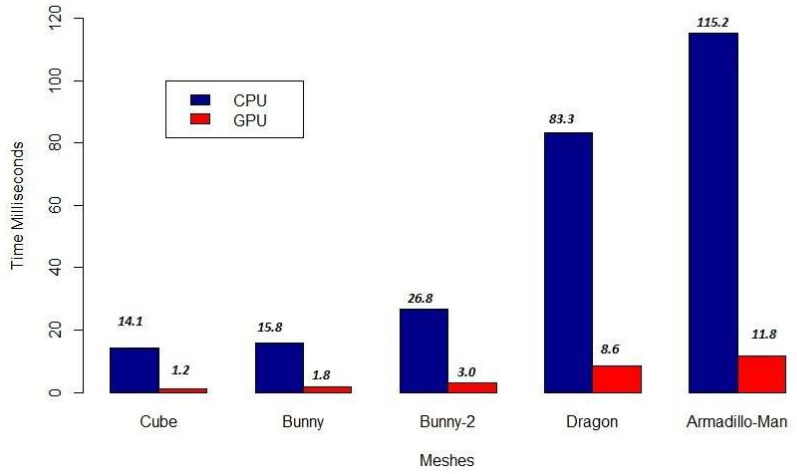
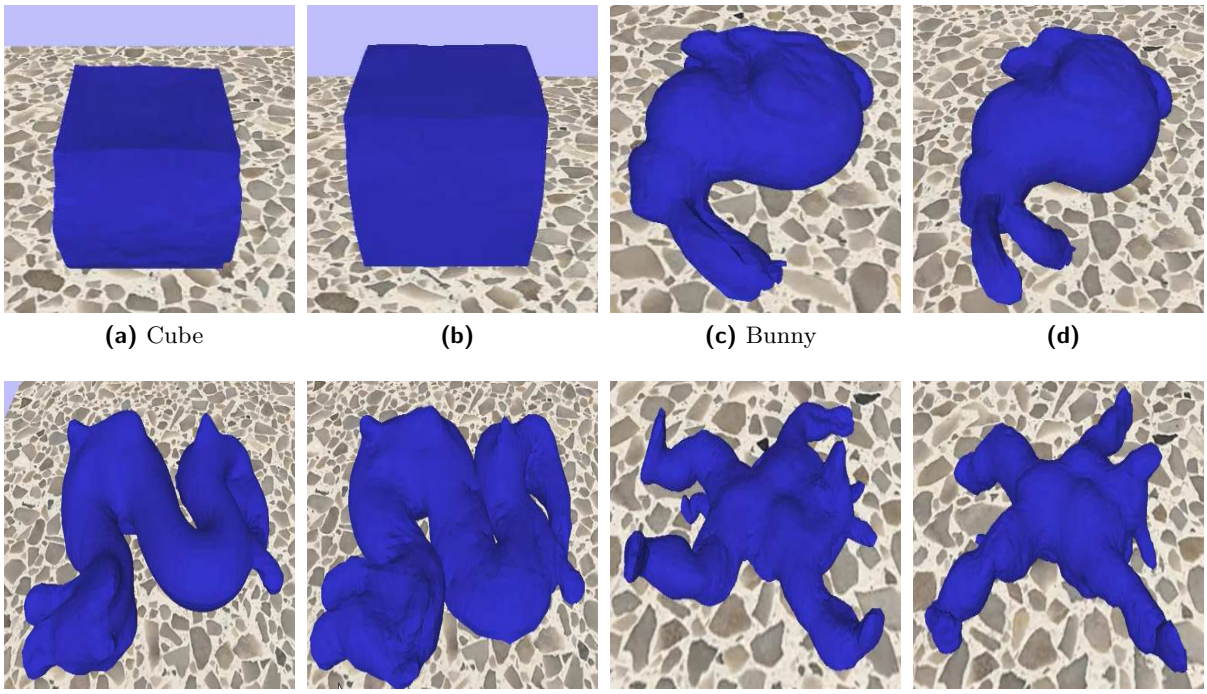


Figure 6.1: Computing time comparison between the CPU and the GPU implementation for each of the presented tetrahedral meshes (Figure 5.1): (1) Cube, CPU:11.4ms, GPU:1.2ms; (2) Bunny, CPU:15.8ms, GPU:1.8ms; (3) Bunny2, CPU:26.8ms, GPU:3.0ms; (4) Dragon, CPU:83.3ms, GPU:8.6ms; (5) Armadillo-Man, CPU:115.2ms, GPU:11.8ms. These values are the average of 1000 samples of the physically-based simulation time spent (including collision detection-response with the floor and deformation limits verification, Section 3.3.2) for computing each step over each mesh. (Figure 6.2)



(a) Cube **(b)** **(c)** Bunny **(d)**

(e) Dragon **(f)** **(g)** Armadillo-Man **(h)**

Figure 6.2: Physically-based deformable objects simulation. Several videos of the graphics application execution, implemented as part of this research, were produced. These figures are scenes of those animations.

Mesh	Frames per second	
	CPU	GPU
Cube	+60	+60
Bunny	+60	+60
Bunny2	37	+60
Dragon	12	+60
Armadillo-Man	8	30

Table 6.2: Frames per second rate of the CPU and the GPU implementation. With the basic functionality of *freeglut* only up to rate of 60 fps is possible. The value +60 means that the application fps rate in greater than 60.

the GPU implementations. The GPU implementation shows an order of 10× speed up over the CPU implementation. The computing time measured includes only the physically-based simulation, but also the *frames per second* (fps) rate of the graphics application (displaying on *freeglut*, Section 5) is computed and shown in the Table 6.2. The fps values with the computing time values provide and overview of the time spent in the rendering process of huge meshes, e.g the Armadillo-Man mesh takes only 11.8ms and because of it runs at $30fps$, the rendering of all tetrahedra takes around 21ms. The time of rendering process is not measured in this work because its focus on the physically-based simulation.

6.2 Simulation Parameters

The Table 6.3 shows the parameter set up for distance and volume preservation forces (Equations 3.7 and 3.8) computing, over the meshes described in the Tables 5.1 and 5.2. With this configuration and using non physically-based tricks such as strain limiting (Section 3.3.2) and point masses values truncation (Section 5.1) stable and realistic physically-based deformable solids simulation are obtained in this work (Figure 6.2). The Figure 6.3 reflects the effects of forces configuration parameters: e.g. adjusting the constant of distance preservation force it is possible to mimic solids of hard or soft materials; the sub-figures in the first row models a rigid Bunny while the ones on second row models a soft Bunny which behaves as a melting object, it has a large constant volume preservation force and a small constant of distance preserving force.

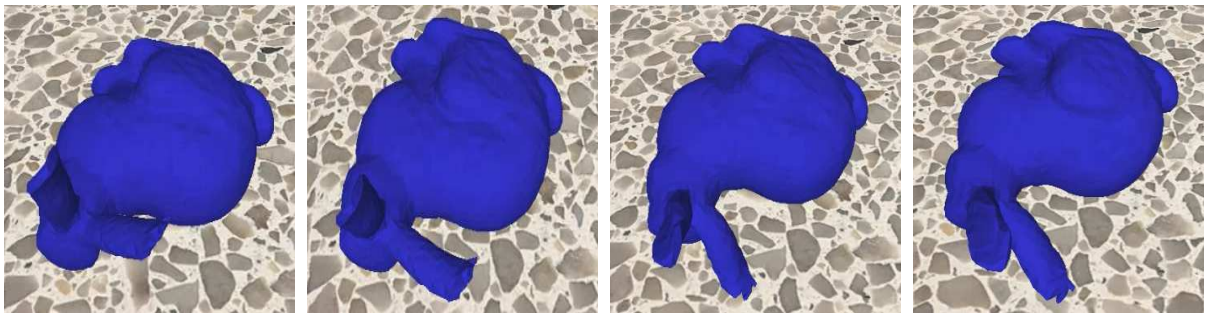
6.2.1 Parameters Selection

None of the references, about volumetric deformable objects simulation using mass-spring model, studied during this research (Section 2.1.1) addresses the issue of the parameter selection or estimation for archiving stable numerical simulation. In general, in most mass-spring deformable objects publications, the physical parameters are determined through a trial and error approach. Several works ([Baraff and Witkin 2003]) explain how to estimate the numerical error of the

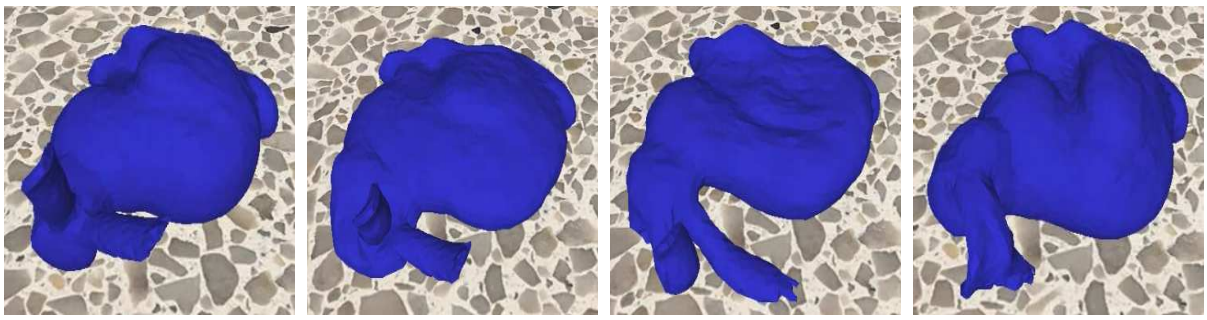
Mesh	Dist.P forces		Vol.P forces		step
	const.	damp.	const.	damp.	
Cube	50	0.001	25	0.01	0.008
Bunny	50	0.001	20	0.01	0.005
Bunny2	50	0.001	20	0.01	0.005
Dragon	30	0.001	20	0.002	0.0025
Armadillo-Man	40	0.00001	20	0.001	0.0025

Table 6.3: Distance and volume preservation forces configuration parameters. Constant and damping coefficient.

numerical integration schemes but there are not many published works about the determination or estimation of the physical parameters in mass-spring deformable objects simulation: e.g. only a recent work ([Natsupakpong and Çavuşoğlu 2010]) closed to this topic was found in this research, in this work the authors obtain the point masses and the springs constant of a mass-spring model taking as reference a closed finite element model that they solve. So, the above arguments demonstrate that is a costly task and almost a harder problem to found the physical parameters for stable physically-based simulation for mass-spring system. Here, the trial and error approach is used jointly to a set of non-physical tricks, mentioned above. It is notable that for "high quality meshes" (Figure 5.2) a wide set of physical parameters are available for getting stable numerical simulation, but its difficult and time consuming to find a few for "bad quality meshes" (such as the Dragon and Armadillo-Man meshes).



(a) Rigid Bunny **(b)** **(c)** **(d)**



(e) Melting Bunny **(f)** **(g)** **(h)**

Figure 6.3: Distance and preservation forces configuration effect. Rigid Bunny has distance and volume preservation forces constants 100 and 10, respectively. Melting Bunny has distance and volume preservation forces constants 10 and 30, respectively.

7 Conclusions

The real-time physically-based simulation is a modern research topic due to the development of current GPU and CPU architectures. Therefore, most references related to this topic cited in this document are recent published works.

This thesis has presented a real-time application for physically-based deformable objects. A *mass-spring system* that considers volumetric deformable solids represented with tetrahedral meshes has been implemented using GLSL. This work demonstrates that modern OpenGL allows us to go further than to implement only rendering projects, introducing some of its most recent features for GPGPU applications. We have shown that the *mass-spring system* is a very intuitive model, computationally in expensive and highly customizable to get physical or non-physical behaviour in deformable objects simulation. However, it is also shown that there does not exist a simple way to find a set of parameters, when employing explicit numerical integration methods, for obtaining stable simulations of non-regular complex geometries. Because of this, we employ some non-physical techniques in order to achieve numerical stability of our physically-based approach of deformable objects simulation. But, the realistic physical behavior of the deformable objects, in the animations shown, is not affected, at least to the spectator view, and a wide range from soft to hard materials can be mimic. The performance of several numerical integration methods have been evaluated and discussed. As well, we present comparisons between the GPU implementation speed up over the CPU one.

This research was focused in the development of a solution for real-time physically-based deformable objects simulation using GLSL and therefore, code optimization was not a main

concern. For this, code optimization can still be done in the physical simulation implementation or another rendering, computationally less expensive, can be implemented. However, with the current implementation we reach computing time less than 12ms for meshes that contain more than 145000 tetrahedra at a rate greater than $30fps$.

7.1 Future Work

Due to the time limitations of a masters thesis, some interesting problems could not tackled. Also, useful aspects in order to insert the developed algorithm into a motion planning application for a deformable robot could not be addressed. The following list tries to enumerate some of them:

- The implementation of implicit numeral integration methods in GPU with GLSL.
- The design and implementation of a real-time strategy for collision detection and response between rigid/deformable and deformable/deformable objects.
- The design and implementation of a real-time strategy for self-collision detection and response of deformable objects.
- The implementation and performance comparison with other GPU-based approaches as CUDA or OpenCL.

References

Akenine-Möller, T., Haines, E., and Hoffman, N. 2008. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.
1 citation(s) on 1 page(s): 11,

Angel, E. and Shreiner, D. 2011. An introduction to modern opengl programming.
1 citation(s) on 1 page(s): 26,

Arnab, S. and Raja, V. 2008. A deformable surface model with volume preserving springs. In *Proceedings of the 5th international conference on Articulated Motion and Deformable Objects*. AMDO '08. Springer-Verlag, Berlin, Heidelberg, 259–268.
1 citation(s) on 1 page(s): 10,

Baraff, D. and Witkin, A. 2003. Siggraph 2003 course notes.
3 citation(s) on 2 page(s): 6 and 41,

Barbič, J. and James, D. L. 2005. Real-time subspace integration for st. venant-kirchhoff deformable models. *ACM Transactions on Graphics (SIGGRAPH 2005) 24,* 3 (aug), 982–990.
2 citation(s) on 1 page(s): 4,

Bell, N., Yu, Y., and Mucha, P. J. 2005. Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA '05. ACM, New York, NY, USA, 77–86.
1 citation(s) on 1 page(s): 4,

Carlson, M., Mucha, P. J., Van Horn, III, R. B., and Turk, G. 2002. Melting and flowing. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA '02. ACM, New York, NY, USA, 167–174.
1 citation(s) on 1 page(s): 4,

Chen, Y., Zhu, Q.-h., Kaufman, A. E., and Muraki, S. 1998. Physically-based animation of volumetric objects. 154–160.
2 citation(s) on 1 page(s): 7,

Cozzi, P. and Riccio, C. 2012. *OpenGL Insights*. CRC Pressnote.
1 citation(s) on 1 page(s): 25,

Crassin, C. and Green, S. 2012. Octree-based sparse voxelization using the gpu hardware rasterizer. In *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. CRC Press, 303–318.
1 citation(s) on 1 page(s): 34,

Diziol, R., Bender, J., and Bayer, D. 2009. Volume conserving simulation of deformable bodies. In *Proceedings of Eurographics*. Munich (Germany).
2 citation(s) on 1 page(s): 10,

Eberly, D. H. and Shoemake, K. 2004. *Game Physics*. Interactive 3d Technology Series. Elsevier/Morgan Kaufmann.
2 citation(s) on 2 page(s): 16 and 17,

Erleben, K., Sporring, J., Henriksen, K., and Dohlman, K. 2005. *Physics-based Animation (Graphics Series)*. Charles River Media, Inc., Rockland, MA, USA.
5 citation(s) on 5 page(s): 4, 7, 14, 16, and 17,

Feldman, B. E., O'Brien, J. F., and Klingner, B. M. 2005. Animating gases with hybrid meshes. In *Proceedings of ACM SIGGRAPH 2005*.
1 citation(s) on 1 page(s): 4,

Georgii, J. 2008. Real-time simulation and visualization of deformable objects. Ph.D. thesis, Technische Universität München. http://mediatum2.ub.tum.de/node?id=627732.
5 citation(s) on 4 page(s): 4, 9, 20, and 22,

Georgii, J., Echtler, F., and Westermann, R. 2005. Interactive simulation of deformable bodies on gpus. In *Proceedings of Simulation and Visualisation 2005*. 247–258.
4 citation(s) on 3 page(s): 2, 9, and 16,

Georgii, J. and Westermann, R. 2005. Mass-spring systems on the gpu. *Simulation Modelling Practice and Theory 13*, 693–702.
1 citation(s) on 1 page(s): 9,

Gibson, S. F. F. and Mirtich, B. 1997. A survey of deformable modeling in computer graphics. Tech. rep.
4 citation(s) on 3 page(s): 3, 4, and 6,

Hrabcak, L. and Masserann, A. 2012. Asynchronous buffer transfers. In *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. CRC Press, 391–414.
1 citation(s) on 1 page(s): 35,

Kirk, D. B. and Hwu, W.-m. W. 2010. *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
3 citation(s) on 3 page(s): 22, 23, and 24,

Knuth, D. E. 1997. *Seminumerical Algorithms*, Third ed. The Art of Computer Programming, vol. 2. Addison-Wesley.
1 citation(s) on 1 page(s): 34,

Lasseter, J. 1987. Principles of traditional animation applied to 3d computer animation. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '87. ACM, New York, NY, USA, 35–44.
1 citation(s) on 1 page(s): 3,

LaValle, S. M. and Kuffner Jr., J. J. 2001. Randomized kinodynamic planning. *I. J. Robotic Res. 20,* 5, 378–400.
1 citation(s) on 1 page(s): 1,

Matyka, M. and Ollila, M. 2003. Pressure model of soft body simulation. *Proc. of Sigrad, UMEA, 2003*, 20–21.
1 citation(s) on 1 page(s): 8,

Maule, M., Maciel, A., and Nedel, L. 2010. Efficient collision detection and physics-based deformation for haptic simulation with local spherical hash. In *Proceedings of the*

2010 23rd SIBGRAPI Conference on Graphics, Patterns and Images. SIBGRAPI '10. IEEE Computer Society, Washington, DC, USA, 9–16.
1 citation(s) on 1 page(s): 8,

Min Hong, Sunwha Jung, M.-H. C. and Welch, S. 2006. Fast volume preservation for a mass-spring system. *IEEE Computer Graphics and Applications 26*, 83–91.
2 citation(s) on 1 page(s): 10,

Mosegaard, J., Herborg, P., and Sørensen, T. S. 2005. A gpu accelerated spring mass system for surgical simulation. *Studies in health technology and informatics 111*, 342–348.
1 citation(s) on 1 page(s): 8,

Müller, M., Charypar, D., and Gross, M. 2003. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA '03. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 154–159.
1 citation(s) on 1 page(s): 4,

Natsupakpong, S. and Çavuşoğlu, M. C. 2010. Determination of elasticity parameters in lumped element (mass-spring) models of deformable objects. *Graphical Models 72,* 6, 61–73.
1 citation(s) on 1 page(s): 42,

Nealen, A., Mueller, M., Keiser, R., Boxerman, E., and Carlson, M. 2006. Physically based deformable models in computer graphics. *Computer Graphics Forum 25,* 4 (dec), 809–836.
4 citation(s) on 2 page(s): 3 and 4,

Pauly, M., Keiser, R., Adams, B., Dutré, P., Gross, M., and Guibas, L. J. 2005. Meshless animation of fracturing solids. In *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. ACM, New York, NY, USA, 957–964.
1 citation(s) on 1 page(s): 4,

Phong, B. T. 1975. Illumination for computer generated pictures. *Commun. ACM 18,* 6 (jun), 311–317.
1 citation(s) on 1 page(s): 36,

Provot, X. 1995. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *IN GRAPHICS INTERFACE*. 147–154.
1 citation(s) on 1 page(s): 38,

Sederberg, T. W. and Parry, S. R. 1986. Free-form deformation of solid geometric models. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '86. ACM, New York, NY, USA, 151–160.
1 citation(s) on 1 page(s): 1,

Selle, A., Lentine, M., and Fedkiw, R. 2008. A mass spring model for hair simulation. *ACM Transactions on Graphics 27,* 3 (aug).
2 citation(s) on 1 page(s): 11,

Shreiner, D., Sellers, G., Licea-kane, B., and Kessenich, J. M. 2013. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 4.1*. Graphics programming. Addison Wesley Professional.
4 citation(s) on 3 page(s): 24, 33, and 36,

Tejada, E. and Ertl, T. 2005. Large steps in gpu-based deformable bodies simulation. *Simulation Practice and Theory. Special Issue on Programmable Graphics Hardware 13,* 9, 703–715.

1 citation(s) on 1 page(s): 9,

Teran, J., Blemker, S., Hing, V. N. T., and Fedkiw, R. 2003. Finite volume methods for the simulation of skeletal muscle. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA '03. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 68–74.

1 citation(s) on 1 page(s): 4,

Terzopoulos, D., Platt, J., Barr, A., and Fleischer, K. 1987. Elastically deformable models. *SIGGRAPH Comput. Graph. 21,* 4 (aug), 205–214.

2 citation(s) on 2 page(s): 3 and 4,

Teschner, M., Heidelberger, B., Müller, M., and Gross, M. 2004. A versatile and robust model for geometrically complex deformable solids. In *Proc. CGI, Crete, Greece*. 312–319.

9 citation(s) on 7 page(s): 4, 8, 12, 14, 15, 16, and 17,

Vassilev, T. and Rousev, R. 2008. Algorithm and data structures for implementing a mass-spring deformable model on gpu. In *Biomedical Physics Papers, Research and Laboratory University Ruse*.

1 citation(s) on 1 page(s): 7,

Vassilev, T. and Spanlang, B. 2002. A mass-spring model for real time deformable solids. In *East-West Vision*.

1 citation(s) on 1 page(s): 7,