Large Steps in GPU-based Deformable Bodies Simulation [⋆]

Eduardo Tejada [a,*] Thomas Ertl [a]

[a] *University of Stuttgart, Institute for Visualization and Interactive Systems, Universitätstr. 38, 70569 Stuttgart, Germany.*

Abstract

The interactive deformation and visualization of volumetric objects is still a challenging problem for many application areas. We present a novel integrated system which implements physically based deformation and volume visualization of tetrahedral meshes on modern graphics hardware by exploiting the last features of vertex and fragment shaders.

We achieve fast and stable deformation of tetrahedral meshes with the GPU-based implicit solver and we present a hardware-based single-pass raycaster for volume rendering deformed tetrahedral meshes. Thus, direct visualization of the inner structures of the deformed mesh is possible, while keeping the data on the graphics hardware throughout the entire simulation.

Key words: GPU computing, deformable bodies, implicit integration, physical simulation, volume rendering.
PACS: 83.20.jpg, 07.05.Tp, 02.70.Bf

1 Introduction

Physical-based animation of deformable objects has gained considerable attention in the last two decades among the computer graphics community [1–5]. The need for plausible real time animations has generated a number of approaches, most of them focused to the simulation of cloth [5–7] and facial

⋆ This work was partially suported by the German Academic Exchange Service with grant number A/04/08711.

* Corresponding author.
Email addresses: eduardo.tejada@vis.uni-stuttgart.de (Eduardo Tejada), thomas.ertl@vis.uni-stuttgart.de (Thomas Ertl).

expressions [8]. Application to surgical training or pre-operative planning of deformable models, when coupled with collision detection techniques, has also been adressed [9].

Works based on Finit Elemente Methods (FEM) have been reported [10,11], where objects are decomposed usually into tetrahedral linear elements, over which the elasticity equations are used. This leads to accurate simulations, whilst material properties are specified using few parameters. In this respect, it is important to note that modelling the deformable body as a continuum will not provide a plausible animation for non-homogeneous materials.

On the other hand, approaches based on mass-spring models [12,8,13] have had higher acceptance among the computer graphics community, due to their simplicity and due to the fact that FEM-based simulations are computationally more expensive than mass-spring based simulations. This becomes a main issue for animation purposes, where the need for plausible animations is more important than accurate simulation of physical phenomena.

Independent of the numerical method used, the formulation of the dynamics of deformable bodies is given by a partial differential equation which, once discretized, is integrated in time as an ordinary differential equation:

$$\ddot{\mathbf{x}} = \mathbf{M}^{-1}\left(-\frac{\delta E}{\delta \mathbf{x}} + \mathbf{f}^{(e)}\right) \tag{1}$$

where E is the internal energy of the body resulting from constraints, $\mathbf{f}^{(e)}$ describes external forces applied to the body, and $\mathbf{M}$ is a diagonal matrix representing the mass of the body.

The integration in time of this equation can be performed implicitly or explicitly. Explicit solvers have the advantage of being easy to implement and fast to compute [7]. However, this advantage is limited by the fact that the time step must decrease rapidly with stiffer equations in order to mantain stability. On the other hand, implicit integration is able to deal with stiff equations allowing the use of larger time steps and assuring stability where explicit methods fail [5].

The interactive simulation of stiff deformable objects is thus an unsolved problem. Interactive frame rates must be achieved in order to present useful feedback to the user. This problem is well suited for the application of the high programmability of current graphics hardware. The development of DirectX Pixel Shader 3.0 [14], and OpenGL `NV_fragment_program2` [15], increases the range of applications where the GPU could help to attain frame rates that otherwise could not be achieved.

We exploit the new features provided by graphics hardware to develop a GPU-based simulation system for deformable tetrahedral meshes. Our simulator performs implicit integration of the equation describing the dynamics of the physical system. This leads to interactive simulations since the high cost of integrating the differential equation implicity becomes irrelevant due to the considerable larger time steps we are able to use compared with explicit integration.

In order to perform the integration in time, we solve a linear system with an sparse non-banded matrix. Our arrangement of the data in textures fits nicely to the application of GPU-based linear systems solvers for sparse matrices [16,17]. Although the performance of these solvers were limited at that time by the need for multiple rendering passes to perform matrix algebra operations, the new looping capability offered by the `NV_fragment_program2` allows us to develop implementations with the performance required for our problem.

The new NVIDIA's `EXT_framebuffer_object` extension allows us to implement render-to-texture operations that overcomes the performance limitations of classical *pbuffer*- and `glCopyTexSubImage*`-based methods. To improve accuracy we use the full IEEE floating point precision provided by the NV40 architecture in both the shader and texture stages. In this respect it is important to know that accessing high precission textures slows down the performance about a factor of 1.5-2 [17].

For means of performance comparison, we implemented the explicit Euler, Verlet, and velocity Verlet integration methods on the GPU, as well as the CPU versions of the explicit and implicit Euler.

With the simulation performed on the GPU, rendering the results of the deformation is performed directly from the deformed data without the need for readbacks and downloads from/to the graphics hardware of the deformed mesh in every step of the simulation.

We exploit this fact to propose a single-pass raycaster for tetrahedral meshes, and extend it to deformable meshes using the recently supported NVIDIA's `NV_vertex_program3` and `NV_fragment_program2` extensions. This becomes an important result for medical applications where the deformation of volumetric models must be coupled with the visualization of inner structures.

Although the advantages of volume over surface rendering has extensively been proved during the last decades, to our knowledge no previous work treats both the volume deformation and visualization problems simultaneously. As we show in the results, the volumetric rendering of the deformed mesh provides a deeper insight, which is a main issue in a wide range of applications.

2 Related work

Müller et al. [11] presented a FEM-based approach for performing deformations in real time, by estimating the rotational part of the deformation at each node which, combined with linear elasticity, results in a plausible animation free of the disturbing artifacts present in linear models and faster than non-linear models. However, since a linear system must be solved for the implicit Euler integration, its use with large meshes is still limited.

Level-of-detail based approaches have been developed in order to accelerate the simulation process [18,6]. Debunne et al. [18] use an automatic space and time adaptive level-of-detail technique. The body is partitioned into a non-nested multiresolution hierarchy of tetrahedral meshes. High resolution meshes are then only used in regions with larger deformations. Birra and Santos [6] use a hierarchical subdivision of $4 - 8$ meshes triggered when the curvature error, calculated for each edge in each step, is greater than a specified threshold.

Teschner et al. [19] present an approach to deformable modeling that uses a low resolution tetrahedral mesh to perform deformations coupled with a high resolution surface mesh to visualize the deformed body. The authors claim that the actual deformation performed over the tetrahedral mesh is able to handle up to 25000 tetrahedra (considering only the numerical integration step) at interactive rates. Explicit Verlet integration is used to solve the Newton's equation of motion.

Physically-based simulation on the GPU has been addressed by researchers during the last years to simulate a variety of phenomena [20–23,17,16]. Several NVIDIA demos perform simple physical simulations modelling cloth, water, and particle systems physics using graphics hardware [20] based on the work by Harris et al. [21]. One of the major hardware limitations Harris et al. found was the need for using `glCopyTexSubImage2D` to perform render-to-texture, which involves a readback of the data from the graphics harware.

Harris et al. [22] present a physically-based cloud simulation. The implementation of the simulator is entirely done on floating-point graphics hardware. To solve the Poisson pressure equation they use a simple Jacobi solver.

Krüger and Westermann [17] present implementations of linear algebraic operators on programmable graphics processors. They demonstrate the effectiveness of the approach by implementing the Conjugate Gradient and Gauss-Seidel solvers for sparse matrices, and by applying these solvers to the 2D wave equation and the incompressible Navier-Stokes equations. Similarly Bolz et al. [16] present a GPU-based implementation of the Conjugate Gradient and Multigrid methods for sparse matrices and apply them to solve the incompressible Navier-Stokes equations. Goodnight et al. [24] present a multigrid solver for boundary value problems on the GPU, which differs from the previous works in that their implementation of the GPU algebraic operations are specific to the multigrid solver, which leads to an optimized code.

Although in these works it is shown that the implementation of the solvers is feasible, the limitations of the programmability of the graphics hardware at the time imposed certain restrictions and created some drawbacks. For instance, many passes are required just to compute the large vector inner products requiered by the algorithms ($O(log_2 N)$) [17,16]. Another problem is the need for pixels tests, e.g. occlusion queries [24]. One important issue that limits the speed of the computations is the need for context changes between *pbuffers* [16,24], which is a very costly operation, to perform render-to-texture.

The work by Georgii et al. [23] is of particular relevance to us. They address the simulation of deformable bodies at interactive rates through a GPU-based computation of the Verlet integration method over tetrahedral meshes, where the edges of the tetrahedra represent springs that join the particles (vertices). Although the frame rates reported show promising results, since their approach is based on explicit integration, instability arises for large time steps due to the stiffness of equations with high spring constants.

GPU-based volume rendering of tetrahedral meshes has been reported in the last years [25–27]. Pre-integration [28] is usually used in these works to improve the accuracy of the ray integral calculation. However, multi-dimensional transfer functions [29] have also shown very promising results.

Guthe et al. [25] developed a GPU-based approach for rendering of unstructured grids using the Projected Tetrahedra algorithm [30]. Multiple 2D textures are used to replace the 3D textures used by previous approaches to store the pre-integration table, in order to allow the use of high-resolution pre-integrated tables.

A hardware-based raycaster for tetrahedral meshes was proposed by Weiler et al. [26]. They use 3D and 2D textures to store the mesh and the current state of the integration process on the GPU. This approach was improved by the authors in [27] through the use of texture-encoded tetrahedral strip to decrease the amount of texture memory used. It also presents an approach to render non-convex meshes by means of multiple restarts of the rendering process at increasing depths. Both approaches use multiple rendering passes where a pass represents a step in the ray integration. This leads to multiple rasterizations of the primitives and render-to-texture operations which turns into a slow down of the render process. However, raycasting a volume provides results with higher accuracy than cell projection-based techniques.

3 Implicit Integration of the Physical Model

For a given tetrahedral mesh K, our physical model is based on the set of n interacting particles (vertices) τ_i in the mesh. Particle τ_i interacts with the $N(\tau_i)$ particles connected to it by an edge.

For sake of notation simplicity, we will refer in the future to particle τ_i as particle i, and x_i and v_i will represent its position and velocity.

Interaction is represented by a linear spring model, for which the energy function E for two particles i and j is given by:

$$E = \frac{1}{2}\kappa_s(|\mathbf{x}_{ij}| - L)^2 \tag{2}$$

where $\mathbf{x}_{ij} = \mathbf{x}_j - \mathbf{x}_i$, L is the original distance between $\mathbf{x}_i$ and $\mathbf{x}_j$, and κ_s is the spring constant. Thus, the force acting on particle i is:

$$\mathbf{f}_i^{(s)} = -\frac{\delta E}{\delta \mathbf{x}_i} = \kappa_s(|\mathbf{x}_{ij}| - L)\frac{\mathbf{x}_{ij}}{|\mathbf{x}_{ij}|} \tag{3}$$

We include damping forces exertet on particle i from the interaction with particle j as:

$$\mathbf{f}_i^{(d)} = -\kappa_d(\mathbf{v}_i - \mathbf{v}_j) \tag{4}$$

The combined force $\mathbf{f}_i$ over particle i is given by $\mathbf{f}_i = \mathbf{f}_i^{(s)} + \mathbf{f}_i^{(d)} + \mathbf{f}_i^{(e)}$, where $\mathbf{f}_i^{(e)}$ is the sum of external forces applied on particle i that do not depend on the position or the velocity of the particle.

The derivative of the force with respect to the position is the matrix given by:

$$\frac{\delta \mathbf{f}_i}{\delta \mathbf{x}_j} = \kappa_s \frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}} + \kappa_s\left(1 - \frac{L}{|\mathbf{x}_{ij}|}\right)\left(\mathbf{I} - \frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}}\right) \tag{5}$$

and with respect to the velocity is:

$$\frac{\delta \mathbf{f}_i}{\delta \mathbf{v}_j} = \kappa_d \mathbf{I}. \tag{6}$$

Arranging the forces, positions and velocities of all n particles in three arrays $\mathbf{f} = (\mathbf{f}_1, .., \mathbf{f}_n)$, $\mathbf{x} = (\mathbf{x}_1, .., \mathbf{x}_n)$, and $\mathbf{v} = (\mathbf{v}_1, .., \mathbf{v}_n)$, respectively, and given the $3n \times 3n$ diagonal matrix $\mathbf{M} = (m_1, m_1, m_1, \cdots, m_n, m_n, m_n)$, where m_i is the mass of particle i, our dynamical problem can be written in terms of the second-order differential equation:

$$\ddot{\mathbf{x}} = \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}) \tag{7}$$

Given the known position $\mathbf{x}(t)$ and velocity $\mathbf{v}(t)$ of the system at time t, the goal is to find the new position $\mathbf{x}(t+h)$ and the new velocity $\mathbf{v}(t+h)$ at time $t+h$, where h is the time step.

Defining $\mathbf{v} = \dot{\mathbf{x}}$, Equation 7 is converted to a first-order differential equation as:

$$\frac{d}{dt}\begin{pmatrix} \mathbf{x} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}, \mathbf{v}) \end{pmatrix} \tag{8}$$

Letting $\triangle\ \mathbf{x} = \mathbf{x}(t+h) - \mathbf{x}(t)$ and $\triangle\ \mathbf{v} = \mathbf{v}(t+h) - \mathbf{v}$, the implicit Euler method approximates $\triangle\ \mathbf{x}$ and $\triangle\ \mathbf{v}$ as:

$$\begin{pmatrix} \triangle\ \mathbf{x} \\ \triangle\ \mathbf{v} \end{pmatrix} = h \begin{pmatrix} \mathbf{v}(t) + \triangle\ \mathbf{v} \\ \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}(t) + \triangle\ \mathbf{x}, \mathbf{v}(t) + \triangle\ \mathbf{v}) \end{pmatrix} \tag{9}$$

Following the groundbreaking work by Baraff and Witkin [5], we solve Equation 9 by defining $\triangle\ \mathbf{x} = h(\mathbf{v}(t) + \triangle\ \mathbf{v})$ and then solving for $\triangle\ \mathbf{v}$ the linear system:

$$\left(\mathbf{I} - h\mathbf{M}^{-1}\frac{\delta\mathbf{f}}{\delta\mathbf{v}} - h^2\mathbf{M}^{-1}\frac{\delta\mathbf{f}}{\delta\mathbf{x}}\right) \triangle\ \mathbf{v} = h\mathbf{M}^{-1}\left(\mathbf{f}(t) + h\frac{\delta\mathbf{f}}{\delta\mathbf{x}}\mathbf{v}(t)\right) \tag{10}$$

formed using the derivatives given in Equations 5 and 6.

Thus, performing a time step using the implicit Euler translates into computing Equations 5 and 6, calculating the combined force $\mathbf{f}$, forming the linear system given by Equation 10, which is then solved for $\triangle\ \mathbf{v}$, and updating the positions and velocities in the states vectors $\mathbf{x}$ and $\mathbf{v}$.

In order to solve this linear system, we used a GPU implementation of the Conjugate Gradients [31] method, described in the next section as part of our GPU-based implicit solver.

4 Hardware-Accelerated Simulation

With the advent of Shader Model 3, new applications for general computing on graphics hardware have become possible. The dynamic control flow offered by the DirectX Pixel Shader 3 API and NVIDIAS's `NV_fragment_program2` allow the implementation of more complex algorithms on the GPU. This and the full IEEE floating point support in texture and shader stages provided by the NV40 architecture, advance the GPU to a more general purpose processing unit.

We exploit these new capabilities to realize, a complex physical-based deformation algorithm implemented on the GPU with minor operations performed on the CPU. As exposed before, our choice of an implicit solver to integrate the dynamics equation governing the deformation of the volume is based on the fact that implicit integration is able to handle time steps that provide interactive responses while working with stiff equations.

In order to solve our dynamic equation on the GPU, we store the information in 32 bits floating point textures downloaded to the graphics hardware. These textures hold the state information given by vectors $\mathbf{x}$ and $\mathbf{v}$, the externals forces given by $\mathbf{f}^{(e)}$, connectivity information between the interacting particles, and partial results obtained during the simulation.

To store the results coming from a GPU computation for later use, we use the `EXT_framebuffer_object` extension recently supported by NVIDIA's beta drivers to perform render-to-texture. Although currently the use of 32 bits floating point textures with this extension is not well supported by the driver implementation, rendering to textures using *framebuffer objects* outperforms the classic *pbuffer*-based approaches used nowadays. The higher accuracy obtained from the use of 32 bits floating point textures is also an important fact to take in consideration for physically-based simulations.

4.1 Data storage

We store the input information in five 2D textures, `positions`, `velocities`, `external_forces`, `neighbours`, and `neighbourhood`. These are square textures with dimensions given by $\lceil\sqrt{n}\rceil$, with the exception of the `neighbours` - texture which dimensions are $\lceil\sqrt{\sum_{i=1}^{n} N(\tau_i)}\rceil$, i.e. the sum of the number of neighbours of each particle. Better choices for the dimensions of these textures could be achieved using the results given in [16].

Texture `positions` holds the position for each particle in the RGB channels, while texture `velocities` holds the velocity and mass of each particle in the RGB and A channels respectively. In texture `external_forces`, the sum of the external forces over the particles is stored in the RGB channels. The connectivity information is stored in two separate textures. Texture `neighbourhood` stores in the RG channels the texture positions on textures `neighbours` where the information about the neighbours of the particle begins. In texture `neighbours`, $N(\tau_i)$ consecutive texels hold in the RG channels the texture positions of the neighbour in the first four textures, in the B channel the original distance between particle i and the neighbour and in channel A the index of particle i (See Figure 1). Additionally in the A channel of texture `neighbourhood`, $N(\tau_i)$ is stored in order to know the number of consecutive texels in `neighbours` that must be accessed.

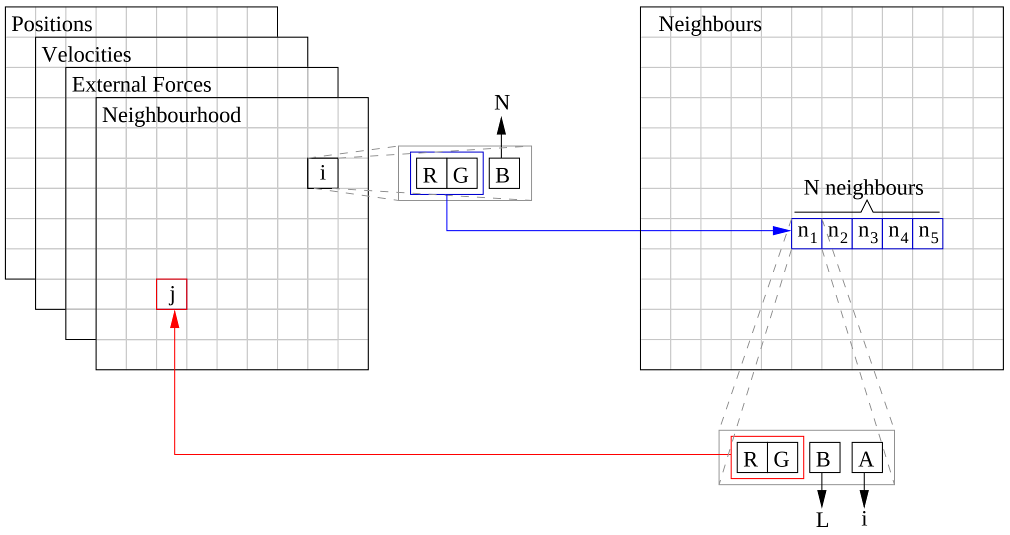

Fig. 1. Textures `neighbours` and `neighbourhood` hold the information of the neighboring particles.

This arrangement maps nicely to hold the sparse non-banded matrix computed when forming the linear system. We can think of each row of neighbours of particle i in texture `neighbours` as being the non-zero non-diagonal positions of the i-th row in the matrix. This holds, since only the elements of the matrix corresponding to two connected particles are non-zero.

It is important to keep in mind that the information in the first four textures for particle i is stored in the same texture position.

κ_s, κ_d, and h are passed to the graphics unit as environment paramenters. Alternatively the spring constant κ_s could be stored in the alpha channel of texture `neighbourhood`.

4.2 Simulation algorithm

A typical CPU implementation of the simulation algorithm is given in Figure 2. We describe here how each step of the algorithm is implemented on the GPU.

```
Simulation Algorithm
(1) Compute $\frac{\delta \mathbf{f}}{\delta \mathbf{x}}$ and $\frac{\delta \mathbf{f}}{\delta \mathbf{v}}$.
(2) Calculate the force vector $\mathbf{f} = \mathbf{f}^{(s)} + \mathbf{f}^{(d)} + \mathbf{f}^{(e)}$.
(3) Form the linear system given by Equation 10
(4) Solve the linear system for $\triangle \mathbf{v}$.
(5) Update the state vectors $\mathbf{x}$ and $\mathbf{v}$.
```

Fig. 2. Deformable models simulation algorithm with the implicit Euler integration method.

4.2.1 Compute derivatives and forces

Each step of the algorithm shown in Figure 2 is performed with a set of fragment programs. The first step is performed is two substeps: calculate the non-diagonal elements of $\frac{\delta \mathbf{f}}{\delta \mathbf{x}}$ and calculate the diagonal elements of $\frac{\delta \mathbf{f}}{\delta \mathbf{x}}$. $\frac{\delta \mathbf{f}}{\delta \mathbf{v}}$ needs no calculations in this step since it is defined as in Equation 6. We will elaborate on this later.

To calculate the non-diagonal elements of $\frac{\delta \mathbf{f}}{\delta \mathbf{x}}$ we render a quadrilateral covering a viewport of size $\lceil \sqrt{\sum_{i=1}^{n} N(\tau_i)} \rceil$. Each fragment represent a non-zero non-diagonal element in the sparse non-banded matrix of the linear system. Note that the last $(\lceil \sqrt{\sum_{i=1}^{n} N(\tau_i)} \rceil)^2 - \sum_{i=1}^{n} N(\tau_i)$ fragments are not used. Since we perform the simulation in the 3D space, each element of this matrix is a 3×3 matrix. Thus, the result of each fragment contains 9 elements. To do this, we render to three target buffers using the ATI's `GL_ATI_drawbuffers` extension. Each target buffer is bounded to a target texture `non_diagonal_dfdx`k, $k = 0, 1, 2$ attached to `GL_COLOR_ATTACHMENT`k`_EXT` of a *framebuffer object.*

The target texture `non_diagonal_dfdx`k will hold the results of the non-zero non-diagonal elements of the $(3 \times i + k)$-th row of the $\frac{\delta \mathbf{f}}{\delta \mathbf{x}}$ matrix, in the RGB channels of the texels corresponding to the neighbours of the i-th particle.

The fragment program takes as input the textures `positions`, `neighbours`, and `neighbourhood`, as well as the parameters κ_s and h. Each fragment computes the results according to Equation 5.

From Equation 5, we can see that each diagonal element of matrix $\frac{\delta \mathbf{f}}{\delta \mathbf{x}}$ is given by the negation of the sum of the non-diagonal elements in the row. Recall that since we are working in 3D space, each element of the diagonal is also a 3×3 matrix. Thus, in a further rendering pass we can compute the diagonal elements by rendering a quadrilateral covering a viewport of size $\lceil\sqrt{n}\rceil$, where n is the number of particles in the system, and summing in each fragment the results of the previous rendering pass corresponding to its neighbours. Thus textures `non_diagonal_dfdx`k are inputs to the current rendering pass. To access the information in textures `non_diagonal_dfdx`k we also need textures `neighbours` and `neighbourhood`.

Since in order to calculate the combined force over each particle we only need the result computed by the fragment corresponding to the particle, we include the calculation of the combined forces (Step 2 of the algorithm) in this rendering pass. In order to do this, from Equations 3 and 4, we need the input textures `positions`, `velocities`, and `external_forces`.

Fragment i calculates the 3×3 matrix corresponding to the i-th element of the diagonal, and the combined force corresponding to particle i. To loop on the neighbours (non-zero non-diagonal entries of the matrix in row i) we use the `LOOP` instruction available in the `NV_fragment_program2` extension. We implement a dynamic break using a counter and comparing it in each iteration with the number of neighbours. Although the `LOOP` instruction is limited to 255 iterations, this limitation could be overcomed using nested loops. However, from our experience with the tetrahedral meshes we used in the experiments, the number of neighbours a particle has never surpassed 20.

The result of this rendering pass is bounded to four target textures `forces` and `diagonal_dfdx`$k; k = 0, 1, 2$. As before, note that the last $(\lceil\sqrt{n}\rceil)^2 - n$ fragments are not used.

4.2.2 Form the linear system

In the third step of the algorithm, the linear system given by Equation 10 must be formed. To do this, we first compute in a rendering pass the right side $\mathbf{b} = h\mathbf{M}^{-1}\left(\mathbf{f}(t) + h\frac{\delta\mathbf{f}}{\delta\mathbf{x}}\mathbf{v}(t)\right)$ of the linear system. Inputs to this rendering pass are the textures `non_diagonal_dfdx`k, `diagonal_dfdx`k, and `forces`, as well as the parameter h and the textures `neighbours`, `neighbourhood`, and `velocities`, since it also holds the masses of the particles. We generate, as in the last step, $(\lceil\sqrt{n}\rceil)^2$ fragments, and each fragment calculates three elements of vector $\mathbf{b}$ to be stored in the RGB channels of texture `b`. Since matrix $\mathbf{M}^{-1}$ is diagonal, we only need to loop over the neighbours of the particle corresponding to the fragment i to calculate $\mathbf{f}(t) + h\frac{\delta\mathbf{f}}{\delta\mathbf{x}}\mathbf{v}(t)$ and multiply the result by $\frac{h}{m_i}$.

Next we need to compute $\mathbf{A} = \left(\mathbf{I} - h\mathbf{M}^{-1}\frac{\delta \mathbf{f}}{\delta \mathbf{v}} - h^2\mathbf{M}^{-1}\frac{\delta \mathbf{f}}{\delta \mathbf{x}}\right)$. Since $\mathbf{I}$ is a diagonal matrix, the non-diagonal elements of matrix $\mathbf{A}$ are given by $-h\mathbf{M}^{-1}\frac{\delta \mathbf{f}}{\delta \mathbf{v}} - h^2\mathbf{M}^{-1}\frac{\delta \mathbf{f}}{\delta \mathbf{x}}$. We generate $(\lceil\sqrt{\sum_{i=1}^{n} N(\tau_i)}\rceil)^2$ fragments. Each fragment multiplies the input from the textures `non_diagonal_dfdx`k by $-\frac{h^2}{m_i}$, to obtain a partial result. Note that we need the index i of the corresponding particle (row), which we fetch from the A channel of texture `neighbours`. Since $\frac{\delta \mathbf{f}_i}{\delta \mathbf{v}_j} = \kappa_d \mathbf{I}$ is a 3×3 diagonal matrix, we add $-\frac{\kappa_d h}{m_i}$ to the diagonal elements of the partial result, and write this final result to the RGB channels of textures `non_diagonal_A`k; $k = 0, 1, 2$.

To compute the diagonal elements, we generate $(\lceil\sqrt{n}\rceil)^2$ fragments, providing as input textures `diagonal_dfdx`k, and multiply their content by $-\frac{h^2}{m_i}$ for fragment i. We add $1 - \frac{\kappa_d h N(\tau_i)}{m_i}$ to the diagonal elements and then write the result to textures `diagonal_A`k; $k = 0, 1, 2$. Note that we also need the texture holding the masses of each particle (`velocities`) and the texture with the number of neighbours for each particle (`neighbourhood`) as input.

With textures `diagonal_A`k, `non_diagonal_A`k, and `b` holding the matrix $\mathbf{A}$ and vector $\mathbf{b}$ respectively, our problem fits nicely to the GPU-based implementation of the Conjugate Gradients algorithm proposed in the work by Krüger et al. [17].

4.2.3 Solve the linear system

Our arrangement of the data in the textures differs from the one proposed by Krüger and is more similar to the one proposed by Bolz et al. [16]. Thus we fitted the matrix operations used by the Unpreconditioned GPU-based CG presented by Krüger to work with our arrangement. One of the major differences in the implementation of the operators proposed in both works is the reduction operator. We exploited the availability of loops in `NV_fragment_program2`, to implement a two pass reduction operator, compared to the $\log N$ passes needed by Krüger and Bolz. In the first pass we perform a reduction by a factor of 255. This result is then combined in a second pass. Thus, we can handle vectors of up to 255^2 elements. Further passes or nested `LOOP` instructions would be needed for larger vectors. The tradeoff between the number of passes and the use of the 16 pipelines available in the fragment shader stage, must be determined in a future work.

Matrix-vector multiplication operations also benefits from the looping capabilities of `NV_fragment_program2`. We eliminate the need for multiple passes looping on the non-zero elements of each row and using the information stored on the alpha channel of texture `neighbours` to access the corresponding element in the vector to be multiplied by the matrix. Each fragment generated (one fragment per element in the resulting vector) performs this operation and then adds the result to the contribution of the diagonal element.

This and the fact that we use *framebuffer objects* to perform render-to-texture, increases the performance of the algebraic matrix operations. This allows the application of this kind of operators in real situations where the size of the linear system makes the computation of its solution on the CPU unfeasible for interactive applications.

4.2.4 Update the state vectors

Once we solve the linear system, the solution vector $\triangle \mathbf{v}$ is used as input to a final rendering pass, where n fragments are generated to update the position and velocity of each particle using the update rules given in Section 3. The result is rendered to textures `positions` and `velocities`, and then the next iteration of the simulation is started.

4.3 Rendering the deformed mesh

Since it is not possible to bind a memory object, such as vertex arrays, using *framebuffer objects*, we use the capability of `NV_vertex_program3` to access textures, in order to be able to use the deformed vertex positions. We access texture `positions` in the vertex program, using texture coordinates passed to each vertex with `glTexCoord2f`, and fetch the correct vertex position, which is then transformed and passed as output to the rasterizer. This allows us to avoid costly readback operations from the graphics hardware.

Figure 3 shows deformations performed on different tetrahedral meshes using our approach [1] . The images were generated using a surface renderer. However, in contrast to previous works, we focused on applications where the visualization of the inner information of the deformed tetrahedral mesh is a key point.

However, rendering deformed tetrahedral meshes requieres certain considerations not addresed in previous GPU-based volume rendering algorithms available in the literature. In the next section we present a single-pass raycaster for tetrahedral meshes, and describe the modifications needed to support rendering of deformable meshes.

[1] Details on the meshes are given in Section 6.

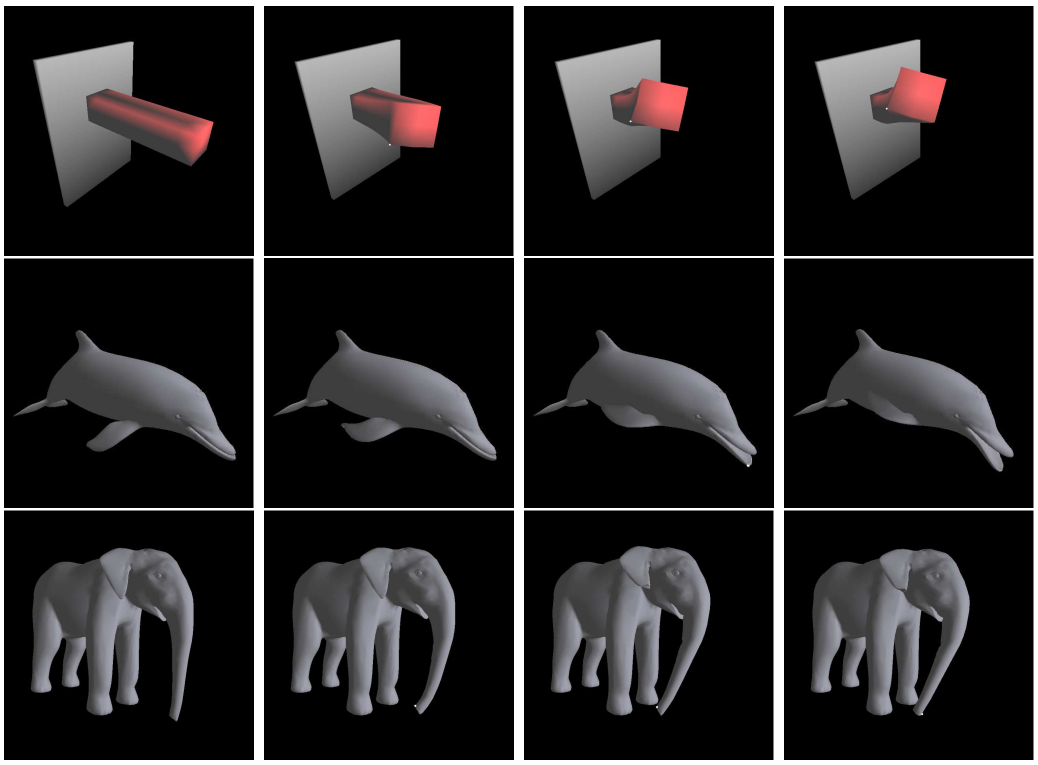
Fig. 3. Surface rendering of deformed tetrahedral meshes.

5 GPU-based Volume Rendering of Deformable Tetrahedral Meshes

In this section we present a single-pass raycaster for tetrahedral meshes implemented using the most recent extensions available for commodity graphics hardware. We also show the modifications performed on it to support deformable meshes.

5.1 Single-pass raycasting of tetrahedral meshes

To perform raycasting on the deformed tetrahedral mesh we need, besides the results of the simulation process that are already stored on texture `positions`, topology information which we store in two 3D textures `tetra_vertices` and `tetra_neighbours`.

The texture `tetra_vertices` holds the texture coordinates on the texture `positions` of the vertices $v_{t,i}; i = 0, 1, 2, 3$ of tetrahedron t in the RG channels, whilst texture `tetra_neighbours` stores the texture coordinates on the textures `tetra_*` of the four neighbouring tetrahedra $ne_{t,i}$ of each tetrahedron t. These texture coordinates are also stored in the RG channels. The z offset in both textures represents the four vertices/neighbouring tetrahedra. The B channel of `tetra_vertices` is used to store the scalar value $s_{t,i}$

attached to each vertex $v_{t,i}$ of the tetrahedron t, whilst the B channel of tetra_neighbours holds the local index $f_{t,i}; 0 \leq f_{t,i} \leq 3$ of the faces of the tetrahedron. Face $f_{t,i}$ is opposite to vertex $v_{t,i}$. We will denote by $\mathbf{v}_{t,i}$ the position in space of vertex $v_{t,i}$ and by $\mathbf{n}_{t,i}$ the normal of face $f_{t,i}$.

The basic implementation of the raytracer on the GPU begins with the rendering of the front faces of the volume to generate the fragments that will represent the casted rays. Using per-fragment interpolated attributes we pass to a fragment program the direction of the casted ray, the texture coordinates in textures tetra_* of the entry tetrahedron, and the local index of the entry face.

The ray traversal is implemented in previous works using multiple passes, calculating in each pass the contribution of the current tetrahedron, finding the new tetrahedron intersected by the casted ray, and writing the partial results and states to working textures. Early ray termination was implemented by means of an additional pass and performing pixel tests, e.g occlusion queries.

The advent of the NV_fragment_program2 with the support to full braching inside a fragment program, eliminates the need for multiple passes. The raycasting algorithm fits nicely to the new looping capabilities. The limit of 255 loops within a LOOP or REP instruction is easily overcome using nested loops. Avoiding the render-to-texture operations and multiple rasterizations needed to implement the multipass raycaster, increases the number of frames per second obtained. Early ray termination is also easily included into the implementation by means of a BRK instruction. Thus ray traversal is implemented stepping from tetrahedron to tetrahedron in the direction of the casted ray.

Normal and gradient information is necessary during ray traversal in order to compute intersections of the ray with the faces of the tetrahedra and to interpolate the scalar value at any point within the tetrahedra. This information is stored in five 2D textures tetra_gradients and tetra_normals$i; i = 0, 1, 2, 3$ of the same size as textures tetra_vertices and tetra_neighbours.

Textures tetra_normals$i$, store the normals $\mathbf{n}_{t,i}$ for each face $f_{t,i}$, whilst texture tetra_gradients holds the gradient information used to find the scalar value at any point within a tetrahedron from the scalar values stored at the vertices. Using barycentric interpolation, where the interpolation weights are given as normalized distances from the opposite faces, the scalar value $s(\mathbf{x})$ at any point $\mathbf{x}$ within a tetrahedron t is calculated as:

$$s(\mathbf{x}) = \sum_{i=0}^{3} w_i s_{t,i}, \; w_i = \frac{\mathbf{n}_{t,i} \cdot (\mathbf{x} - \mathbf{v}_{t,3-i})}{\mathbf{n}_{t,i} \cdot (\mathbf{v}_{t,i} - \mathbf{v}_{t,3-i})} \tag{11}$$

However, given the constant gradient $\mathbf{g}_t$ within a tetrahedron, we can calculate

$s(\mathbf{x})$ as:

$$s(\mathbf{x}) = \mathbf{g}_t \cdot (\mathbf{x} - \mathbf{v}_{t,0}) + s_{t,0} = \mathbf{g}_t \cdot \mathbf{x} - \mathbf{g}_t \cdot \mathbf{v}_{t,0} + s_{t,0} \tag{12}$$

which means that the interpolation of a scalar at any point within a tetrahedron can be computed with the dot product of the coordinates of the point and the gradient $\mathbf{g}_t$ plus a constant $\hat{g}_t = -\mathbf{g}_t \cdot \mathbf{v}_{t,0} + s_{t,0}$. Therefore, we store $\mathbf{g}_t$ and $\hat{g}_t$ in the texture `tetra_gradients`.

During ray traversal we calculate the exit point of the ray for the current tetrahedron as described in [26]. Given the eye point $\mathbf{e}$, and the normalized direction $\mathbf{r}$ of the ray, we determine the exit face taking the smaller $\lambda_{t,i} = \frac{(\mathbf{v}_{t,3-i} - \mathbf{e}) \cdot \mathbf{n}_{t,i}}{\mathbf{r} \cdot \mathbf{n}_{t,i}}$ of the non-visible faces of the tetrahedron. A face is visible if $\mathbf{r} \cdot \mathbf{n}_{t,i} < 0$. Given the smaller $\lambda_{t,i}$, the exit face will be $f_{t,i}$, and the exit point $\mathbf{x}$ is computed as $\mathbf{x} = \mathrm{e} + \lambda_{t,i}\mathbf{r}$. Point $\mathbf{x}$ becomes the entry point for the next step in the ray traversal.

Given the entry and exit points, we calculate the scalar values at those points as in Equation 12, and the distance d between them. These three parameters are then used to perform a lookup in a 3D pre-integrated table [28], calculated using incremental pre-integration as in the work by Weiler et al. [26]. The associated color $\widetilde{C}_k$ and opacity α_k resulting from the lookup are then used to accumulate the contribution of the tetrahedron to the ray integral as

$$\begin{aligned} \widetilde{C}'_k &= \widetilde{C}'_{k-1} + (1 - \alpha'_{k-1})\widetilde{C}_k \\ \alpha'_k &= \alpha'_{k-1} + (1 - \alpha_{k-1})\alpha_k \end{aligned} \tag{13}$$

obtaining the approximations $\widetilde{C}'_k$ and α'_k of the ray integral after k iterations.

5.2 Rendering deformable meshes

For deformable meshes we cannot assure the convexity of the tetrahedral mesh, or the validity of the normals and gradients stored in the textures at the beginning of the simulation. Therefore, we modified the raycasting algorithm to support non-convex meshes, and introduced two rendering passes after each simulation step, before the actual raycasting algorithm is used to render the deformed mesh, one for updating the normals and one for updating the gradients.

To update the normals, we render a polygon generating one fragment per tetrahedron, and use the textures `positions` and `tetra_vertices` as inputs. In the fragment program, we access the positions $\mathbf{v}_{t,i}$ in the texture `positions` of each vertex $v_{t,i}$ of tetrahedron t, using the information stored in `tetra_vertices`. We then compute the new normals for each face and write the results in four render buffers, previously attached to textures `tetra_normals`i. We do not use a 3D texture of depth 4, as for the vertices, since the current driver implementation of the `EXT_framebuffer_object` does not support yet the use of 3D textures. However, the document describing the extension includes this functionality as part of the specification. Thus, we expect this support in final releases of the driver.

Once the new normals are computed, we update the gradients in a further rendering pass, generating a fragment for each tetrahedron. Input to the fragment program are the textures `tetra_normals`, `tetra_vertices`, and `positions`. In each fragment the gradient of the corresponding tetrahedron is calculated as:

$$\mathbf{g}_t = \sum_{i=0}^{3} \frac{s_{t,i}}{\mathbf{n}_{t,i} \cdot (\mathbf{v}_{t,i} - \mathbf{v}_{t,3-i})} \mathbf{n}_{t,i}. \tag{14}$$

and then we calculate $\hat{g}_t = -\mathbf{g}_t \cdot \mathbf{v}_{t,0} + s_{t,0}$ and write both results to texture `tetra_gradients`.

To handle non-convex meshes we use an idea similar to the one presented by Weiler et al. [27] for non-convex tetrahedral strips. By means of multiple passes of the raycasting algorithm, we accumulate the contribution of each "section" of the volume intersected by a casted ray. In the first pass we only take the result of the fragment of the closest face intersected by the ray (through the use of depth tests) and write it to a texture. We store the depth of the entry fragment and in the next pass we compare it with the depth of the current fragment. If the depth is less or equal than the stored one, we set the fragment depth to $1 - depth/10$, where *depth* is the original normalized depth. If the fragment depth is greater than the one stored, we blend the accumulated color and opacity of the previous pass with the result of the ray traversal performed by the fragment and write the new depth for further use in the next pass. Note that we assume that no entry point will be farther away than 0.9 (in normalized space).

6 Results and Discussion

In this section we present results regarding the performance of the algorithms implemented, both on CPU and GPU, when used with different tetrahedral meshes. All performance measurements were carried out on a standard PC equipped with an NVIDIA GeForce 6800 Ultra graphics board, a 3.8GHz P4 CPU, and 2GB RAM.

Table 1 shows the comparative results of the GPU-implementation of the implicit and explicit Euler, as well as the Verlet and velocity Verlet solvers. Computation time for the *framebuffer object* based and `glCopyTexSubImage*` based implementations are given in ms. As we can see, despite the unoptimized implementation of the `EXT_framebuffer_object` extension, using *framebuffer objects* reports better results in almost all cases. As stated before, a fully optimized support of *framebuffer objects* is expected in a final realese of NVIDIA's drivers. Similar support from other vendors is also expected in the near future.

	Mesh size	Computation time [ms] (fbo/`glCopyTexSubImage*`)			
	Tetra./Vert.	Expl. Euler	Verlet	Veloc. Verlet	Impl. Euler
Bar	80/45	0.17/0.96	0.17/1.18	0.18/1.19	3.46/54.25
Machine	4996/1132	0.16/1.46	0.16/1.96	0.16/1.96	2.85/93.40
Dolphin	13766/3990	0.17/3.33	0.17/3.99	0.17/3.99	17.23/14.05
Panda	17312/5199	0.16/4.20	0.16/5.01	0.16/5.01	18.52/18.62
Elephant	24106/6829	0.16/4.81	0.15/5.62	0.15/5.62	29.06/214.07
Knee	112299/19896	0.17/28.94	0.17/29.87	0.17/29.87	104.82/102.36
Foot	156612/24686	0.14/41.43	0.17/41.48	0.17/41.55	40.76/62.22

Table 1
Comparison between the implicit Euler solver and explicit methods implemented on the GPU. Time is given in ms. for the implementation based on *framebuffer objects* and `glCopyTexSubImage*` respectively.

It is important to remark that with the explicit integration methods, we were able to use, for a string constant of $\kappa_s = 3000$, a maximum time step of $h = 0.001s$. With higher string constants or larger time steps, instability occurred. On the other hand, with the implicit Euler and a time step of $h = 0.01s$ we found no stability problem using constants greater than $\kappa_s = 30000$. In order to simulate stiffer systems with the explicit methods, we would have to reduce the time step to impractical values. On the other hand increasing the time step in the implicit Euler would lead to faster simulations than explicit methods for the same string constant.

In Table 2 we present the details of the three major steps of the GPU implementation of the implicit Euler, namely the forming of the linear system, the Conjugate Gradients algorithm, and the update of $\mathbf{x}$ and $\mathbf{v}$. We can see that our implementation of the Conjugate Gradients is able to solve a linear system $\mathbf{Ax} = \mathbf{b}$, for a matrix A of size 24686^2, in 34.59 ms (28.91 fps.).

The total time is reproduced again in this table for means of comparison with the explicit and implicit Euler implemented on the CPU. In some cases, the CPU implementation of the implicit Euler was not able to solve the equation

	GPU implicit Euler				CPU solvers	
	Linear sys.	Conj. Grad.	Update	Total	Expl. Euler	Impl. Euler
Bar	0.72	2.63	0.11	3.46	0.71	91.37
Machine	0.71	2.03	0.11	2.85	28.26	47886.2
Dolphin	0.73	15.39	0.11	17.23	90.13	*
Panda	0.84	17.57	0.11	18.52	118.28	*
Elephant	1.10	27.85	0.11	29.06	159.85	*
Knee	11.66	93.05	0.11	104.82	4271.77	*
Foot	6.14	34.59	0.13	40.76	5839.39	*

* maximum response time exceeded.

Table 2
Computation time in detail of the GPU-based implementation with *framebuffer objects* of the implicit Euler compared to CPU versions of the explicit and implicit Euler. Time is given in ms.

before the program was aborted due to exceeded time without response. In debugging mode, the program solved the linear system sucessfully, but the processing time was considerably increased by the debugging operations. Thus, we opted for not presenting those biased results here.

	Update gradients and normals		Raycasting	
	FBO	`glCopyTexSubImage*`	FBO	`glCopyTexSubImage*`
Bar	0.16	1.70	0.32	0.22
Machine	0.16	62.93	1.19	0.65
Dolphin	0.16	172.10	99.23	1971.92
Panda	0.16	215.98	68.17	862.73
Elephant	0.16	300.45	323.22	6369.22
Knee	0.16	2015.73	398.04	7002.56
Foot	0.16	1943.93	0.84	1.12

Table 3
Computation time in ms. of the gradients and normals update and the raycasting processes, using *framebuffer objects* and `glCopyTexSubImage*`.

Table 3 shows the computation times in ms. of the update gradients and normals process and the raycasting. We see a decrease in the processing time for the Foot dataset, although the data set contains more than 150000 tetrahedra. This is due to the fact that the data set is thin and elongated. Thus viewing the tetrahedra in the direction of the smaller dimension, leads to fewer loops in the fragment program; i.e., fewer tetrahedra are intersected by each ray.

For our two largest datasets, Foot (156612 tetrahedra) and Knee (112299 tetrahedra), we obtained frames rates of 1190.5 fps. and 2.51 fps. respectively, whilst Weiler et al. [26] obtained for meshes of similar sizes (148955 and 124152 tetrahedra) frames rates of 5.13 fps. and 2.27 fps. respectively. Thus, we can state that our results outperform those presented by Weiler et al.

Comparing our results of the simulation processing time, with those reported by Georgii et al. [23], we find that our implementations of the explicit methods outperform their implementation of the Verlet solver. For a mesh with 84104 tetrahedra they attain a performance of 121 fps., whilst even with the use of the quite more expensive raytracing (compared to surface rendering algorithms), we obtain a total processing time of 1.17 ms, meaning 877.20 fps., for the Foot dataset (156612 tetrahedra).

The largest time step they were able to use is $h = 0.004$. As we discussed before, the implicit method could achieve similar simulation times, by setting larger time steps, while ensuring stability. Comparing our results of the implicit Euler for the dataset Foot containing 156612 tetrahedra (40.76 ms. of integration, gradients and normal update, and raycasting time) with the results of the largest mesh Georgii et al. used, containing 84104 tetrahedra (8.26 ms. of integration and rendering time), we could state that our approach outperforms the simulation time of their solver, when using a time step of $h = 0.04$, in a factor of 2.03. However, the main advantage of our solver over theirs is the stability of the simulation for stiff equations.

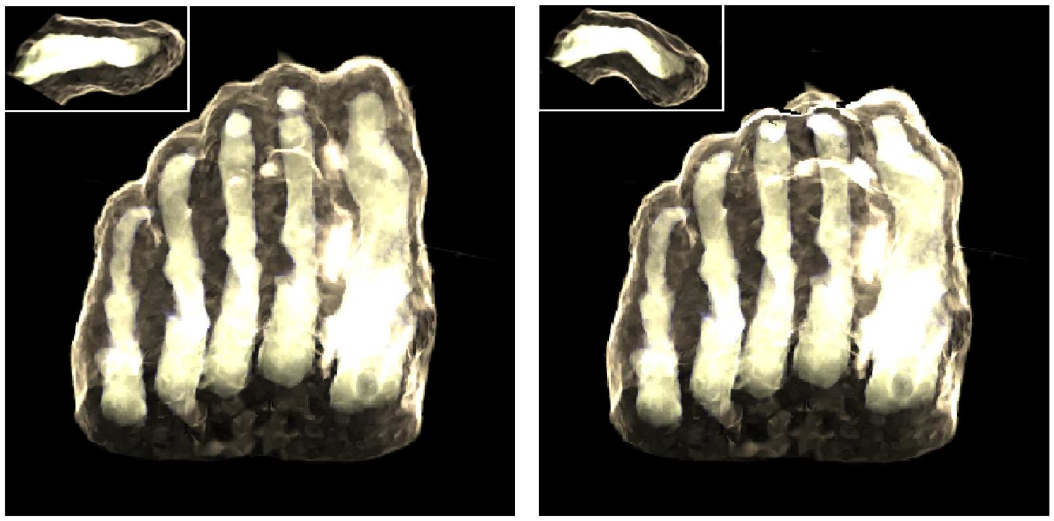

Fig. 4. Original and deformed mesh of the Foot dataset. A detail on a finger is shown in the upper link corner of each image.

Figures 4 and 5 show volumetric renderings of the Knee and Foot datasets for the original and a deformed state. The insight gained from the volume rendering of the deformed mesh is an important result of our work. The application of this integrated deformation-visualization simulator to medicine would bring new possibilities for the pre-operative planning and surgical training.

In these figures we can appreciate the deformation of the inner structures in both tetrahedral meshes. As we can see, the images generated are of high quality, considering the rough fitting of the tetrahedral meshes to the volume data.

To perform the deformations shown in Figures 4 and 5, we used a uniform κ_s constant over the entire meshes. However, local material properties could be adjusted as stated in Section 4, storing in texture `neighbourhood` a specific constant for each particle. Right now, we are incorporating a transfer-function-driven mechanismus to specify the parameter κ_s for each particle by means of a lookup performed using the intensity values associated to the vertices.

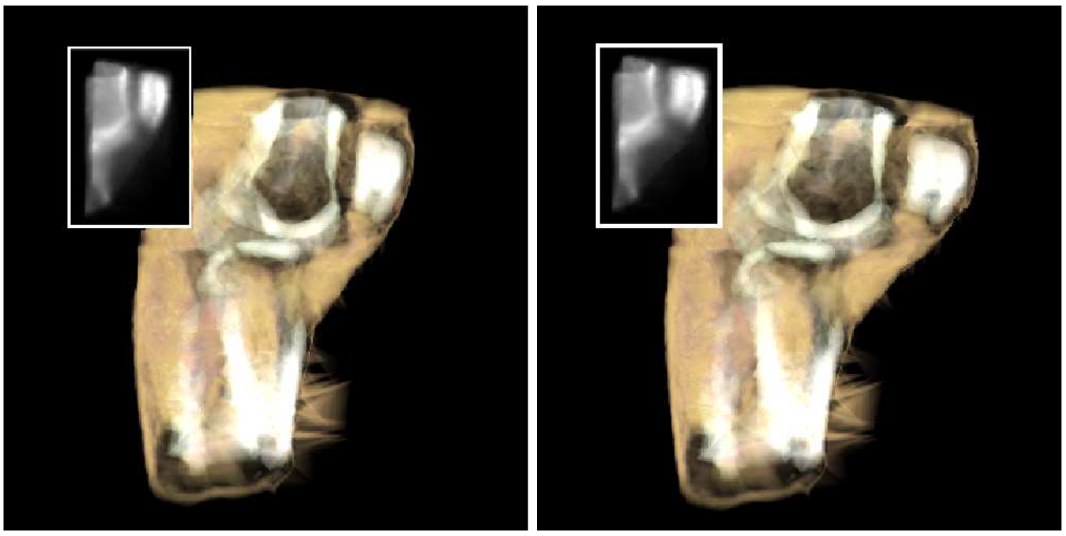

Fig. 5. Original and deformed mesh of the Knee dataset.

7 Conclusion

We have presented a GPU-based implementation of a simulation system for deformable bodies, exploiting the new general purpose programming capabilities of graphics hardware. Our approach integrates the simulation and visualization stages for deformable tetrahedral volumes, focussing on applications where the visualization of inner information, stored as intensity values at the vertices of the mesh, is a key issue.

Since stability and response at interactive frame rates are main goals of many applications, ranging from computer animation to medicine, our concern was to develop a stable yet fast simulator. The fact that our simulator is based on an implicit solver, ensures the stability of the simulation process, whilst the application of the most recent extensions available for commodity graphics hardware allowed us to achive the high performance required to present useful feedback to the user.

We described the implementation of the implicit solver in detail and proposed a single-pass raycaster. We also presented the modifications that must be performed on the raycaster in order to support the rendering of deformable volumes direct from the results of each simulation step.

Our tests show satisfactory performance results for meshes of even hundreds of thousands of tetrahedra, and the rendering quality demonstrated by our raycaster is the same as the one achieved with previous multi-pass implementations for tetrahedral meshes, whilst offering higher interactive frame rates.

Acknowledgmentes

We would like to thank Alex Cuadros-Vargas, University of Sao Paulo, Brazil, for providing us with the tetrahedral meshes of the Foot and Knee datasets.

References

[1] S. F. F. Gibson, B. Mirtich, A survey of deformable modeling in computer graphics, Tech. rep. (1997).

[2] D. Terzopoulos, J. Platt, A. Barr, K. Fleischer, Elastically deformable models, Computer Graphics 21 (4) (1987) 205–214.

[3] D. Terzopoulos, K. Fleischer, Deformable models, The Visual Computer 4 (1988) 306–331.

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

[5] D. Baraff, A. Witkin, Large steps in cloth simulation, in: SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 1998, pp. 43–54.

[6] F. P. Birra, M. P. dos Santos, Stable cloth animation with adaptive level of detail, in: WSCG (Short Papers), 2004, pp. 31–38.

[7] P. Volino, N. Magnenat-Thalmann, Comparing efficiency of integration methods for cloth simulation, in: Proceedings of Computer Graphics International, Hong Kong, 2001, pp. 265–274.

[8] Y. Lee, D. Terzopoulos, K. Walters, Realistic modeling for facial animation, in: SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 1995, pp. 55–62.

[9] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, W. Strasser, P. Volino, Collision detection for deformable objects, Computer Graphics Forum 24 (1) (2005) 61–81.

[10] M. Müller, L. McMillan, J. Dorsey, R. Jagnow, Real-time simulation of deformation and fracture of stiff materials, in: Proceedings of the Eurographic workshop on Computer animation and simulation, Springer-Verlag New York, Inc., New York, NY, USA, 2001, pp. 113–124.

[11] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, B. Cutler, Stable real-time deformations, in: SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation, ACM Press, New York, NY, USA, 2002, pp. 49–54.

[12] D. Terzopoulos, J. Platt, K. Fleischer, From glopp to glop: Heating and melting deformable objects, in: Proceedings of Graphics Interface, 1989.

[13] M. Desbrun, P. Schröder, A. Barr, Interactive animation of structured deformable objects, in: Proceedings of the 1999 conference on Graphics interface '99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999, pp. 1–8.

[14] Microsoft Corporation. DirectX 9 SDK, http://www.microsoft.com/directx.

[15] NVIDIA OpenGL Extension Specifications for the CineFX 3.0 Architecture (NV4x), http://developer.nvidia.com/object/nvidia_opengl_spect.html.

[16] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, Sparse matrix solvers on the gpu: conjugate gradients and multigrid, ACM Transactions on Graphcis 22 (3) (2003) 917–924.

[17] J. Krüger, R. Westermann, Linear algebra operators for gpu implementation of numerical algorithms, ACM Transactions on Graphcis 22 (3) (2003) 908–916.

[18] G. Debunne, M. Desbrun, M.-P. Cani, A. H. Barr, Dynamic real-time deformations using space & time adaptive sampling, in: SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 2001, pp. 31–36.

[19] M. Teschner, B. Heidelberger, M. Müller, M. Gross, A versatile and robust model for geometrically complex deformable solids, in: CGI '04: Proceedings of the Computer Graphics International (CGI'04), IEEE Computer Society, Washington, DC, USA, 2004, pp. 312–319.

[20] M. J. Harris, G. James, Physically-based simulation on graphics hardware (2003).
URL
`http://developer.nvidia.com/docs/IO/8230/GDC2003_PhysSimOnGPUs.pdf`

[21] M. J. Harris, G. Coombe, T. Scheuermann, A. Lastra, Physically-based visual simulation on graphics hardware, in: HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2002, pp. 109–118.

[22] M. J. Harris, W. V. Baxter, T. Scheuermann, A. Lastra, Simulation of cloud dynamics on graphics hardware, in: HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003, pp. 92–101.

[23] J. Georgii, F. Echtler, R. Westermann, Interactive simulation of deformable bodies on gpus, in: Simulation and Visualisation 2005, 2005.

[24] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys, A multigrid solver for boundary value problems using programmable graphics hardware, in: HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003, pp. 102–111.

[25] S. Guthe, S. Roettger, A. Schieber, W. Strasser, T. Ertl, High-Quality Unstructured Volume Rendering on the PC Platform, in: Proc. EG/SIGGRAPH Graphics Hardware Workshop '02, 2002, pp. 119–125.

[26] M. Weiler, M. Kraus, M. Merz, T. Ertl, Hardware-Based Ray Casting for Tetrahedral Meshes, in: Procceedings of IEEE Visualization '03, IEEE, 2003, pp. 333–340.

[27] M. Weiler, P. N. Mallón, M. Kraus, T. Ertl, Texture-Encoded Tetrahedral Strips, in: Proceedings Symposium on Volume Visualization 2004, IEEE, 2004, pp. 71–78.

[28] K. Engel, M. Kraus, T. Ertl, High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading, in: Eurographics / SIGGRAPH Workshop on Graphics Hardware '01, Annual Conference Series, Addison-Wesley Publishing Company, Inc., 2001, pp. 9–16.

[29] J. Kniss, G. Kindlmann, C. Hansen, Multidimensional transfer functions for interactive volume rendering, IEEE Transactions on Visualization and Computer Graphics 8 (3) (2002) 270–285.

[30] P. Shirley, A. Tuchman, A polygonal approximation to direct scalar volume rendering, in: VVS '90: Proceedings of the 1990 workshop on Volume visualization, ACM Press, New York, NY, USA, 1990, pp. 63–70.

[31] J. R. Shewchuk, An introduction to the conjugate gradient method without the agonizing pain, Tech. rep., Pittsburgh, PA, USA (1994).