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 by means of a GPU-based implicit solver and present a hardware-based single-pass raycaster for 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

Physically-based animation of deformable objects has gained considerable attention in the last two decades among the computer graphics community [1–10]. The need for plausible real-time animations has generated a number of approaches, most of them focused on the simulation of cloth [4] and facial expressions [5]. Works based on Finite Element Methods (FEM) have been reported [6,7], where objects are decomposed usually into tetrahedral elements,

⋆ This work was partially supported 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).

over which the elasticity equations are used. This leads to accurate simulations, whilst material properties are specified using few parameters. However, mass-spring models [5,8] have had higher acceptance due to their simplicity and the fact that they are computationally less expensive.

Independent of the model used, the dynamics of deformable bodies is governed by a partial differential equation which, once discretized, is integrated in time as an ordinary differential equation (Section 3), using explicit or implicit solvers. Explicit solvers are easy to implement and fast to compute [9]. However, this advantage is limited by the relative small time steps needed to maintain stability with stiff equations. On the other hand, implicit integration assures stability with larger time steps where explicit methods fail [4].

However, the interactive simulation of stiff deformable objects is an unsolved problem. This problem is well suited for the application of the high programmability of current graphics hardware, recently increased by the dynamic control flow offered by DirectX Pixel Shader 3.0 [11] and OpenGL `NV_fragment_program2` [12], and the full IEEE floating point support in texture and shader stages provided by the NV40 architecture.

We exploit these new features to develop a hardware-accelerated simulation system for deformable tetrahedral meshes based on implicit integration. With the simulation performed on the GPU, rendering the deformed body can be realized directly without the need for readbacks and downloads from/to the graphics hardware in every step of the simulation. We exploit this fact to also propose a single-pass raycaster for deformable tetrahedral meshes. For means of performance comparison we also implemented different explicit solvers on the GPU.

2 Related work

Müller et al. [7] presented a FEM-based approach for real-time deformations. By estimating the rotational part of the deformation and using linear elasticity, they create plausible animations free of the disturbing artifacts present in linear models and faster than non-linear models. However, since a they solve a linear system on the CPU for the implicit integration, its use with large meshes is still limited.

Teschner et al. [10] perform deformations on low resolution tetrahedral meshes, coupled with high resolution surface meshes used to visualize the deformed body. Explicit Verlet integration on the CPU is used to solve Newton's equation of motion. The actual deformation process is able to handle up to 25000 tetrahedra at interactive rates.

Physically-based simulation on the GPU has been addressed by researchers

during the last years to simulate a variety of phenomena [13–17]. Several NVIDIA demos perform simple physical simulations modelling cloth, water, and particle systems physics using graphics hardware [13] based on the work by Harris et al. [14].

Krüger and Westermann [16] present a set of linear algebraic operators implemented on the GPU. The effectiveness of the approach is demonstrated by implementing the Conjugate Gradient and Gauss-Seidel algorithms to solve the 2D wave and incompressible Navier-Stokes equations. Similarly Bolz et al. [17] implement the Conjugate Gradient and Multigrid methods on the GPU and apply them to solve the incompressible Navier-Stokes equations. A GPU-based Multigrid solver is also presented by Goodnight et al. [18]. In these works many passes are required to compute the large vector inner products required by the algorithms ($O(log_2 N)$) [16,17]. Also, the need for context changes between *pbuffers* [17,18], to perform render-to-texture, and for pixels tests, e.g. occlusion queries [18], limits the performance of these approaches.

The work by Georgii et al. [15] is of particular relevance to us. They address the simulation of mass-spring models 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. 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.

Weiler et al. proposed a hardware-accelerated raycaster for tetrahedral meshes [19]. This approach was improved by the authors [20] through the use of tetrahedral strips to decrease the amount of texture memory used. They also present an approach to render non-convex meshes by means of multiple restarts of the rendering process. Both works use multiple rendering passes to perform ray traversal. This leads to multiple rasterization and render-to-texture operations which turns into a slow down of the rendering process.

3 Implicit Integration of the Physical Model

Our physical model is based on the set of n interacting particles in a given tetrahedral mesh. Particle i interacts with the N_i particles connected to it by an edge. Interaction is represented by a linear spring model, for which the energy function E for two particles i and j is given by:

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

where L is the original distance between the particles, $\mathbf{x}_i$ and $\mathbf{x}_j$ their positions, $\mathbf{x}_{ij} = \mathbf{x}_j - \mathbf{x}_i$, and κ_s the spring constant.

Given the particles velocities v_i and v_j, we also include damping forces exerted on particle i from the interaction with particle j. Thus, the forces acting on particle i due to particle j are:

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

$$\mathbf{f}_i^{(d)} = -\kappa_d(\mathbf{v}_i - \mathbf{v}_j) \tag{3}$$

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 the particle that do not depend on its position or velocity. Then, the derivatives of the force with respect to the position and the velocity are the matrices given by:

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j} = \kappa_s \frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}} + \kappa_s \left(1 - \frac{L}{|\mathbf{x}_{ij}|}\right)\left(\mathbf{I} - \frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}}\right) \tag{4}$$

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{v}_j} = \kappa_d \mathbf{I}. \tag{5}$$

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:

$$\ddot{\mathbf{x}} = \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}) \tag{6}$$

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 6 is converted to a first-order differential equation:

$$\frac{d}{dt}\begin{pmatrix}\mathbf{x}\\ \mathbf{v}\end{pmatrix} = \begin{pmatrix}\mathbf{v}\\ \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}, \mathbf{v})\end{pmatrix} \tag{7}$$

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

$$\begin{pmatrix}\Delta\mathbf{x}\\ \Delta\mathbf{v}\end{pmatrix} = h\begin{pmatrix}\mathbf{v}(t) + \Delta\mathbf{v}\\ \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}(t) + \Delta\mathbf{x}, \mathbf{v}(t) + \Delta\mathbf{v})\end{pmatrix} \tag{8}$$

Following the groundbreaking work by Baraff and Witkin [4], we replace $\Delta\mathbf{x} = h(\mathbf{v}(t) + \Delta\mathbf{v})$ in the lower part of Equation 8 and take the first order approximation of a Taylor series expansion on $\mathbf{f}$, to form the system $\mathbf{A}\Delta v = \mathbf{b}$:

$$\left(\mathbf{I} - h\mathbf{M}^{-1}\frac{\partial\mathbf{f}}{\partial\mathbf{v}} - h^2\mathbf{M}^{-1}\frac{\partial\mathbf{f}}{\partial\mathbf{x}}\right)\Delta\mathbf{v} = h\mathbf{M}^{-1}\left(\mathbf{f}(t) + h\frac{\partial\mathbf{f}}{\partial\mathbf{x}}\mathbf{v}(t)\right) \quad (9)$$

which must be solved for $\Delta\mathbf{v}$ in order to find $\mathbf{x}(t+h)$ and $\mathbf{v}(t+h)$.

4 Hardware-Accelerated Simulation

To solve the linear system of Equation 9 on the GPU, we store the input data in 32 bits floating point textures. These textures hold the state information (vectors $\mathbf{x}$ and $\mathbf{v}$), the externals forces $\mathbf{f}^{(e)}$, connectivity information, and partial results obtained during the simulation. To store the results coming from a GPU computation for later use, we use the recently supported `EXT_framebuffer_object` extension.

4.1 Data storage

The input data is stored in five 2D square textures, `positions`, `velocities`, `external_forces`, `neighbours`, and `neighbourhood`, with dimensions given by $\lceil\sqrt{n}\rceil$, with the exception of `neighbours` which dimensions are $\lceil\sqrt{\sum_{i=1}^{n} N_i}\rceil$ [1] .

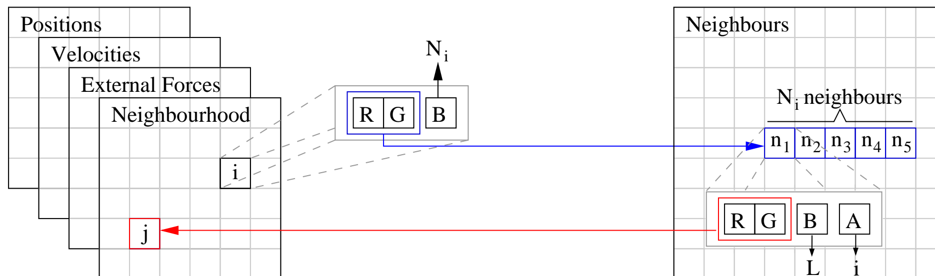

Fig. 1. Textures `neighbours` and `neighbourhood` hold the information of the neighboring particles.

For each particle i, texture `positions` holds its position in space, while texture `velocities` holds its velocity and mass. The sum of the external forces is stored in `external_forces`. Connectivity information is stored in two separate textures (See Figure 1). Texture `neighbourhood` stores a pointer to the position of texture `neighbours`, where the information of the N_i neighbours of the particle is stored. This information includes a pointer back to the position of the neighbour in the first four textures, the original distance between

[1] Better choices for the dimensions of these textures could be achieved using the results given in [17].

the particle and the neighbour, and the index of the particle. The number of neighbours N_i is stored in the B channel of texture `neighbourhood`. Although κ_s, κ_d, and h are passed as environment parameters, κ_s could be stored alternatively in the A channel of texture `neighbourhood` to allow the use of different local material properties.

This arrangement maps nicely to hold the sparse non-banded matrix computed when forming the linear system. We can think of each row of N_i neighbours 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.

4.2 Simulation algorithm

A typical 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 $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ and $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$.
(2) Calculate the force vector $\mathbf{f} = \mathbf{f}^{(s)} + \mathbf{f}^{(d)} + \mathbf{f}^{(e)}$.
(3) Form the linear system $\mathbf{A}\Delta\mathbf{v} = \mathbf{b}$ given by Equation 9
(4) Solve the linear system for $\Delta\mathbf{v}$ and update the state vectors $\mathbf{x}$ and $\mathbf{v}$.

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

4.2.1 Compute derivatives and forces

We perform two rendering passes to compute the derivatives and forces. In the first pass we calculate the non-diagonal elements of $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. For this, we render a quadrilateral covering a viewport of size $\lceil\sqrt{\sum_{i=1}^{n} N_i}\rceil$ to generate the fragments representing the non-zero non-diagonal elements of $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. Since each element of $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is a 3×3 matrix, the result of each fragment has 9 elements. We thus render the results to three target textures `non_diagonal_dfdx`k, $k = 0, 1, 2$ using the `GL_ATI_drawbuffers` and `EXT_framebuffer_object` extensions. The 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 $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, in the texels corresponding to the neighbours of the i-th particle. Textures `positions`, `neighbours`, and `neighbourhood`, as well as the parameters κ_s and h, are inputs to this pass.

In the second pass the diagonal elements of $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ and the force vector f are computed. Each diagonal element of $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is given by the negation of the sum of the non-diagonal elements in its row. Thus, in the second rendering pass we compute the diagonal elements by generating $\lceil\sqrt{n}\rceil$ fragments, and summing in each fragment the results of the previous rendering pass corresponding to its neighbours. Therefore, textures `non_diagonal_dfdx`k are inputs to the current

rendering pass. To access the information in textures `non_diagonal_dfdx`k we need textures `neighbours` and `neighbourhood`. Since the combined forces are also computed in this pass, we also need textures `positions`, `velocities`, and `external_forces` as input.

Thus, fragment i calculates the 3×3 matrix corresponding to the i-th diagonal element of $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, and the combined force corresponding to particle i. The result is stored in four target textures `forces` and `diagonal_dfdx`$k; k = 0, 1, 2$. We use the `NV_fragment_program2` extension to loop on the neighbours.

4.2.2 Form the linear system

The linear system is formed in three rendering passes. In the first pass we calculate the right side $\mathbf{b}$ of Equation 9 by generating $(\lceil \sqrt{n} \rceil)^2$ fragments. Each fragment calculates three elements of vector $\mathbf{b}$ to be stored in texture `b`. Since matrix $\mathbf{M}^{-1}$ is diagonal, we only need to loop over the neighbours of the particle i corresponding to the fragment to calculate $\mathbf{f}(t) + h\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\mathbf{v}(t)$ and multiply the result by $\frac{h}{m_i}$. For this, we need textures `non_diagonal_dfdx`k, `diagonal_dfdx`k, and `forces`, as well as the parameter h and the textures `neighbours`, `neighbourhood`, and `velocities` as inputs.

Next we must compute the matrix $\mathbf{A}$ on the left side of the linear system. In the second pass we calculate the non-diagonal elements given by $-h\mathbf{M}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{v}} - h^2\mathbf{M}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. To that end, we generate $(\lceil \sqrt{\sum_{i=1}^{n} N_i} \rceil)^2$ fragments, and 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`. We then add $-\frac{\kappa_d h}{m_i}$ to the diagonal elements of the partial result, and write this final result to the 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 $1 - \frac{\kappa_d h N_i}{m_i}$ to the diagonal elements and then write the result to textures `diagonal_A`$k; k = 0, 1, 2$. Note that we also need textures `velocities` and `neighbourhood` as input due to m_i and N_i.

With textures `diagonal_A`k, `non_diagonal_A`k, and `b` holding the matrix $\mathbf{A}$ and vector $\mathbf{b}$, our problem fits nicely to the GPU-based implementation of the Conjugate Gradients algorithm proposed by Krüger et al. [16].

4.2.3 Solve the linear system

Our arrangement of the data differs from the one proposed by Krüger and is more similar to the one proposed by Bolz et al. [17]. Thus, we fitted the GPU

matrix operations proposed by Krüger to work with our arrangement. One of the major differences with both works is the implementation of 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, so we can handle vectors of up to 255^2 elements. Further passes or nested `LOOP` instructions would be needed for larger vectors.

Matrix-vector multiplications also benefit from the `NV_fragment_program2` extension. 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 performs this operation and then adds the result to the contribution of the diagonal element.

Once the linear system is solved, the solution vector $\Delta\mathbf{v}$ is used as input to a final rendering pass, where n fragments are generated to update the position and velocity of each particle. 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 texture fetch operations available in `NV_vertex_program3` to access the deformed vertex positions. Each vertex fetches its correct position, which is then transformed and passed to the rasterizer. This allows us to avoid costly readback operations from the GPU.

Figure 3 shows deformations performed on two tetrahedral meshes using our approach. The images were generated using surface rendering. However, we focused on applications where the visualization of the inner information of the deformed mesh is a key issue. Yet, volume rendering the deformed mesh requires considerations not addressed in previous GPU-based volume rendering algorithms. Thus, in the next section we present a single-pass raycaster for deformable tetrahedral meshes.

5 GPU-based Volume Rendering of Deformable Tetrahedral Meshes

To perform raycasting of deformed meshes we need, besides the results of the simulation stored on texture `positions`, topology information which we store in two 3D textures `tetra_vertices` and `tetra_neighbours`. Texture `tetra_vertices` holds pointers to the vertices $v_{t,i}; i = 0, 1, 2, 3$ in the texture `positions` of each tetrahedron t, whilst texture `tetra_neighbours` stores

pointers to the neighbouring tetrahedra $ne_{t,i}$ in textures `tetra_*`. 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 $v_{t,i}$, 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 $v_{t,i}$ and by $\mathbf{n}_{t,i}$ the normal of $f_{t,i}$.

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`$i; i = 0, 1, 2, 3$.

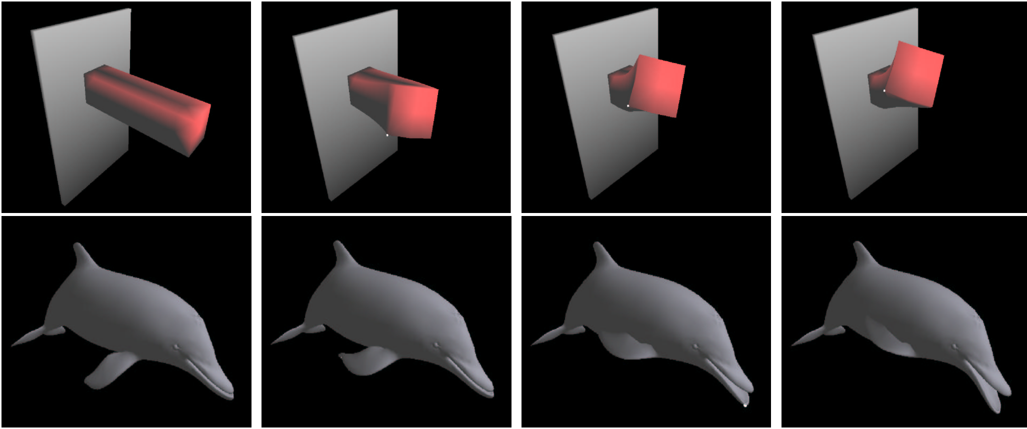

Fig. 3. Surface rendering of deformed tetrahedral meshes.

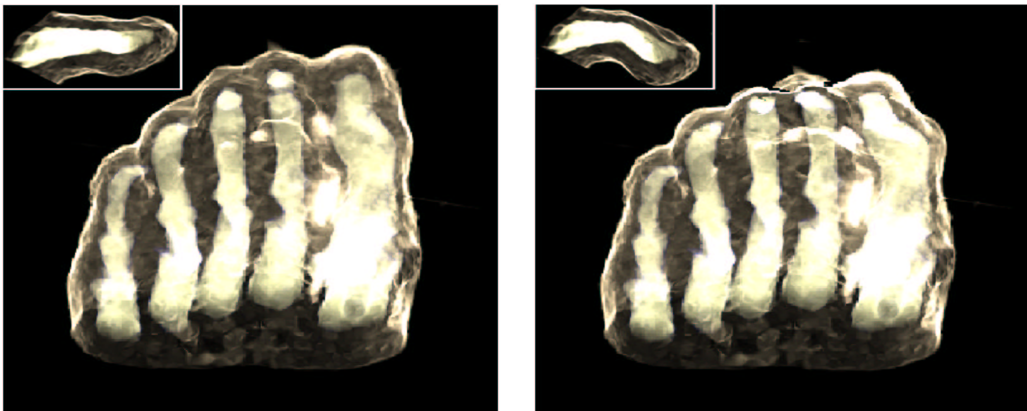

Fig. 4. Original and deformed mesh of the Foot dataset. A detail on a toe is shown in the upper link corner of each image.

5.1 *Single-pass raycasting of tetrahedral meshes*

The basic GPU raycaster starts with the rendering of the front faces of the volume to generate the fragments which will represent the casted rays. Ray traversal was 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. However, the advent of the full branching supported by `NV_fragment_program2` eliminates the need for multiple passes. Early ray termination is also easily included into the implementation by means of a `BRK` instruction.

Our algorithm starts rendering, as usual, the front faces of the volume. 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, the local index $i = 0, 1, 2, 3$ of the entry face, and the position in viewing coordinates of each fragment (which is also the first intersection of the ray with the volume).

During ray traversal we calculate the exit point of the ray for the current tetrahedron t as described in [19]. 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. For this, we need the normals $\mathbf{n}_{t,i}$ to each face $f_{t,i}$ of the current tetrahedron, which are stored in the textures `tetra_normals`i.

Known the entry and exit points, and since the gradient $\mathbf{g}_t$ within a tetrahedron is constant for barycentric interpolation, we calculate the scalar values at those points as $s(\mathbf{x}) = \mathbf{g}_t \cdot (\mathbf{x} - \mathbf{v}_{t,0}) + s_{t,0} = \mathbf{g}_t \cdot \mathbf{x} - \mathbf{g}_t \cdot \mathbf{v}_{t,0} + s_{t,0}$, where x is the entry/exit point. Therefore, we store $\mathbf{g}_t$ and $\hat{g}_t = -\mathbf{g}_t \cdot \mathbf{v}_{t,0} + s_{t,0}$ in the texture `tetra_gradients` to perform this calculation.

Once determined the scalar values at the entry and exit points, we calculate the distance d between them, and use these three parameters to perform a lookup in a 3D pre-integrated table [21], calculated using incremental pre-integration [19]. 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 $\widetilde{C}'_k = \widetilde{C}'_{k-1} + (1 - \alpha'_{k-1})\widetilde{C}_k$ and $\alpha'_k = \alpha'_{k-1} + (1 - \alpha_{k-1})\alpha_k$, obtaining the approximations $\widetilde{C}'_k$ and α'_k of the ray integral after k iterations.

5.2 Rendering deformable meshes

After a mesh is deformed we cannot assure its convexity, or the validity of the normals and gradients stored initially. Therefore, we introduced two further rendering passes after each simulation step. In the first pass we update the normals, rendering a polygon to generate one fragment per tetrahedron. Each fragment computes the new normals and gradient for its corresponding

tetrahedron t from the vertices positions $\mathbf{v}_{t,i}$ stored in `positions`. The result is then written in textures `tetra_normals`i, $i = 0, 1, 2, 3$.

Once the new normals are computed, we update the gradients in the second pass, where each fragment calculates $\mathbf{g}_t = \sum_{i=0}^{3} \frac{s_{t,i}}{\mathbf{n}_{t,i} \cdot (\mathbf{v}_{t,i} - \mathbf{v}_{t,3-i})} \mathbf{n}_{t,i}$. and $\hat{g}_t = -\mathbf{g}_t \cdot \mathbf{v}_{t,0} + s_{t,0}$ for its corresponding tetrahedron. These results are the written to texture `tetra_gradients`. Input to this pass are the textures `tetra_normals`i, `tetra_vertices`, and `positions`.

We handle non-convex meshes in a similar way as Weiler et al. [20]. We perform multiple passes of the raycasting algorithm, to 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 closest intersected fragment (using depth tests) and write it to a texture. The fragment depth d_s (in normalized space) is stored in a second texture and a new pass is started, in which we compare the stored depth with the current fragment depth d_c. If $d_c \leq d_s$, we set the fragment depth to $1 - d_c/10$ [2] and exit, otherwise we blend the stored color and opacity with the contribution of the ray traversal performed by he current fragment. The new depth, color and opacity are then stored to be used in the next pass. Figure 4 shows results of this algorithm for the Foot mesh.

6 Results and Discussion

In this section we present results describing the performance of our algorithms. All performance measurements were carried out on a standard PC equipped with an NVIDIA GeForce 7800 graphics board, a 3.8 GHz P4 CPU, and 2GB RAM.

	Mesh size	Frame rate [fps]				CPU Euler [fps]	
	Tetrahedra	Expl. Euler	Verlet	Veloc. Verlet	Impl. Euler	Explicit	Implicit
Bar	80	3880	3408	3454	282	5824	67
Dolphin	13766	1039	764	764	113	102	*
Panda	17312	945	612	614	100	80	*
Elephant	24106	900	539	541	48	58	*
Knee - small	35118	445	302	302	56	44	*
Knee	112299	169	119	119	22	14	*
Foot	156612	220	149	149	12	10	*

* maximum response time exceeded.

Table 1
Performance in fps of the integration methods on the GPU and CPU.

Table 1 shows the comparative results of the GPU-based implicit Euler and the straightforward GPU implementations of the explicit Euler, Verlet and velocity Verlet solvers. Frame rates for the integration process (not including rendering time) are given in fps. Timings for the naive CPU versions of the

[2] Note that we assume that no entry point will be farther away than 0.9.

implicit and explicit Euler are also included. In some cases, the CPU implementation of the implicit Euler was not able to solve the equation before the program aborted due to exceeded time without response. In debugging mode, the program solved the linear system successfully with a significant increase of the processing time.

To compare our results with those reported by Georgii et al. [15], we tested our implementation including surface rendering. For the Knee and Foot datasets, we obtained a performance of 88 fps and 144 fps respectively with the Verlet method. For a smaller mesh with 84104 tetrahedra, Georgii et al. obtain a performance of 121 fps using the Verlet method. In this respect, it is important to note that the integration time of Georgii et al. was considerable increased by the use of an extra artificial force for volume preservation. Although we initially included the volume preservation force in the explicit methods of our simulator, we did not obtain the results expected for our largest meshes. Thus, we excluded this force from our implementation considering also that it does not ensure stability in general.

On the other hand, our rendering time was increased by the texture fetches performed in the vertex shader, whilst Georgii et al. perform rendering only every 5th step of the simulation. However, texture fetches in the vertex stage will be eliminated with the expected support for binding vertex arrays to *framebuffer objects*, which will turn into a faster rendering.

It is important to remark that, for the Bar mesh, we were able to use a maximum time step of $h = 0.001s$ for $\kappa_s = 3000$ with the explicit integration methods. With higher κ_s or larger h, instability occurred. On the other hand, with the implicit Euler and $h = 0.01s$ we found no stability problem using κ_s larger than 30000. Georgii et al. used a time step of $h = 0.004s$ for their experiments with the Verlet solver. Comparing their results for the mesh with 84104 tetrahedra (121 fps), with our results for the quite larger Foot mesh using implicit Euler (11 fps including rendering time), we could state that, by setting $h = 0.05s$, similar simulation times could be obtained for the implicit method, while ensuring stability. This aspect is the main advantage of our solver.

Concerning the single-pass raycaster for deformable tetrahedral meshes which we proposed, we obtained for the Knee and Foot meshes frame rates of 11.21 fps and 6.2 fps, whilst Weiler et al. [19] obtained, for meshes of similar sizes (148955 and 124152 tetrahedra), frame rates of 5.13 fps and 2.27 fps. The performance obtained with the coupled simulation-visualization system was 7.30 fps and 5.62 fps with the explicit and implicit Euler solvers respectively for the Knee mesh, whilst for the Foot mesh we attained a performance of 4.34 fps and 2.57 fps with the explicit and implicit Euler schemes.

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, focusing 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 achieve 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.

Acknowledgments

We would like to thank Alex Cuadros-Vargas, University of Sao Paulo, for providing us with the tetrahedral meshes of the Foot and Knee datasets, and Prof. Rüdiger Westermann, Technical University Munich, for valuable discussion.

References

[1] S. F. F. Gibson, B. Mirtich, A survey of deformable modeling in computer graphics, Tech. rep., MERL - A Mitsubishi Electric Research Laboratory (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, Large steps in cloth simulation, in: SIGGRAPH '98: Proc. of the 25th annual conference on Computer graphics and interactive techniques,

1998, pp. 43–54.
[5] Y. Lee, D. Terzopoulos, K. Walters, Realistic modeling for facial animation, in: SIGGRAPH '95: Proc. of the 22nd annual conference on Computer graphics and interactive techniques, 1995, pp. 55–62.
[6] M. Müller, L. McMillan, J. Dorsey, R. Jagnow, Real-time simulation of deformation and fracture of stiff materials, in: Proc. of the Eurographic workshop on Computer animation and simulation, 2001, pp. 113–124.
[7] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, B. Cutler, Stable real-time deformations, in: Proc. of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation, 2002, pp. 49–54.
[8] M. Desbrun, P. Schröder, A. Barr, Interactive animation of structured deformable objects, in: Proc. of the 1999 conference on Graphics interface '99, 1999, pp. 1–8.
[9] P. Volino, N. Magnenat-Thalmann, Comparing efficiency of integration methods for cloth simulation, in: Proc. of Computer Graphics International, 2001, pp. 265–274.
[10] M. Teschner, B. Heidelberger, M. Müller, M. Gross, A versatile and robust model for geometrically complex deformable solids, in: CGI '04: Proc. of the Computer Graphics International (CGI'04), 2004, pp. 312–319.
[11] Microsoft Corporation. DirectX 9 SDK, http://www.microsoft.com/directx.
[12] NVIDIA OpenGL Extension Specifications for the CineFX 3.0 Architecture (NV4x), http://developer.nvidia.com/object/nvidia_opengl_spect.html.
[13] M. Harris, G. James, Physically-based simulation on graphics hardware, http://developer.nvidia.com/docs/IO/8230/GDC2003_PhysSimOnGPUs.pdf (2003).
[14] M. J. Harris, G. Coombe, T. Scheuermann, A. Lastra, Physically-based visual simulation on graphics hardware, in: HWWS '02: Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2002, pp. 109–118.
[15] J. Georgii, F. Echtler, R. Westermann, Interactive simulation of deformable bodies on GPUs, in: Simulation and Visualisation 2005, 2005, pp. 247–258.
[16] J. Krüger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, ACM Transactions on Graphics 22 (3) (2003) 908–916.
[17] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, ACM Transactions on Graphics 22 (3) (2003) 917–924.
[18] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys, A multigrid solver for boundary value problems using programmable graphics hardware, in: HWWS '03: Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2003, pp. 102–111.
[19] M. Weiler, M. Kraus, M. Merz, T. Ertl, Hardware-based ray casting for tetrahedral meshes, in: Proc. of IEEE Visualization '03, 2003, pp. 333–340.
[20] M. Weiler, P. Mallón, M. Kraus, T. Ertl, Texture-encoded tetrahedral strips, in: Proc. Symposium on Volume Visualization 2004, 2004, pp. 71–78.
[21] 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, 2001, pp. 9–16.