Interactive simulation and rendering of heterogeneous deformable bodies

Article · October 2013

CITATIONS
7

READS
89

2 authors, including:

Joachim Georgii
Fraunhofer Institute for Medical Image Computing MEVIS
59 PUBLICATIONS **539** CITATIONS

SEE PROFILE

Interactive Simulation and Rendering of Heterogeneous Deformable Bodies

Joachim Georgii, Rüdiger Westermann

Computer Graphics & Visualization Group
Technische Universität München
Email: {georgii, westermann}@in.tum.de

Abstract

In this paper, we present an interactive method for physics-based simulation and rendering of deformable bodies exhibiting heterogeneous material properties. To fully exploit the capabilities of consumer class computer systems, a CPU simulation engine is run in parallel with a GPU render engine. On the simulation side, we have developed a numerical multigrid solver for the governing equations of motion of deformable bodies. This solver allows for the combination of implicit solution methods and varying material parameters into an unconditionally stable simulation support system. The deformable model is rendered using a high-resolution surface mesh, which is subsequently updated on the GPU to exploit parallelism and memory bandwidth. Rendering of the deformed mesh can thus be performed without any additional bus-transfer. The interactive system is open to a variety of different applications ranging from surgery simulation, medical imaging and solid mechanics to computer animation and virtual sculpting.

1 Introduction and Contribution

Interactive and realistic deformation and rendering of volumetric objects is still difficult to achieve due to a number of reasons: First, no existing technique is able to realistically simulate the dynamic behavior of reasonably sized bodies at interactive rates. Second, volumetric objects as they appear in reality are usually composed of highly heterogeneous material exhibiting real and varying elasticity or density. Such objects require stable simulation systems, and the performance of the solver should ideally not depend on the material stiffness.

In this paper, we describe a system that comprehensively addresses the aforementioned issues. The proposed system consists of a CPU simulation engine that is run in parallel with a GPU (graphics processing unit) render engine. We present a multigrid solver for the simulation of the dynamic behavior of an elastic solid under external forces. Although the beneficial properties of multigrid methods are well known [1, 11], to the very best of our knowledge such methods have not yet been considered in computer graphics for the interactive simulation of deformable bodies. Our proposed solver is by far faster than previous approaches and it generates numerically stable results. The implicit nature of this method makes it amenable to the simulation of real-world objects, which can show an elastic modulus with a dynamic range of several orders of magnitude. In the real world, these parameters considerably affect the dynamic behavior of volumetric objects, which makes them important in a number of applications ranging from surgery simulation and image registration to solid mechanics. Besides physical realism, such parameters also provide a plausible means for controlling the dynamic behavior of deforming bodies (see Figure 8).

It is worth noting, that the proposed CPU multigrid scheme is even faster than optimized GPU simulation techniques [3, 22], yet providing significantly higher numerical accuracy and stability. Moreover, much larger models can be handled using our approach, due to limited video memory on current GPUs. As an additional advantage over pure GPU approaches, the simulation system we present in this work produces highly balanced load on the CPU and the GPU. Thus, the engine is even more superior to such approaches if rendering is considered.

To render the deforming volumetric body we have developed a render engine for triangular meshes that continually change their shape. This engine is entirely implemented on the GPU, and it only re-

quires the transfer of per-node displacements of the coarser finite element mesh from the CPU. The integration of the simulation and the render engine into a simulation support system enables the instantaneous visualization of physics-based deformations due to external forces.

1.1 Related Work

Physics-based deformation techniques for volumetric bodies have a long tradition in computer animation and medical imaging [9, 19]. Less accurate finite difference and mass-spring models have been considered as well as more elaborate and physically accurate finite element techniques [2]. As we want to consider realistic and varying stiffness, explicit [7, 24] or mixed explicit/implicit [16] time integration schemes for finite elements don't seem to be suited, due to the Courant condition that significantly limits the largest possible time step for very stiff materials. Unconditionally stable implicit solvers usually employ conjugate gradient methods [18, 8] or matrix pre-inversion [5]. These approaches, however, do not scale linearly and therefore they are limited in the number of elements they can handle interactively. Along a different avenue, multiresolution techniques have been employed to accelerate the simulation of deformable objects based on adaptive refinements [6, 7, 12, 13]. Even though an explicit multigrid scheme for surfaces was presented [23], it can not be used in our case due to the Courant condition. The specification of arbitrary material properties is still a challenge to many of these techniques, due to both stability and performance reasons.

2 System Overview

The simulation support system consists of two parallel processes. The simulation engine is implemented on the CPU and computes the displacement of an elastic solid under external forces. The render engine is realized on the GPU. It receives computed displacements and updates the geometry of an associated surface model accordingly. While the simulation engine consecutively displaces the underlying finite element grid, i.e., the simulation geometry, the render engine subsequently changes the geometry of the render object, i.e., the render geometry. It is attached to the simulation geometry via a weighting function. Without loss of generality, tetrahedral grids serve for the simulation geometry and triangular meshes serve for the render geometry. A clear conceptual separation between the simulation geometry and the render geometry enables the flexible variation of the geometries used, i.e. regular grids, point sets, or volumetric render grids.

The system is parallelized using posix threads to instantiate a separate simulation and rendering thread. This is necessary because the execution of rendering commands on the GPU may block the application process, whereas only the calling thread is blocked in a multi-threaded environment. The spawning process allocates memory that is shared by both threads to write and to read computed displacements. Both threads are synchronized via conditional variables, which allow synchronization based upon the actual value of data. Even more, by exploiting hyper-threading architectures, idle times of the threads can be reduced noticeably.

The particular system design has several other properties that accommodate its use in the confined area:

- By decoupling the resolution of the simulation geometry from the resolution of the render object, one can flexibly trade simulation or rendering quality for speed.
- The use of two separate grids, of which the render grid is attached to the simulation grid via barycentric weights, enables approximate deformation of objects that are made of far more elements than the simulation engine can handle.
- After an initial delay of one simulation frame the system operates in full pipeline mode. Optimally, this allows the system to double its performance compared to an implementation on a single processing unit.
- To update the render geometry, only the displaced simulation vertices have to be transferred. Bandwidth requirements can thus be reduced.
- The parallel approach allows the exploitation of parallelism and memory bandwidth in the fragment units of recent GPUs to update and to display the render geometry.

Running the proposed system involves a number of preprocesses as well as model driven computations at run-time. In the following we will describe the different modules the system is comprised of from a high-level view.

2.1 Precomputation

Starting with an initial object representation, a tetrahedral hierarchy is generated that constitutes the basis for the multigrid method. If such a hierarchy is already given, it can be used directly. Dedicated data structures to render the deformed high-resolution render surface are created and initialized on the GPU.

1. Construct a 3D finite element mesh, either by using a triangle mesh and a tetrahedral mesh generation package such as NETGEN [21], or by using an adaptive subdivision scheme for tetrahedral grids [10].
2. Assign material properties such as stiffness and density to finite elements depending on material characteristics, using pre-computed transfer functions [14] or segmentation results.
3. Where deformations are not allowed, fix boundary vertices of the finite element model.
4. Generate a finite element mesh hierarchy including geometric correspondences between meshes.

   Generate a triangle mesh hierarchy, e.g. by using a mesh decimation package, and generate a finite element mesh for each hierarchy level as described. Alternatively, use different levels generated by the tetrahedral subdivision scheme.
5. Construct a triangular render geometry. This can be the surface of the finest resolution finite element mesh, a simplified or detailed version of this mesh, or a completely different mesh.
6. Bind the render mesh vertices to vertices of the finite element mesh.
7. Store vertices of the highest resolution finite element model into a 2D texture map. Upload both the 2D texture and the render geometry including per-vertex indices and associated weights into that texture to the GPU.

2.2 Runtime Computations

At runtime, the following steps are performed by the simulation support system:

1. Based on external forces, compute the displacement of finite element vertices using the multigrid method.
2. Store the displacement vectors (at object boundaries only) in a 2D texture map and upload this texture to the GPU.
3. Displace the render surface on the GPU and render this surface.

3 Simulation

The motion of a deforming volumetric object can be simulated by a displacement field in an elastic solid. Given such a solid in the reference configuration Ω, the deformed solid is modeled using a displacement function $u(x), u : \mathbb{R}^3 \rightarrow \mathbb{R}^3$. This function describes the displacement vector at every point $x \in \Omega$, yielding the deformed configuration $x + u(x)$.

3.1 Linear Elasticity Theory

Driven by external forces, the dynamic behavior of the deformed solid is governed by the Lagrangian equation of motion

$$M\ddot{u} + C\dot{u} + Ku = f \tag{1}$$

where M, C, and K are respectively known as the systems mass, damping and stiffness matrices. u consists of the linearized displacement vectors of all vertices and f contains the force vectors applied to these vertices.

If a finite element method is used to model the system, system matrices are built by assembling all element matrices. Since each element in the finite element discretization only has a very small number of neighbors, this system is very sparse. We are using tetrahedral elements with linear nodal basis functions. Displacements are expanded in a basis of shape functions Φ as

$$u(x) = \Phi(x)u^e,$$

where $u^e = (\underline{u}_1^T, \ldots, \underline{u}_4^T)^T$ contains the single node displacements.

Instead of simple mass lumping, we compute a consistent mass matrix M^e as

$$M^e = \int_{\Omega^e} \rho\, \Phi^T(x)\Phi(x)dx,$$

where ρ is the element density. Assembling these components leads to a real symmetric matrix. We use either simple proportional damping $C = \alpha M$ or Rayleigh damping $C = \alpha M + \beta K$.

The stiffness matrix K accounts for the internal energy associated with the displacement field, and it is thus dependent on the elastic energy stored in the solid and on the work done by body forces and tractions applied through the displacement field u. In linear elasticity the strain tensor $\mathcal{E}$ describes the linear relation between deformation and displacement:

$$\mathcal{E}_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2}$$

In an isotropic and fully elastic body, stress (Σ) and strain tensors are coupled through Hooke's law

$$\Sigma = \lambda \left(\sum_{i=1}^{3} \mathcal{E}_{ii} \right) \cdot I_{33} + 2\mu \, \mathcal{E}, \qquad (3)$$

with the Lamé coefficients λ and μ [4].

Given the nodal basis functions as well as strain and stress tensors, the potential energy V of every element can be computed, from which the first variation of the equilibrium equation $\frac{\partial V}{\partial u_k} = 0$ is derived. The resulting single element equations are finally assembled into a system of linear equations.

To solve this system, implicit Euler integration is often employed. Because artificial damping is typical to this integration scheme, a Newmark scheme is used for second order accurate time integration:

$$\begin{aligned} \dot{u}^{t+dt} &= \dot{u}^t + \left(\frac{1}{2} \ddot{u}^t + \frac{1}{2} \ddot{u}^{t+dt} \right) dt \\ u^{t+dt} &= u^t + \dot{u}^t \, dt + \left(\frac{1}{4} \ddot{u}^t + \frac{1}{4} \ddot{u}^{t+dt} \right) dt^2 \end{aligned}$$

By discretizing u as well as the partial derivatives of u with respect to time, and by replacing $\dot{u}^{t+dt}$ and $\ddot{u}^{t+dt}$ in equation (1), the system of linear equations $\tilde{K} u^{t+dt} = \tilde{f}^{t+dt}$ can be formulated. Because in linear elasticity the stiffness matrix does not change, and because both M and C do not change either, the system matrix $\tilde{K}$ remains unchanged as long as topology changes do not occur.

3.2 Multigrid Method

For the efficient simulation of an elastic deformable solid we have developed a geometric multigrid method. In particular, this method includes geometry-specific relaxation, restriction, and interpolation operators. These operators form the essential multigrid components.

Multigrid methods give rise to scalable linear solvers. A relaxation method like Gauss-Seidel is used to efficiently damp high-frequency error. The remaining low-frequency error can be accurately and efficiently solved for on a coarser grid. Recursive application of this basic idea to each consecutive system on a hierarchy of grid levels leads to a multigrid V-cycle.

In this work, we define an appropriate finite element hierarchy, which allows for an efficient implementation of multigrid components. The result is a method that uniformly damps all error frequencies with a computational cost that depends only linearly on the problem size.

3.2.1 Unstructured Hierarchy

The geometric multigrid method requires a mesh hierarchy that represents the deformable object at different resolution levels. On this hierarchy, appropriate transfer operators to map quantities between different levels have to be designed. Starting with a finite element mesh at the coarsest resolution level, a common way to construct the hierarchy in a top-down approach is to split each tetrahedron as shown in Figure 1. The octahedron is subsequently split into four tetrahedra, such that eight children are generated overall. This approach results in a nested hierarchy, which allows the transfer operators to be defined straight forwardly, but it requires the initial mesh to be fine enough to achieve a proper representation of the object at ever finer resolution levels.

For volumetric objects given on a Cartesian grid this requirement does not pose a restriction, because at every level the entire domain is covered. If we start with a coarse representation of an arbitrary object, however, rather shallow hierarchies are constructed and the multigrid method can not be used to its full potential. In addition, because the subdivision scheme does not account for the objects surface at the finest scale, it leads to poor visual results. To avoid these drawbacks, we propose linear transfer operators that do not require a nested hierarchy, and which can be integrated efficiently into the multi-

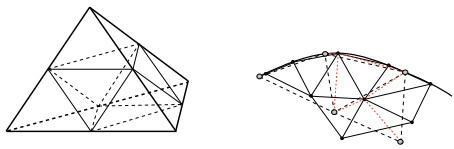

Figure 1: Left: tetrahedral subdivision lends itself directly to a nested hierarchy. Right: geometric relations between elements in the non-nested hierarchy are illustrated (the 2D case is shown for simplicity). Dotted and solid lines indicate the coarse and the fine mesh respectively. Barycentric interpolation weights are highlighted by dotted (red) lines.

grid scheme. These operators establish relations in a multilevel hierarchy of unstructured and unrelated meshes by means of barycentric interpolation as illustrated in Figure 1.

Given the restriction operator I_h^H and the interpolation operator $I_H^h = (I_h^H)^T$, where h denotes the fine mesh and H the coarser mesh, the system matrix at H is computed as

$$K^H = I_h^H K^h I_H^h.$$

Only in this case we can ensure optimal convergence of the multigrid scheme, which lacks the tendency of finite elements methods to behave differently on distinct levels of detail [4].

Overall, with both the nested and the non-nested unstructured hierarchy we achieve nearly the same performance. However, the latter one allows for an improved approximation of the continuum model. In particular, it is possible to refine the mesh in critical regions in a pre-process while keeping tetrahedral elements well shaped.

3.2.2 Linear Elasticity Multigrid

Given the linear transfer operators I_h^H and I_H^h as well as an initial approximation $\hat{u}^h$ of the displacement values of a deformable solid on a fine grid, then a new approximation u^h can be computed as follows (e^h and e^H denote the error):

① compute residual	$r^h = f^h - K^h \hat{u}^h$
② restrict residual	$r^H = I_h^H r^h$
③ solution on coarse grid	$K^H e^H = r^H$
④ transfer correction	$e^h = I_H^h e^H$
⑤ correction	$u^h = \hat{u}^h + e^h$

The algorithm is extended to a 2-grid approach by pre-smoothing the residual prior to stage ① to avoid the transfer of quantities from the h-grid that can not be reduced on the H-grid, and by post-smoothing the result after stage ⑤ to avoid high frequencies introduced by numerical inaccuracies. In general, $1-2$ Gauss-Seidel steps are sufficient for pre-smoothing and post-smoothing, and they keep the relative error smaller than 10^{-3} in our examples. However, if the external force field does not change anymore, the relative error decreases to 10^{-6} after few timesteps. Given the solution of the system of equations on the coarse grid using matrix inversion, a full multigrid V-cycle can be derived.

In every simulation step and for every vertex of the finite element mesh at the finest level, displacement vectors are computed by the simulation engine. Updated vertex positions of the boundary elements are uploaded to a 2D RGBA texture – the displacement map – onto the GPU. The fourth color channel is used to store an additional scalar per-vertex attribute, which can be visualized.

3.3 Simulation Results

In the following, we give several examples that demonstrate the efficiency of the physics-based simulation engine. The solution of the system of linear equations is computed on a P4 3.0 GHz processor using the proposed multigrid method. One Gauss-Seidel step was used both for pre- and post-smoothing in all examples of Table 1. An integration time step of 0.03sec allows for a stable simulation with a relative error less than 10^{-3}. As shown in Table 1 and Figure 2, the multigrid method scales linearly with the number of elements, and it achieves excellent performance rates even for large models. To our best knowledge, a similar performance can not be achieved so far using any other physics-based finite element method. In particular, the comparison to a diagonal preconditioned conjugate gradient method (see Figure 2) shows an ever increasing performance gain of our proposed multigrid method for larger meshes.

model	elements	vertices	levels	timesteps per sec.
cloth1	8192	4225	7	150
cloth2	131072	66049	9	9
car	7008	2341	2	160
horse	10268	2815	2	125
bunny	11206	3019	2	113
horse	49735	12233	3	20
bunny	89648	19266	3	13
bridge	196608	35937	6	8
bridge	1572864	274625	7	1

Table 1: Timing statistics for differently sized triangular (first two rows) and tetrahedral models. The performance for the triangular grids is not as high as expected due to the high vertex count of the used meshes.

Compared to previous approaches, the implicit multigrid solver enables much larger integration time steps of up to one second. Even more importantly, both the time step and the number of itera-

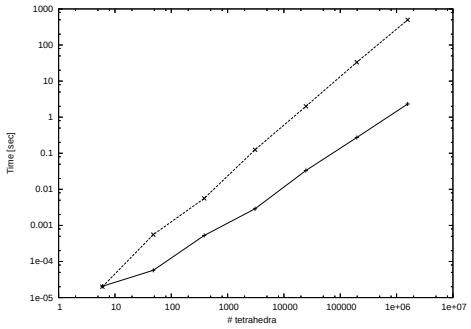

Figure 2: Comparison of the time required until convergence on a double logarithmic scale for the multigrid method (solid lines) and the diagonal preconditioned conjugate gradient method (dashed lines). The former method scales linearly with the number of tetrahedra, while the latter one requires ever more iterations to achieve the same relative error of 10^{-4} in the solution. The timings where measured using a cube model that was subsequently refined by the split operation shown in Figure 1, and a fixed integration time step of 0.02sec.

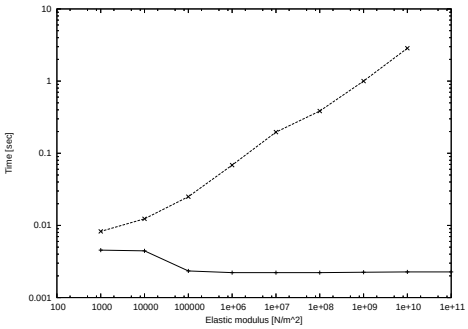

Figure 3: Performance measures for a bridge model consisting of $3k$ tetrahedral elements. For ever stiffer materials, the conjugate gradient method (dashed line) requires more steps to be performed to achieve the same relative error of 10^{-4} as the multigrid method (solid line). The elastic modulus affects the performance of the conjugate gradient method significantly, while it does not affect the performance of the multigrid method. Only for extremely soft materials, the performance of the multigrid method drops down, as forces cause only very local deformations that cannot be solved for on a coarser grid.

tions until convergence do not depend on material stiffness as illustrated in Figure 3. Especially for stiff materials, the multigrid method is far superior to the ill-conditioned conjugate gradient method. The multigrid solver enables stable simulation of heterogeneous bodies with an elastic modulus varying from 10^3 to 10^{11}. In Figure 6, different stiffness values have been assigned to different parts of a finite tetrahedral mesh. This mesh was generated from a triangular horse model. In particular, all four legs exhibit very high stiffness, while the abdomen is made of soft material. As a matter of fact, the abdomen moves to the ground due to gravity, while the legs keep the rest of the horse in shape. Figure 11 shows the influence of wind force and gravity to bars of different density and stiffness. As the force is constant everywhere, softer bars are deformed much more significantly than stiffer ones.

Finally, let us mention that the proposed multigrid solver is not limited to tetrahedral models. Arbitrary triangular models like cloth can be simulated in the same way, by adapting the system to triangular finite elements. This change only influences the internal structure of the stiffness matrix as well as the interpolation operator required by the multigrid scheme. Figure 5 shows some deformations of triangular surface models under external forces.

4 Rendering

On the GPU, the render engine updates a render mesh, i.e. a triangular mesh, according to uploaded displacements of the boundary of the simulation mesh. Vertices of the render mesh are bound to vertices of the simulation mesh via interpolation weights, which are pre-computed and stored on the GPU. The deformation of high resolution meshes can thus be driven by the simulation engine at minimum bus transfer. Furthermore, parallelism as well as memory bandwidth on recent GPUs can be exploited to update the render mesh.

4.1 Render Surface

The high-resolution render surface, which resides in local GPU memory, is represented as an index array that contains for every triangle references into a geometry image with associated per-vertex attributes. All containers are internally stored as 2D textures. Every vertex in the geometry image gets assigned additional references into the displacement texture that is sent from the CPU. These references are accompanied by barycentric interpolation weights. Four references are stored, one for every vertex of

the tetrahedron closest to the render vertex. Once displaced vertices of the simulation mesh are uploaded on the GPU, a fragment program fetches respective vertex coordinates $\vec{v}_i$ and interpolation weights w_i and computes the new vertex position as $\sum_{i=0}^{3} \vec{v}_i \cdot w_i$. If additional vertex attributes are sent with the displacement texture, i.e. color, texture coordinates, these values are interpolated as well. The fragment output is finally rendered into a 2D texture render target. To render the displaced triangle mesh, different possibilities are available on recent GPUs – OpenGL SuperBuffers and vertex texture fetches using Shader 3.0 or GLSL.

To provide the application program with better control of the GPUs local video memory, the OpenGL SuperBuffer extension [20] has been introduced. It defines a *memory object* that holds a piece of raw video memory. By using non-standardized memory objects, i.e. ATI's UberBuffers, a memory object can subsequently be bound as the current texture render target and as a vertex array used to draw particle primitives. Because this approach restricts the application to a particular GPU architecture, however, in the current work we exploit the possibility to perform texture fetches in the vertex units of the GPU. Although this method is actually less efficient than the one using UberBuffers, similar performance can be expected in the near future due to optimized texture caches in the vertex units of upcoming graphics processors.

On traditional graphics architectures, textures could only be accessed in a fragment shader program. The Shader 3.0 and the GLSL specification also enable texture access in the vertex units hence providing an effective means for displacing geometry on the GPU. This functionality is supported on recent *n*VIDIA graphics hardware, and it will also be supported on upcoming ATI cards. To render a displaced surface, we render a static vertex array stored in GPU memory. In a vertex shader program, vertex positions are fetched from the texture that contains the displaced vertex positions, and the vertex coordinate is displaced accordingly. Therefore, any read back of data to CPU memory is avoided.

4.2 Results

The timings in Table 2 have been generated on a *n*VIDIA 6800 GT graphics card equipped with 256MB local video memory.

model	#tris	Update	Update & Rendering
sarah	16k	1ms	515fps
car	36k	1ms	215fps
bunny	64k	2ms	100fps
horse	64k	2ms	80fps

Table 2: Performance of GPU surface render engine for different models. In the third column, timings are given for uploading displacement textures of size 32^2 and 64^2, respectively, to the GPU, and for updating the render geometry. The last column shows the overall performance of the GPU render engine including the update of the render surface, normal calculation, texture fetches in the vertex shaders to access the displaced vertex coordinates and the normals, as well as per-pixel lighting.

Figure 7 shows the bunny model consisting of 11k tetrahedral elements. Besides a high-resolution render surface (32k triangles), a GPU fur shader can be applied [15]. Rendering performance decreases, as several layers have to be rendered. However, geometry has only to be updated once. Figure 9 and 10 give additional examples for interactive deformation and rendering using the proposed system.

5 Future Work

With regard to the internal properties of the material to be simulated, the presented system considers the linear Cauchy strain measure. This measure omits higher order terms in the strain tensor, and for large deformations this approximation does not allow for an accurate simulation of the resulting displacements.

Multigrid methods are not limited to the solution of system of linear equations, and the proposed solver can be extended to the simulation of non-linear strain measures (see Figure 4). Solving a non-linear system of equations, however, significantly reduces the number of elements that can be rendered at acceptable rates. A good trade-off between Cauchy and Green strain tensors can be achieved using the so-called corotational strain of linear elasticity [17], in which finite elements are first rotated into their initial configuration before the strain is computed. As this method fits nicely into the multigrid framework, one future research direction will be to incorporate corotational strain into the implicit solver.

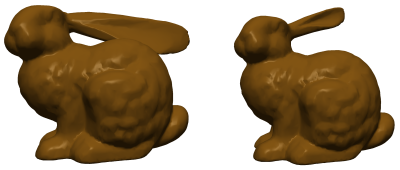

Figure 4: The images show the displacements due to external forces as they were simulated using the linear Cauchy strain tensor (left) and the non-linear Green strain tensor (right), respectively.

6 Conclusion

We have described a system for interactive and physics-based simulation and rendering of deformable bodies. This system consists of a CPU simulation engine that runs in parallel with a GPU render engine. It allows users to interact with volumetric objects by simulating and directly visualizing the results of external forces acting on these objects. By leveraging commodity parts, the building blocks of the system can be easily upgraded as technology improves. We will extend our simulation engine to allow for the corotational strain of linear elasticity as well as the non-linear Green strain measure, as both can be combined with our multigrid solver.

References

[1] Mark Adams and James W. Demmel. Parallel multigrid solver for 3d unstructured finite element problems. In *ACM/IEEE Supercomputing*, 1999.

[2] K.-J. Bathe. *Finite Element Procedures*. Prentice Hall, 1995.

[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *Proceedings of ACM SIGGRAPH 2003*, 2003.

[4] D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge Univ. Press, 2001.

[5] M. Bro-Nielsen and S. Cotin. Real-time volumetric deformable models for surgery simulation using finite elements and condensation. In *Proceedings of Eurographics*, 1996.

[6] S. Capell, S. Green, B. Curless, T. Duchamp, and Z. Popovic. A multiresolution framework for dynamic deformations. In *ACM SIGGRAPH Symposium on C. Animation*, 2002.

[7] G. Debunne, M. Desbrun, A. Barr, and M.-P. Cani. Dynamic real-time deformations using space & time adaptive sampling. In *Proceedings of SIGGRAPH '01*, 2001.

[8] O. Etzmuß, M. Keckeisen, and W. Straßer. A fast finite element solution for cloth modelling. In *Pacific Conference on Computer Graphics and Applications '03*, 2003.

[9] S. F. Gibson and B. Mirtich. A survey of deformable models in computer graphics. In *Technical Report TR-97-19, Mitsubishi*, 1997.

[10] Gunther Greiner and Roberto Grosso. Hierarchical tetrahedral-octahedral subdivision for volume visualization. *The Visual Computer*, 16(6):357–369, 2000.

[11] M. Griebel, D. Oeltz, and M.A. Schweitzer. An algebraic multigrid method for linear elasticity. *SIAM J. Sci. Comput.*, 25(2):385–407, 2003.

[12] Eitan Grinspun, Petr Krysl, and Peter Schröder. Charms: a simple framework for adaptive simulation. In *Proceeding of SIGGRAPH '02*, 2002.

[13] M. Hauth, J. Groß, and W. Straßer. Interactive physically based solid dynamics. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, 2003.

[14] Gordon Kindlmann and James W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 79–86, 1998.

[15] J. Lengyel, E. Praun, A. Finkelstein, and H. Hoppe. Real-time fur over arbitrary surfaces. In *ACM Symposium on Interactive 3D Graphics*, 2001.

[16] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, and Barbara Cutler. Stable real-time deformations. In *ACM SIGGRAPH/EG symp. on Computer animation '02*, 2002.

[17] M. Müller and M. Gross. Interactive virtual materials. In *Proceedings of the 2004 conference on Graphics interface*, 2004.

[18] M. Müller, L. McMillan, J. Dorsey, and R. Jagnow. Real-time simulation of deformation and fracture of stiff materials. In *Eurographics Workshop on Computer Animation and Simulation '01*, 2001.

[19] A. Nealen, M. Müller, R. Keiser, E. Boxerman, and M. Carlson. Physically based deformable models in computer graphics. In *Eurographics '05*, 2005.

[20] nVidia. Data Storage and Transfer in OpenGL. http://developer.nvidia.com/docs/IO/8229/Data-Xfer-Store.pdf.

[21] J. Schöberl. Netgen - an advancing front 2d/3d-mesh generator based on abstract rules. *Comput.Visual.Sci*, 1997.

[22] E. Tejada and T. Ertl. Large steps in gpu-based deformable bodies simulation. *to appear in Simulation Modelling Practice and Theory*, 2005.

[23] X. Wu and F. Tendick. Multigrid integration for interactive deformable body simulation. In *Medical Simulation '04*.

[24] Y. Zhuang and J. Canny. Real-time simulation of physically realistic global deformation. In *IEEE Vis '99*, 1999.

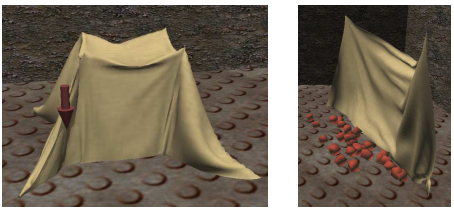

Figure 5: The simulation engine allows for the deformation of triangle meshes as well.

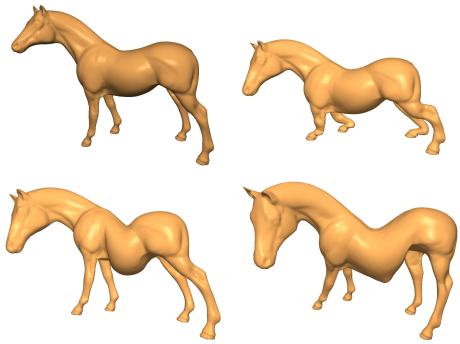

Figure 6: The deformation of a tetrahedral horse model is shown. From top left to bottom right we show the initial model, the model exhibiting homogeneous and inhomogeneous stiffness under gravity, and the heterogenous model under external forces.

Figure 7: Interactive deformation and rendering of the bunny finite element model (11206 simulation tetrahedra). The high resolution surface with 32k triangles is rendered using a GPU-based fur shader.

Figure 8: An ever softer tetrahedral bunny model under the influence of gravity is simulated.

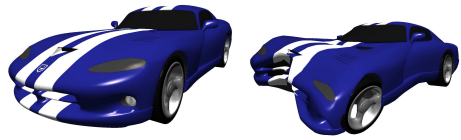

Figure 9: Interactive deformation and rendering of the car model (7k simulation tetrahedra) can be performed with 150 fps. The high resolution surface consists of 36k triangles.

Figure 10: Interactive deformation and rendering of a manikin. The fur shader accounts for the bodies dynamic deformation as can be seen on the right hand side.

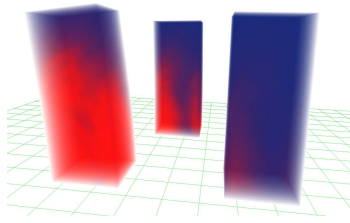

Figure 11: A visualization of the internal states (i.e. von Mises stress) of three towers exhibiting different stiffness is shown. A standard approach for direct volume rendering is used to generate the images. A constant wind force is applied to all models. Stress values are color coded ranging from red (high) to blue (low).