Interactive Simulation of Deformable Bodies on GPUs

Joachim Georgii   Florian Echtler   Rüdiger Westermann

Computer Graphics & Visualization Group
Technical University Munich

Abstract

We present a mass-spring system for interactive simulation of deformable bodies. For the amount of springs we target, numerical time integration of spring displacements needs to be accelerated and the transfer of displaced point positions for rendering must be avoided. To fulfill these requirements, we exploit features of recent graphics accelerators to simulate spring elongation and compression in the graphics processing unit (GPU), saving displaced point masses in graphics memory, and then sending these positions through the GPU again to render the deformed body. This approach allows for interactive simulation and rendering of about one hundred thousand elements and it enables the display of internal properties of the deformed body. To further increase the physical realism of this simulation, we have integrated volume preservation and additional physics based constraints into the GPU mass-spring system.

1 Introduction

To study the motion of a mechanical system caused by external forces, physics based simulation is needed. For a set of connected rigid or flexible parts exhibiting material dependent properties, the equations of motion can be formulated and solved to predict the dynamic behavior of such systems. Even for simple abstractions, however, calculations involved are usually too expensive as to allow for real-time simulation of reasonably sized objects. To visualize system dynamics, the geometric representation of the system has to be modified according to the computed motion. If the simulation is carried out on the CPU, in every animation frame the displaced geometry has to be sent to the GPU for rendering purposes, thus decreasing performance.
In this paper, we have implemented a mass-spring system based on tetrahedral grids. Element edges are treated as springs, which connect pairs of mass points. Under the influence of external forces, e.g. forces exerted by user interaction, gravity or collision forces, the object deforms into a configuration where the external forces are compensated by opposing internal forces. In the most basic form, only the springs themselves exert forces by which they seek to preserve their rest length when compressed or stretched. The resulting force can then be obtained using Hooke's Law.
The proposed implementation distinguishes from previous approaches in that physics based simulation and rendering of deformable bodies is performed on programmable graphics

hardware. In parallel fragment units, forces applied to a vertex are computed and accumulated. To render the deformed volumetric body, we exploit a feature on recent ATI graphics hardware that allows graphics memory to be treated as a render target, a texture, or vertex data. Thus, displaced vertex coordinates can be directly rendered without any read-back to CPU memory. In this way, a significant speed up can be achieved both for numerical simulation as well as for rendering. As our test have shown, the GPU realization performs about a factor of 20 faster than CPU solution. Moreover, because the entire body resides in GPU memory, not only can the surface of the body be displayed but also interior properties like forces or stresses.

2 Related Work

Simulation of deformable bodies has evolved as an important topic of research. Most commonly, simulation techniques for deformable models in computer animation account for elastic or viscoelastic (damped) Hookean materials. In solid mechanics the governing equations describing such materials have been studied extensively, and they have been frequently employed in computational science and engineering.

In computer graphics, many different approaches for simulating deformable objects have been derived over the last decades (see [GM97] for a thorough overview of early work in this field). From a large scale perspective, previous techniques for simulating the changes of volumetric objects due to external and internal forces can be classified according to the underlying object discretization, the object's intrinsic deformation behavior, i.e. strain measure, and the method employed to integrate the equations of motion over time.

Finite element methods, originally arising from the area of computational sciences, are commonly employed to realistically model continuous deformable objects. Volumetric objects are usually decomposed into linear tetrahedral elements, and the equations of elasticity theory are solved based on this discretization. The solution to these systems can either be found implicitly, e.g. in [BNC96, MMDJ01], or by using time explicit solvers as it was done in [ZC99, DDBC01, WDGT01, MDM$^+$02]. However, these approaches result in high numerical effort, and hence only allow for the simulation of small models in realtime.

The most efficient techniques for simulating deformable objects based on physical properties are mass-spring systems. Lots of work has been done in this specialized field, including [LTW95, DSB99, BFA02, FGL03]. Mass-spring systems cannot simulate the real physical behavior of the deformable body, as they only use a simplified model. Continuous bodies are approximated by a finite set of point masses, which are connected via links to account for material stiffness. But, using this simple method, one can achieve great visual results in applications, where physical accuracy is not necessary. As a consequence of model's simplicity, mass-spring systems were often extended to more realistically approximate properties in the specific application area [Pro95]. For example, [LTW95, THMG04] expanded mass-spring systems to incorporate volume-preserving forces and plasticity.

But mass-spring systems lack in that finding proper spring constants to realistically simulate real materials is quite cumbersome. [Gel98] showed, how to choose spring constants to model homogenous materials. Spring constants can also be configured automatically by

neural networks that have been trained to mimic the dynamic behavior of special materials [RNP01].
After this short overview of previous work, we now will continue with some deeper insight into physics and dynamics of mass-spring systems. Afterwards, we will show how to implement these techniques on a programmable graphics processor. At the end, we will show our results and compare them to CPU solutions.

3 Theory

3.1 Mass-Spring Systems

Let us continue with a closer look at the ideas behind mass-spring systems. Starting with a volumetric body representation, we imagine the mass of the body condensed in discrete vertices, which are connected to each other via springs. Doing so, we get a representation based on single mass-points which are connected in an irregular way. However, the springs can freely rotate, but have to satisfy Hooke‘s law

$$F_i = D\frac{|l| - l_0}{|l|} \cdot l.$$

Here, D describes the stiffness of the spring, while $l = x - x_i$ is the distance between two connected mass-points p and p_i. l_0 is the rest length of the spring in its initial configuration. Thus, the resulting force of this spring acting on p is calculated as shown above. Now you can accumulate the forces over all neighbors of p, which should be in balance with external forces F_{ext} acting on the single mass-point.
To update the position x of a vertex in a dynamic simulation, we use Lagrange‘s law of motion

$$m\,\ddot{x} + c\,\dot{x} + \sum_{i \in N} F_i = F_{ext}$$

with m being the mass of p and c being the damping constant. N denotes the 1-neighborhood of p. Typically, we restrict ourselves to explicit time integration schemes to avoid solving a nonlinear equation system, as forces F_i depend on the positions of all mass-points. Given a timestep dt, we can easily get the new position for every vertex using the Verlet integration scheme, as it was found to be one of the most efficient scheme with regard to the largest possible timestep in [THMG04]. Then, the new position of the mass-point can be computed as

$$x(t + dt) = \frac{F_{tot}(t)}{m} dt^2 + 2x(t) - x(t - dt).$$

Here, the total resulting force F_{tot} is defined as

$$F_{tot}(t) = F_{ext} - \sum_{i \in N} F_i - c\frac{x(t) - x(t - dt)}{dt}.$$

As force calculations are based on the positions of the last time step, we can calculate the forces F_{tot} for every mass-point of the body in parallel. After accumulating the forces for every mass-point, we can then update the positions of every vertex based on their corresponding forces also in parallel. Note, that after the position update, all springs have changed their length in general, and thus external and internal forces are no longer in balance. This is the reason why it is necessary to choose small integration timesteps to get convergence in a short period of time.
In addition, mass-spring systems are vulnerable to large time steps because of the explicit time integration scheme. With respect to the so-called courant constant, you have very hard constraints for the largest possible time step to achieve stable simulation; this has the drawback, that many time steps per second have to be processed, resulting in a high numerical workload. In the next chapter, we will describe an effective way to use the GPU as a high performance parallel processor in order to exploit the intrinsic parallelism of the approach. As the actual graphics processors generation is fully programmable, we can simulate the physics as well as render the body with its updated positions without need for copying large amounts of data via the bus system.

3.2 Volume Preservation

We want to mention an addition to conventional mass-springs systems to enforce volume preservation. It is well known, that mass-spring simulations cannot preserve volume, as they are not based on volumetric simulation. Therefore, it can (and will) easily happen, that you run into stable states where parts of the model are flipped. Since in that case the springs are no longer stretched, the model is in a totally stable configuration. To avoid such an unwanted behavior, we introduce artificial volume preserving forces as introduced by [LTW95]. We regard the volumetric body as an irregular tetrahedral mesh with vertices used as mass-points and edges interpreted as springs; then, based on its initial configuration, we can establish a rest length for every spring and a rest volume for every tetrahedron. For a single mass-point p, we then collect additional forces for every adjacent tetrahedron based on the actual volume v compared to his rest volume v_0 using an artificial volume stiffness parameter D_v:

$$F_v = D_v(v - v_0)n$$

Here, n is the unit-length normal of the opposite tetrahedral face. As points might move, the normal has to be recalculated in every timestep. Using this technique, we avoid flipped tetrahedral elements, since in that case the volume would be negative. The resulting large force F_v pushes the tetrahedron towards its initial state.

4 GPU Implementation

Early generations of graphics processors were solely optimized for the rendering of lit, shaded and textured triangles. The rendering pipeline was implemented by a set of special-purpose but fixed-function engines, prohibiting the use of such chips in non-rendering

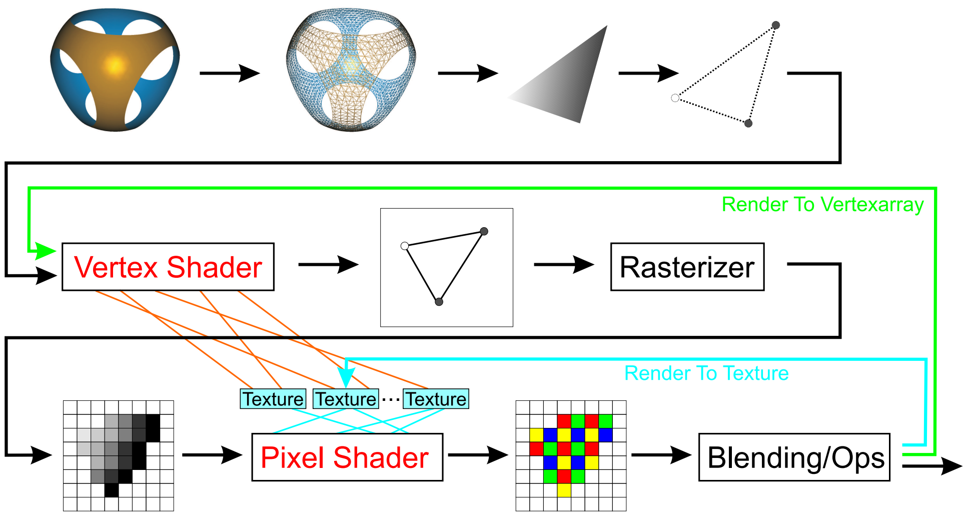

Figure 1: Stages of the programmable graphics pipeline are illustrated.

applications. Nowadays, this design is abandoned in favor of programmable function pipelines that can be accessed via high level shading languages [MGAK03, Mic02].
On current GPUs, fully programmable parallel geometry and fragment units are available providing powerful instruction sets to perform arithmetic and logical operations. In addition to computational functionality, fragment units also provide an efficient memory interface to server-side data, i.e. texture maps and frame buffer objects. Not only can application data be encoded into such objects to allow for high performance access on the graphics chip, but rendering results can also be written to such objects, thus providing an efficient means for the communication between successive rendering passes. Figure 1 gives an overview of the rendering pipeline as it is implemented on current GPUs.
In recent years, a popular direction of research is leading towards the implementation of general techniques of numerical computing on such hardware [HBSL03, BFGS03, KW03]. The results of these efforts have shown that for compute bound applications as well as for memory bandwidth bound applications the GPU has the potential to outperform software solutions. However, this statement is valid only for such algorithms that can be compiled to a stream program, which then can be processed by SIMD kernels as provided on recent GPUs. As we will show, mass-spring systems exhibit this property.

5 Mass-Spring Model

In the current scenario, without loss of generality we restrict the discussion to tetrahedral meshes. To develop an appropriate data structure for this kind of meshes we first have to think about the operations to be performed on this structure.
Thinking in terms of tetrahedral elements, for each of the four corner points a force vector

has to be computed. This vector depends on the elements' edge lengths and volumes as well as the respective rest and hardness values. For each mesh vertex these forces are finally summed up by looping over all tetrahedra.
This strategy, however, suffers from certain peculiarities of current graphics hardware. Because current GPUs have a 128 Bit shader-to-memory interface, only one single 4-component float vector can be written to memory at a time. A shader can write multiple float vectors in one single pass, but this is actually as expensive as writing one vector in multiple passes.
In addition, so called dependent texture fetches slow down the performance considerably. Dependent texture fetches read from a texture address that has been determined from a texture value that was fetched earlier. Such operations essentially prevent the GPU from issuing all texture fetches in parallel, and it is therefore desirable to have as few dependent fetches as possible.
If we investigate the aforementioned approach, it is easy to see that it requires 4 (internal) rendering passes, each pass performing 4 dependent texture fetches as the shader program first has to read the indices of the four corner points. This makes a total of 16 dependent fetches per tetrahedron and simulation step.
A more efficient data structure is based on the observation, that to predict the dynamic behavior of the mass-spring model, in every frame every mesh vertex v must calculate the exerting force vector. This vector is influenced by the one-ring neighborhood around v, i.e. all mesh vertices that are connected to v via a spring. To account for volume preservation, every vertex computes the volume loss or gain of all adjacent tetrahedra and tries to correct this change by an additional displacement. Therefore, access to all tetrahedral elements sharing the center vertex v is required. Because both operations are performed in parallel for every vertex in the mesh, they are perfectly suited for a realization on data parallel stream architectures like GPUs.
To optimally exploit the architecture of recent GPUs (including parallel computations and high memory bandwidth), we have implemented a vertex based data structure. First, a 2D vertex texture is created, which stores mesh vertices in the RGB color components. We then construct a stack of equally sized 2D textures, each of which encodes one of the tetrahedral elements adjacent to the respective vertex in the vertex texture. Tetrahedra are encoded by three references into the vertex texture and a stiffness values, as well as three spring rest lengths and the rest volume of the tetrahedron. These values are stored in a pair of RGBA textures.
Now, on a per-vertex basis we keep track of forces due to compression or stretching by following reference to connected mass points. In a fragment shader, exerted forces are computed for every vertex in parallel, and at every node the resulting forces are accumulated. If the force calculation is not executed per tetrahedron, but per mass point, only three dependent fetches are needed per point, making a total of 12 such operations per tetrahedron and simulation step. Note that the index of one of the points (the current center vertex) is already known and does not need to be fetched. In pseudo code notation the program now looks as follows:

```
for every point p0
    for every tetrahedron t incident on p0

        // via three texture fetches
        get center vertex coordinate p0
        get corner indices i1-3, get element stiffness es
        get rest spring lengths l1-3, get rest volume v,

        // via three dependent fetches
        get coordinates of p1-3 through i1-3

        calculate force on p0
        add to total force on p0
    end for
end for
```

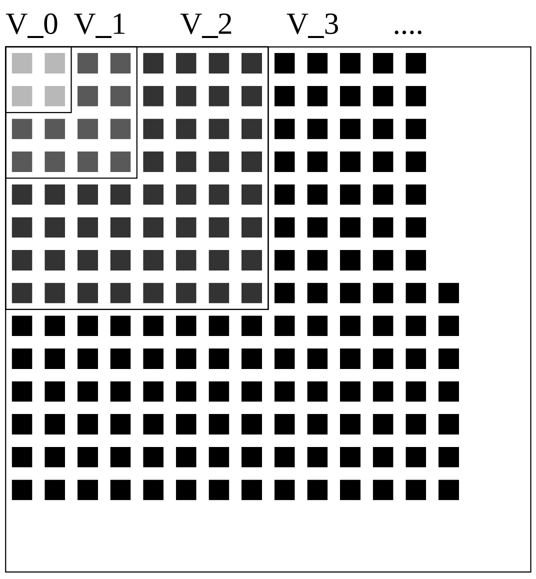

Figure 2: Stack of valence textures. The smallest square contains all points that have highest valence. This results in various valences in range V_0. The next bigger power-of-two square contains all vertices from the smallest square, and is filled with vertices with remaining highest valences. Therefore, range of valence of the new vertices can be determined as V_1. This can be repeated until no vertices are left, resulting in a stack of power-of-two textures.

5.1 Valence Textures

The described data structure has the drawback that the texture stack must be large enough to keep a number of references equal to the maximum valence of any of the mesh vertices. In typical meshes, however, we see a rather inhomogeneous distribution of valences. For instance, in the meshes we have used to demonstrate our approach valences ranging from 6 to 32 can be found. To avoid the memory overhead that is introduced by storing for every vertex as many neighbors as the maximum valence in the mesh, we construct different texture stacks.

Initially, mesh vertices are sorted with respect to valence. Then, we recursively generate stacks of 2D textures at ever decreasing size, which store topology and additional parameters. We build a 2D texture large enough to keep all vertices, and we construct a stack of V_{min} equally sized textures, where V_{min} is the minimum valence of all vertices. These textures are filled with respective references into the vertex texture. Note that the layout of values in these textures is with respect to decreasing valence from top/left to bottom/right (see Figure 2). We then remove all vertices with a valence equal to V_{min} from this texture, and we continue the recursive process with this texture. This procedure is repeated until all vertices have been discarded.

In every animation frame, a set of quadrilaterals covering as many fragments as there are values in the respective stack textures is rendered, and force contributions are computed for every remaining vertex. Already accumulated results are rendered into a texture render target, which can be accessed in upcoming passes to retrieve summed values. In a final pass, the differently sized render targets covering the force vectors of vertices having a particular valence are merged into a texture as large as the vertex texture.

5.2 Rendering Deformable Bodies

In OpenGL, textures are accessed via texture objects encapsulating the raw image data and texture parameters like wrapping or sampling behavior. While this makes textures easy to use it imposes some restrictions on the systems functionality, because of the lack of a direct texture memory access. For instance it was previously not possible to directly render primitives into the texture memory as if it would be the frame buffer. Over time some extensions have been proposed and implemented to diminish this restriction like the render-to-texture extension. However, these extensions are often complex to use and address only special cases thus limiting functionality.

The memory object interface allows the application to allocate graphics memory directly, and to specify how this memory is to be used. This information, in turn, is used by the driver to allocate memory in a format suitable for the requested uses. When the allocated memory is bound to an *attachment point* (a render target, texture, a vertex or color array), no copying takes place. The net effect for the application program therefore is a separation of raw GPU memory from OpenGLs semantic meaning of the data. In our current implementation, a memory object is subsequently bound as the current texture render target and as a vertex array used to hold displaced vertex positions.

On the GPU we construct an indexed vertex array, which references into the vertex texture. In this array, every tetrahedron is represented by 4 triangles. This array remains fixed over

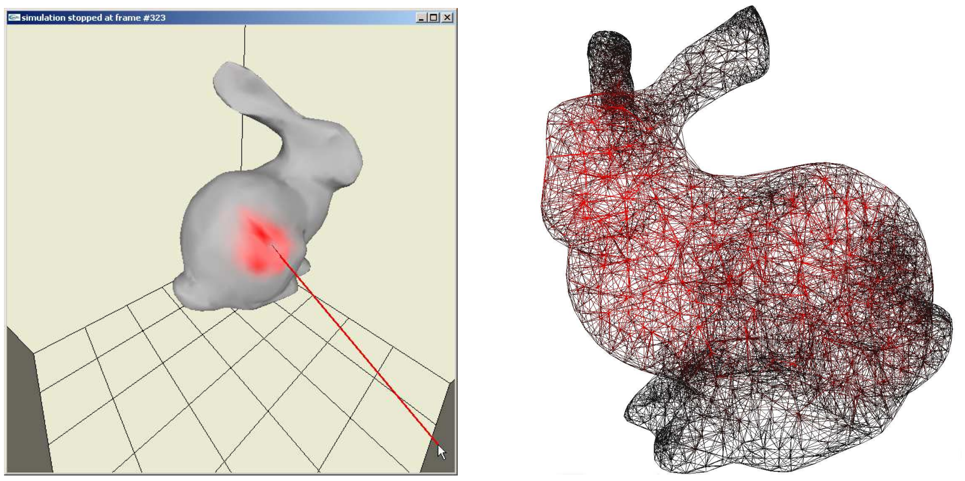

Figure 3: On the left hand side, forces are visualized on the objects surface during user interaction. On the right hand side, forces are visualized on all interior edges, while the bunny is pushed on its snout.

the entire animation and never has to be updated unless the mesh topology is changed. In every animation pass, the vertex array is rendered and the updated vertex texture is issued as coordinate array. Thus, any data transfer between the CPU and the GPU can be avoided, and we can also render the 3D mesh structure.
As an additional rendering option, the strength of the computed displacements can be visualized either on the mesh boundary or on the interior edges (see Figure 3). The mesh boundary is rendered as an additional indexed vertex array, and triangle primitives are lit and shaded. Interior structures are rendered as colored line primitives, because volume rendering of tetrahedral grids is still too expensive as to allow for interactive rates.

6 Discussion and Results

Figure 4 shows deformations on different objects that have been performed using the proposed GPU mass-spring system. As our implementation accelerates both simulation and rendering of the deformed bodies, our timings include the entire simulation. Table 1 shows the timings for differently sized models. The peek performance we achieve on the ATI Radeon X800 is reached with about 84k tetrahedral elements. For larger models, simulation performance basically remains the same, but rendering becomes significantly slower due to increasing geometry load.
When comparing our results to those published in [THMG04], we recognize a speedup of about a factor of 20. Our peek-rate is 8.9 million tetrahedra per second (TPS) compared to 310 thousands TPS (including rendering) reported in Teschner et al. Please note, that – as it is typical for explicit time integration schemes – rendering does not take place in every

model	*tetrahedra*	*computation & rendering time [ms]*	*TPS rating*	*FPS*
Cuboid	5012	2.70	1854440	370
Liver	7536	2.80	2690352	357
Stanford Bunny	9804	2.95	3313752	338
Double Bunny	19608	3.26	6019656	307
Large Bunny	84104	8.26	8999128	121

Table 1: GPU simulation performance values (with rendering) on ATI Radeon X800. As the integration timestep is fixed to 4ms, a framerate of 250 fps or above denotes real-time simulation.

simulation frame, but in every 5th step, so that we achieve a visual update rate of about 50Hz. If only the net simulation time excluding rendering is compared, we are still about a factor of 10 faster.

However, the drawback of our method is that it introduces a significant memory overhead. Neighboring tetrahedra are stored for every vertex separately, hence every tetrahedron is stored 4 times in total. Furthermore, as tetrahedra share common edges, spring forces are calculated multiple times, depending on the valence of the adjacent vertices.

It is also worth noting, that in the case of simple mass-spring systems without volume-preservation, the calculation of spring forces only has to be done twice – one time for every adjacent vertex. Without volume preservation, however, stability can not be guaranteed in general.

Although edge-based data-structures are more memory-efficient, they introduce other problems. In particular, because in a final step the forces have to be accumulated per vertex, respective floating point values have to be blended. However, floating point blend operations are currently not supported in full precision and hence, such scatter operations cannot be implemented efficiently. To overcome this burden, memory-intensive data structures have to be used. In a first pass you can calculate spring-forces for every edge and store them in an edge force texture. As you need to calculate volume-preserving forces for every tetrahedron, you need a tetrahedra-based data structure too. A second pass would combine spring forces fetched from the edge force texture with volume-preservation forces for every tetrahedron. The result is a large texture containing 4 force vectors per tetrahedron. As the shader-to-memory interface is limited to 128 Bit, we have to use 4 passes to write the tetrahedra force texture, which slows down the overall performance. Additionally, you have to use several passes to sum up all forces that correspond to a single vertex using a vertex based datastructure as described before, that contains only indices in the tetrahedra-based force texture.

7 Conclusion

In this paper, we have demonstrated a physical based simulation engine that is exclusively realized on the GPU. Accumulation of forces and time integration are performed in fragment shader programs, thus exploiting the computing power of current GPUs. Displaced vertices are stored in OpenGL memory objects, which can subsequently be interpreted as textures or vertex arrays. Hence we can render the deformed object directly, without the need to download the data to the CPU and sending them again to the GPU for rendering purposes. Using these techniques, we achieve a significant performance gain compared to CPU implementations of mass-spring systems.

Acknowledgements

We would like to thank ATI for providing the Uberbuffer extension, and in particular Mark Segal for providing information about the technical details of recent ATI cards.

References

[BFA02] Robert Bridson, Ronald Fedkiw and John Anderson. *Robust treatment of collisions, contact and friction for cloth animation*. In: Proceedings of the 29th annual conference on Computer graphics and interactive techniques, pp. 594–603. ACM Press, 2002.

[BFGS03] J. Bolz, I. Farmer, E. Grinspun and P. Schröder. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. In: ACM Computer Graphics (Proc. SIGGRAPH ’03), pp. 917–924, 2003.

[BNC96] M. Bro-Nielsen and S. Cotin. *Real-time volumetric deformable models for surgery simulation using finite elements and condensation*. In: Proceedings of Eurographics ’96, 1996.

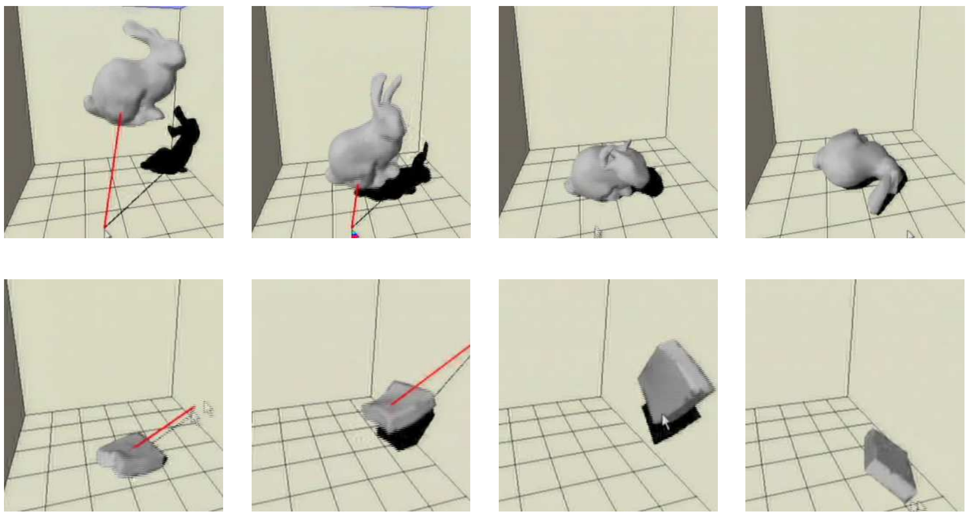
Figure 4: Interactive deformations of the bunny and the cuboid model.

[DDBC01] 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.

[DSB99] M. Desbrun, P. Schröder and A. Barr. *Interactive animation of structured deformable objects*. In: Graphics Interface, pp. 1–8, 1999.

[FGL03] A. Fuhrmann, C. Gross and V. Luckas. *Interactive Animation of Cloth Including Self Collision Detection*, 2003.

[Gel98] Allen Van Gelder. *Approximate simulation of elastic membranes by triangulated spring meshes*. J. Graph. Tools, 3(2):21–42, 1998.

[GM97] S. F. Gibson and B. Mirtich. *A survey of deformable models in computer graphics*. In: Technical Report TR-97-19, Mitsubishi, 1997.

[HBSL03] M.J. Harris, W.V. Baxter, T. Scheuermann and A. Lastra. *Simulation of Cloud Dynamics on Graphics Hardware*. In: Proceedings ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 12–20, 2003.

[KW03] J. Krueger and R. Westermann. *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*. In: ACM Computer Graphics (Proc. SIGGRAPH ’03), pp. 908–916, 2003.

[LTW95] Y. Lee, D. Terzopoulos and K. Waters. *Realistic modeling for facial animation*. In: Proceedings of SIGGRAPH ’95, 1995.

[MDM$^+$02] M. Müller, J. Dorsey, L. McMillan, R. Jagnow and Barbara Cutler. *Stable real-time deformations*. In: Proceedings of the ’02 ACM SIGGRAPH/Eurographics symposium on Computer animation, pp. 49–54, 2002.

[MGAK03] W. Mark, R.S. Glanville, K. Akeley and M. Kilgard. *Cg: A system for programming graphics hardware in a C-like language*. In: ACM Computer Graphics (Proc. SIGGRAPH ’03), pp. 896–907, 2003.

[Mic02] Microsoft. *DirectX9 SDK*. http://www.microsoft.com/DirectX, 2002.

[MMDJ01] 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.

[Pro95] Xavier Provot. *Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior*. In: Wayne A. Davis and Przemyslaw Prusinkiewicz, ed., Graphics Interface ’95, pp. 147–154. Canadian Human-Computer Communications Society, 1995.

[RNP01] A. Radetzky, A. Nrnberger and D. P. Pretschner. *Simulation of elastic tissue in virtual medicine using neuro-fuzzy system*. In: Proceedings of Medical Imaging ’98 (SPIE Proceedings Volume 3335), 2001.

[THMG04] Matthias Teschner, Bruno Heidelberger, Matthias Mueller and Markus Gross. *A Versatile and Robust Model for Geometrically Complex Deformable Solids*. In: Proceedings of Computer Graphics International 2004, pp. 312–319, 2004.

[WDGT01] X. Wu, M. S. Downes, T. Goktekin and F. Tendick. *Adaptive Nonlinear Finite Elements for Deformable Body Simulation Using Dynamic Progressive Meshes*. In: Proceedings of Eurographics ’01, pp. 349–358, 2001.

[ZC99] Y. Zhuang and J. Canny. *Real-time Simulation of Physically Realistic Global Deformation*. In: IEEE Vis ’99, 1999.