# Mass-Spring Systems on the GPU

Joachim Georgii [a,*] , Rüdiger Westermann [a]

[a] *Computer Graphics & Visualization Group, Technische Universität München* [1]

**Abstract**

We present and analyze different implementations of mass-spring systems for interactive simulation of deformable surfaces on graphics processing units (GPUs). 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 on the GPU, saving displaced point masses in graphics memory, and then sending these positions through the GPU again to render the deformed surface. Two different simulation algorithms implementing scattering and gathering operations on the GPU are compared with respect to performance and numerical accuracy. We discuss GPU specific issues to be considered in simulation techniques showing similar computation and memory access patterns to mass-spring systems.

*Key words:* Physics-based simulation, GPU simulation, mass-spring systems

## 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 the system dynamics, the geometric representation of the system has to be modified according to the computed motion. In every simulation frame, geometry has to be updated, and the data structure used by the simulation engine has to be converted into a suitable format for rendering. If

* Corresponding author.
*Email addresses:* georgii@in.tum.de (Joachim Georgii), westermann@in.tum.de (Rüdiger Westermann).

[1] http://wwwcg.in.tum.de

the simulation is carried out on the CPU, displaced geometry has to be sent to the GPU for rendering. With the ability to do more simulation steps per time interval, the bandwidth required will grow substantially thus prohibiting real-time or even interactive rates.

In this paper, we present and analyze different implementations of a mechanical system as described on recent GPUs. Although we concentrate the discussion on mass-spring models as they typically arise in medical and engineering simulations, the concepts we propose can also be employed in other applications. In many applications, despite different rules to update every part of the system, retrieval and evaluation of adjacent states is an intrinsic mechanism.

In particular, we have implemented a mass-spring system based on triangular mesh structures. Edges are treated as springs connecting pairs of mass points. Under the influence of external forces, e.g. forces exerted by user interaction, gravity, or collision, 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 apply forces, seeking to preserve their rest length when compressed or stretched.

At the core of this implementation we have developed two different data structures amenable to the kind of operations performed. The first one is point-centric and can be updated by gathering information from adjacent parts, the second one is edge-centric and uses scattering to update the position of mass points. Both data structures are compared to each other with respect to memory requirement, performance and numerical accuracy.

## 2 Modeling

We assume the surface to be deformed is modelled by a triangulation of discrete mass points. Since triangle edges are considered as springs between these points, surface patches are prohibited from degenerating to a line. Springs can rotate arbitrarily, and the forces they exert on connected points are obtained from Hooke‘s law

$$F_{ij} = F_{ij}(x_i, x_j) = D_{ij}\frac{|l_{ij}| - |l_{ij}^0|}{|l_{ij}|} \cdot l_{ij}. \tag{1}$$

Here, $D_{ij}$ describes the stiffness of the spring connecting points $p_i$ and $p_j$, and $l_{ij}$ is the distance between these points. The rest length of the spring in its initial configuration is denoted $|l_{ij}^0|$. For every mass point, forces exerted by all connected springs have to be accumulated. These forces should balance the external forces $F_{ext}^i$ acting on a single mass point. As external forces are exerted continually, the balance between internal and external forces has to be achieved dynamically.

In the current implementation, positions of mass points, $x_i$, are updated with respect to their velocity and acceleration using the Lagrangian law of motion

$$m_i \ddot{x_i} + c \dot{x_i} + \sum_{j \in N_i} F_{ij}(x_i, x_j) = F_i^{ext} \tag{2}$$

with $m_i$ being the mass of $p_i$ and $c$ the damping constant. $N_i$ denotes the 1-neighborhood of $p_i$. As forces $F_{ij}$ depend on the positions of all mass points, a non-linear system of equations has to be solved in general. To avoid this, we restrict ourselves to explicit time integration. Given a time step $dt$, for every point its new position is calculated using Verlet integration. As this scheme does not require point velocities to be explicitely calculated or stored, the current velocity is always consistent with the current point position. New point positions $x_i$ can then be computed as

$$x_i(t + dt) = \frac{F_i^{tot}(t)}{m_i} dt^2 + 2\, x_i(t) - x_i(t - dt),$$

where the total force $F_i^{tot}$ is computed as

$$F_i^{tot}(t) = F_i^{ext} - \sum_{j \in N_i} F_{ij} - c \frac{x_i(t) - x_i(t - dt)}{dt}.$$

As the force calculation is solely based on point positions at the current time step, forces $F_i^{tot}$ as well as updated point positions can be computed in parallel. Since the position update affects all springs in general, external and internal forces are no longer in balance. This results in a dynamic behavior of the system. To let the system converge, a reasonably small integration time step satisfying the Courant condition has to be chosen.

In chapter 4 we will describe two different approaches to exploit parallelism and memory bandwidth on recent GPUs for the dynamic simulation of a mass-spring system as described. On such architectures, not only can the simulation of such systems be carried out efficiently but the deformed surface can be directly rendered without any data read-back to the CPU.

## 3 GPU Architecture

On current GPUs, fully programmable parallel geometry and fragment units are available, which can be accessed via high level shading languages (1; 2). These units provide 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.

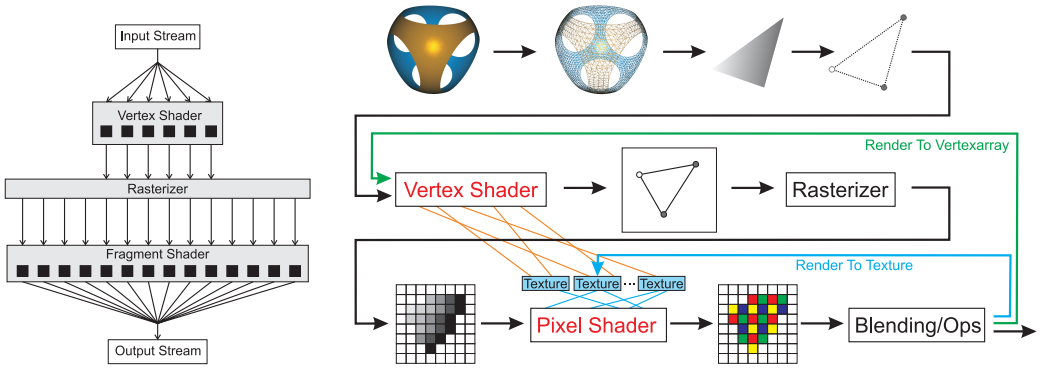


Fig. 1. Conceptual overview of the programmable graphics pipeline.

In this work we employ OpenGL Superbuffers on the ATI X800 XT graphics card to subsequently interpret a contiguous block of video memory as a 2D texture map, a texture render target or a vertex array (3). In particular this mechanism allows us to send data computed in the fragment units through the GPU again to render geometric primitives without any read-back to CPU memory. Figure 1 gives an overview of the rendering pipeline as it is implemented on current GPUs.

In recent years, a popular direction of research has lead towards the implementation of general techniques of numerical computing on such hardware. A comprehensible collection of research papers in this particular area can be found in (4; 5). 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.

## 4 GPU Implementation

Typically, the implementation of a mass-spring system is performed in the following steps:

(1) Calculation and accumulation of spring forces at mass points
(2) Time integration of mass points
(3) Update of point positions

As the first step requires adjacent points to be accessed, a GPU data structure able to efficiently perform this kind of operation needs to be developed. In the following we will investigate the use of two different data structures in this particular scenario – point-centric and edge-centric. Depending on which data structure is used, mass-spring simulation is implemented as shown below.

**Point-centric**

```
for all mass points i do
   initialize total force F_i


   // Gather force
   for all neighboring mass points j do
      calculate spring force F_ij
      add F_ij to total force F_i
   endfor

   update vertex position x_i
endfor
```

**Edge-centric**

```
for all mass points i do
   initialize total force F_i
endfor
for all springs ij do
   calculate spring force F_ij
   // Scatter force to incident mass points
   add F_ij to total force F_i of left mass point
   add F_ij to total force F_j of right mass point
endfor
for all mass points i do
   update vertex position x_i
endfor
```

The major difference is the way spring forces are calculated. In the point-centric approach, for every mass point adjacency information is gathered. In the edge-centric approach, every edge computes its spring force only once and scatters this force to the mass points it connects.

### *4.1 Point-centric approach (PCA)*

In the point-centric approach, a surface point needs to maintain its current and last position for time integration, its mass as well as references to all adjacent points including spring stiffness and rest length. Furthermore, the memory requirement for each vertex is not constant and depends on its valence (the number of incident edges).

On the GPU, per-point attributes and references are stored in equally sized 2D texture maps. We store references into a 2D texture in a single float component, and we use shader arithmetic to decode the appropriate 2D texture coordinates. A stream of as many fragments as there are points is generated by rendering a view plane aligned quadrilateral that covers exactly this number of pixels. In the fragment units the edge-centric algorithm is carried out by fetching attributes of adjacent points from the respective texture maps. Forces are then calculated and used to integrate to the next position. These positions are written to an additional render target, which becomes the container of point coordinates in the following simulation pass.

Using the proposed data structure, two different realizations of mass-spring simulation on the GPU are possible. First, if all vertices have the same valence, all computations necessary to update mass point positions can be performed in one rendering pass. Only if the valence exceeds the number of available texture units the execution has to be broken into multiple passes. Second, if the valence is not constant the computation has to be split into multiple rendering passes. To avoid processing of points that have no further neighbor, a particular texture layout can be employed (6). In this layout, with increasing valence points are stored within an ever smaller rectangular sub-area of the entire 2D texture. The application program renders an appropriately sized quadrilateral to produce exactly the same number of fragments as there are points not yet updated. This approach requires multiple rendering passes, which communicate their results via an additional render target. Moreover, in

every single pass the position of the center mass point has to be read, which increases the number of texture fetches to be performed.

The point-centric approach comes at the expense of calculating each spring force twice, as every spring is incident to two mass points. Moreover, the data structure becomes very inefficient for large valences. As it requires as many textures as the maximal valence of the mesh, the number of memory access operations as well as the amount of memory to be kept can quickly become the bottleneck of the simulation. In particular because texture fetches to adjacent mass points are dependent fetches, due to the fact that references are stored in texture maps, for typical meshes this approach exhibits significant drawbacks.

### *4.2 Edge-centric algorithm (ECA)*

An edge-centric data structure overcomes the aforementioned drawbacks of a point-centric approach. For every edge, references to both incident points as well as spring stiffness and rest length are stored in an appropriately sized *edge texture*. In a first rendering pass, spring forces are calculated and rendered into a texture target – the *force texture*. As there are three times more edges than points in a triangulation, this texture is different in size than the texture keeping point coordinates – the *point texture*. The problem now becomes to scatter the computed forces to the respective points, and for every point to accumulate the received contributions. Such an operation is not supported on recent GPUs, and gathering the forces using a point-centric data structure results in the same problems as described.

To enable GPU scattering, we harness the power of vertex processing. For every edge, a point primitive is rendered twice into a render target that is equal in size than the point texture. First, primitives are rendered at the respective position of the left mass point of every edge. In the second pass, the target position is at the entry of the right mass point in this texture. As both target positions are already stored in the edge texture, this structure can directly be rendered as server-side vertex array. In a vertex shader program, references stored are decoded into appropriate point coordinates to be rasterized at the respective position in the render target. In every pass, the force texture is used as additional color array. In the second pass, forces are negated in a fragment shader before they are combined using accumulative blending in the current render target. In this way, multiple points are rendered into the same entry of a point-centric render target, which finally stores the force per mass point.

Independent of the points' valences in the mesh, the edge-centric approach only requires four rendering passes. The first three passes operate as described, and in the fourth pass the time integration of mass point positions is performed using computed per-point forces. In addition, forces are only computed once for each edge, reducing the number of arithmetic operations and texture fetches to be performed.

## 5 Discussion and results

We now start the evaluation of the proposed data structures in the particular scenario. Memory requirements as well as the number of texture fetches and arithmetic operations are compared. We distinguish between texture fetches and dependent texture fetches, the latter being dependent on the result of an earlier texture lookup. Such fetches are known to be a potential bottleneck in GPU applications, as the pipeline has to be stalled until the result of the first texture fetch is available.

In the following, $N$ denotes the maximal valence of a vertex in the triangulation. $n_v$ and $n_e$ are the number of vertices and edges, respectively. For regular meshes, $n_v$ and $n_e$ are related as $N = 2n_e/n_v$. In the general case, $N$ can be significantly larger.

Table 1 shows the statistics for the point-centric and the edge-centric approach. In PCA, for every mass point and every adjacent point the reference to this point, the stiffness of the spring connecting both points, and the springs rest length are encoded in a RGB texel. In ECA, two references to the points connected by the spring, spring stiffness, and rest length, are encoded in one RGBA quadruple. While forces are stored in a RGB texture map, a RGBA texture is used to keep each point position and mass. Throughout the discussion we do not account for the overhead introduced by the Verlet time integration, as it adds the same additional expense to both approaches, i.e. $2n_v$ texture fetches to get the current and the previous point position and about 10 arithmetic operations to perform the integration.

On our target architecture, a P4 3 GHz processor equipped with an ATI X800 XT card, we can render about 240 million point coordinates per second from the server-side edge texture as described. Even for the largest mesh we consider in our investigations, consisting of $512^2$ vertices, this throughput allows us to perform GPU scattering in ECA about 480 times per second. As will be shown below, this time is justifiable compared to the time required by force calculation and time integration.

For a regular mesh of valence 6 as shown in Figure 2, the PCA requires $n_v + n_e$ more texture fetches than ECA at the same number of dependent fetches. In addition, the number of operations to be performed in the fragment units is slightly increased. The memory requirement of ECA, on the other hand, is

| | mem. RGB | mem. RGBA | tex. fetches | dep. tex. fetches | ops |
|---|---|---|---|---|---|
| PCA | $N \cdot n_v$ | $n_v$ | $n_v + N \cdot n_v$ | $N \cdot n_v$ | $10\ N \cdot n_v$ |
| ECA | $n_e$ | $n_e + n_v$ | $n_e$ | $2\ n_e$ | $14\ n_e$ |

Table 1
Comparison of memory requirement, texture fetches and arithmetic operations for PCA and ECA.

| | force calculation | force accumulation | Verlet integration | total |
|---|---|---|---|---|
| PCA $128^2$ | 0.54ms | | 0.20ms | 0.74ms |
| ECA $128^2$ | 0.40ms | 0.12ms | 0.20ms | 0.72ms |
| PCA $256^2$ | 2.34ms | | 0.74ms | 3.08ms |
| ECA $256^2$ | 1.76ms | 0.52ms | 0.74ms | 3.02ms |
| PCA $512^2$ | 9.82ms | | 3.18ms | 13.0ms |
| ECA $512^2$ | 7.34ms | 2.08ms | 3.18ms | 12.6ms |

Table 2
Performance comparison between PCA and ECA. All timings are measured for a regular mesh with valence 6.

slightly higher compared to PCA. This is due to the force texture needed to store the result of the first pass. As can be seen in Table 2, due to the lower number of arithmetic and memory access operations, even for regular meshes exhibiting rather low valence ECA outperforms PCA in terms of run-time.

For irregular meshes, on the other hand, the benefits of ECA will grow substantially, as it does not depend on the maximal valence of the mesh. With increasing valence, both memory requirements and texture fetch operations of PCA will increase as well. Moreover, a potentially large number of rendering passes has to be performed. Even if an optimized texture layout is employed to minimize the number of fragments to be processed, this texture cannot be packed densely in general and thus introduces some overhead in the current application. For example, PCA performs 1.6 times slower in total for a mesh with valences in the range from 3 to 12.

The most crucial limitation of ECA in the current scenario is with respect to additive blending in the render target that is used to accumulate the force contributions. As 32 Bit floating point blending is currently not supported on any GPU, force accumulation was performed inadequately in 8 Bit fixed point precision. In contrast to PCA, where force accumulation is carried out in the fragment shader with 24 Bit floating point precision, numerical precision is therefore a problem in ECA. However, next generation graphics cards will overcome this limitation, as they might allow for a read-back from the current render target or a full precision floating point blending.

Let us conclude the discussion with some remarks concerning collision detection on the GPU. As has been shown in previous work (7; 8) collisions between moving particles and simple polygonal objects or objects described analytically can be determined quite efficiently on the GPU. Self collisions, on the other hand, have not yet been considered. In our current implementation (see Figure 3 and 4) we have integrated the binning approach for approximative self collision detection proposed in (9). With respect to a partitioning of 3D space using a regular grid, every mass point gets assigned the unique identi-

fier of the cell containing this point. A GPU sorter is employed to generate a texture map in which points contained in the same partition are most likely to be stored in adjacent elements. By using multiple staggered space partitions, the majority of self collisions can be detected. For large integration time steps or many points within the same cell, however, the number of missed collisions increases significantly.

## 6 Conclusion

In this paper, we have presented a physics-based simulation engine for deformable surfaces that is exclusively realized on the GPU. We have developed and analyzed different implementations built upon point- and edge-centric data structures. While the computation and accumulation of forces is performed in the fragment units of recent GPUs, both data structures distinguish in the way computed force contributions are derived in the fragment shader. With respect to performance it was shown, that even for regular meshes an edge-based approach is superior to a point-based approach. Furthermore, such an approach greatly simplifies the GPU implementation and requires less coding effort. For non-regular meshes as they typically arise in adaptive triangulations, the benefits of edge-based data structure turn out much more dominantly.

The major drawback of an edge-centric data structure is with respect to numerical precision. In the simulation of mass-spring models, where the dynamic range of exerting forces is very high, accumulation of forces using 8 Bit internal format is not sufficient. Although 16 Bit floating point precision on recent Nvidia cards gives visually accurate results, from the numerical point of view only full 32 Bit floating point blending will produce appropriate results.

Independent of the data structure used, displaced point coordinates are stored in 2D texture maps, and they can be displayed directly without any readback to CPU memory using OpenGL memory objects. As a matter of fact, in applications where the system dynamics should directly be visualized, a GPU implementation comes at an additional advantage.

## References


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

[2] Microsoft, Directx9 sdk, http://www.microsoft.com/DirectX (2002).

[3] ATI, Superbuffers OpenGL Extension, www.ati.com/developer/gdc/SuperBuffers.pdf (2004).

[4] M. Harris, General-purpose computing using graphics hardware, http://www.gpgpu.org/.
[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. Purcell, A survey of general-purpose computation on graphics hardware, in: Eurographics 2005, 2005.
[6] J. Georgii, F. Echtler, R. Westermann, Interactive simulation of deformable bodies on GPUs, in: Proceedings of Simulation and Visualization 2005, 2005.
[7] A. Kolb, L. Latta, C. Rezk-Salama, Hardware-based simulation and collision detection for large particle systems, in: SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2004.
[8] Nvidia, Sdk white paper: Cloth, http://developer.nvidia.com/ (2005).
[9] P. Kipfer, M. Segal, R. Westermann, Uberflow: A GPU-based particle engine, in: Proceedings Eurographics Graphics Hardware Conference, IEEE, 2004.

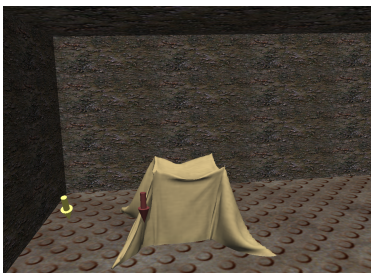

Fig. 2. A cloth patch fixed on 4 points under influence of gravity.

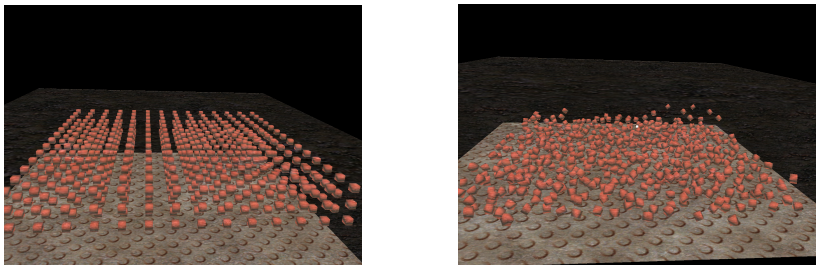

Fig. 3. Demonstration of the approximative self collision detection.

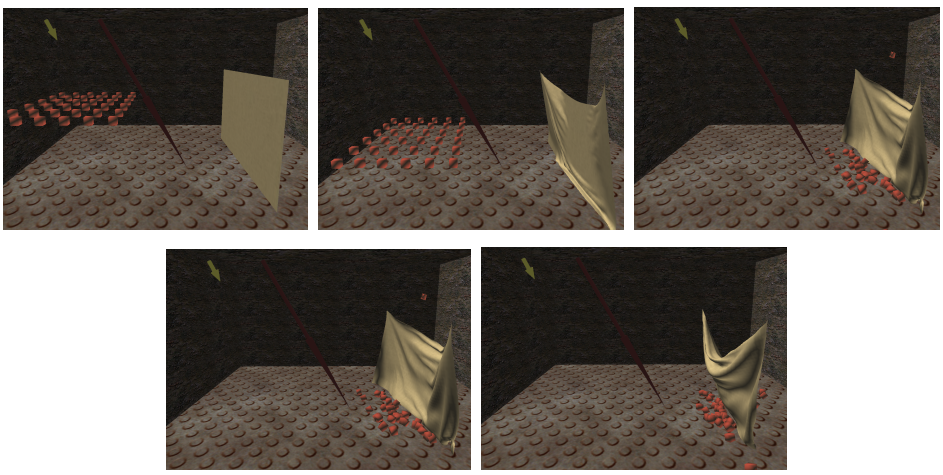

Fig. 4. Interaction of different objects in an example scenario.