Algorithm and Data Structures for Implementing a Mass-spring Deformable Model on GPU

Tzvetomir I Vassilev, Roumen I Rousev

Abstract: *The current paper proposes data structures and an algorithm for implementing a general mass-spring system on the modern programmable graphics processors. A mass-spring cloth model was implemented to run on GPU using the GL shading language and performance of the CPU and GPU were compared. Results are given at the end of the paper.*

Key words: *GPU programming, Mass-spring systems.*

INTRODUCTION

Mass-spring systems have been widely used by researchers in computer graphics for modelling different deformable objects. Many scientists have utilised it for cloth modelling [1, 2, 5, 7, 8]. The main applications of cloth simulation are in fashion design industry and in electronic commerce when customers shop for garments on the web and try them on in a virtual booth. However mass-spring models were also exploited for simulating volume preservation solids [6] and other elastic deformable objects [8].

The graphics processor (GPU) on today's commodity video cards has evolved into an extremely powerful and flexible processor [4]. The latest graphics architectures provide tremendous memory bandwidth and computational horsepower, with fully programmable vertex and pixel processing units that support vector operations up to full IEEE floating point precision. High level languages have emerged for graphics hardware, making this computational power accessible. Architecturally, GPUs are highly parallel streaming processors optimized for vector operations, with both multiple instruction on multiple data (MIMD) (vertex) and single instruction on multiple data (SIMD) (pixel) pipelines. Not surprisingly, these processors are capable of general-purpose computation beyond the graphics applications for which they were designed.

The aim of this work is to design suitable data structures and an algorithm for implementing a general mass-spring model on modern GPUs with the purpose of speeding the simulation up. The rest of the paper is organized as follows. The next section addresses related work. Section 3 gives some details about using the GPU for general purpose computations, section 4 explains our implementation, section 5 presents results of the performance tests, and the last section concludes the paper.

RELATED WORK

Modelling cloth and other deformable objects

Methods to model cloth for computer graphics have been investigated for more than two decades. Mass-spring particle systems are mainly used [2, 5, 7] while some employ finite element methods [8]. Provot [5] introduced a simple mass-spring topology (see Figure 1) which is commonly used owing to its efficiency and simplicity. He used linear (Hook) springs and applies explicit Euler integration. To account for super-elongation, caused by the linear springs, he constrains particles' positions in a post correction step so that springs can not extend above a certain threshold: usually 5-10% of their natural length, depending on the material properties to be simulated. Repositioning according to the length constraint of one spring, however, can lead to over-elongation of other springs and may require several iterations of the post correction steps to resolve. Vassilev et al [7] improved this by modifying the particles' velocities instead of their positions.

The elastic model of cloth is a mesh of $l \times n$ mass points, each of them being linked to its neighbours by massless springs of natural length greater than zero. There are three different types of spring:

- Springs linking vertices [i, j] with [i+1, j], and [i, j] with [i, j+1] are called “stretch” springs;

• Springs linking vertices [*i*, *j*] with [*i*+1, *j*+1], and [*i*+1, *j*] with [*i*, *j*+1] are called "shear" springs;

• Springs linking vertices [*i*, *j*] with [*i*+2, *j*], and [*i*, *j*] with [*i*, *j*+2] are called "bend" springs.

As the names indicate, the first type of spring implements resistance to stretching, the second – to shearing and the third – to bending.

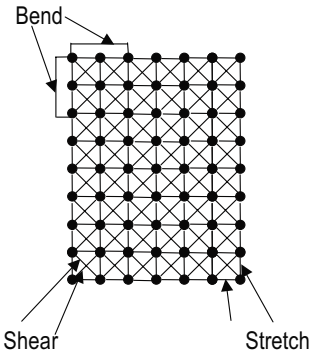

Fig. 1. Spring types in the cloth model

Let $\mathbf{p}_{ij}(t)$, $\mathbf{v}_{ij}(t)$, $\mathbf{a}_{ij}(t)$, where i=1,…,I and j=1,…,n, be respectively the positions, velocities, and accelerations of the mass points at time t. The system is governed by the basic Newton's law:

$$\mathbf{f}_{ij} = m_{ij}\,\mathbf{a}_{ij}, \tag{1}$$

where m_{ij} is the mass of point ij and $\mathbf{f}_{ij}$ is the sum of all forces applied at point ij. The force $\mathbf{f}_{ij}$ can be divided in two categories.

Internal forces arise from the tensions of the springs. The overall internal force applied at the point ij is a result of the stiffness of all springs linking this point to its neighbours:

$$\mathbf{f}_{int}(\mathbf{p}_{ij}) = -\sum_{kl} k_{ijkl}\left((\mathbf{p}_{kl}-\mathbf{p}_{ij}) - l^{0}_{ijkl}\frac{\mathbf{p}_{kl}-\mathbf{p}_{ij}}{\|\mathbf{p}_{kl}-\mathbf{p}_{ij}\|}\right), \tag{2}$$

where k_{ijkl} is the stiffness of the spring linking ij and kl, and l^{0}_{ijkl} is the natural length of the same spring.

The **external forces** can differ in nature depending on what type of simulation we wish to make. The forces most frequently included are:

- Gravity: $\mathbf{f}^{gr}_{ij} = m\mathbf{g}$, where $\mathbf{g}$ is the gravity acceleration;
- Viscous damping: $\mathbf{f}^{vd}_{ij} = -C_{vd}(\mathbf{v}_i - \mathbf{v}_j)$, where C_{vd} is a damping coefficient,
- Collision response.

From the above we may compute the force $\mathbf{f}_{ij}(t)$ applied to point ij at any time t. The fundamental equations of Newtonian dynamics can be integrated over time by a simple Euler method:

$$\begin{aligned}
\mathbf{a}_{ij}(t+\Delta t) &= \frac{1}{m_{ij}}\mathbf{f}_{ij}(t)\\
\mathbf{v}_{ij}(t+\Delta t) &= \mathbf{v}_{ij}(t) + \Delta t\,\mathbf{a}_{ij}(t+\Delta t),\\
\mathbf{p}_{ij}(t+\Delta t) &= \mathbf{p}_{ij}(t) + \Delta t\,\mathbf{v}_{ij}(t+\Delta t)
\end{aligned} \tag{3}$$

where Δt is a chosen time step. The Euler Equations 3 are known to be very fast and to give good results, provided the time step Δt is less than the natural period of the system $T_0 \approx \pi\sqrt{m/K}$, where K is the highest stiffness in the system. Numerous recent works in cloth simulation, see for example [1, 2], have shown that improvements in stability are possible by using implicit integration. However, for complex garments with mapping of KES measurements to the spring properties, explicit integration still proved to be beneficial in terms of efficiency in our case [7]. The advantages of Euler integration became particularly apparent when computation of the collision detection and response, which require small time steps, were taken into consideration. Similar results were also indicated by Volino and Magnenat-Thalmann [9].

The deformable solids [6] are based on a similar mass-spring system. All surface vertices of the solid are connected to each other with regular springs, similar to the stretch springs of the cloth. However there are additional "support" springs, which connect the surface points to the solid centre. The external forces are similar to those of the cloth model. The system of equations is again integrated using the Euler equations (3).

To summarise most of the mass-spring systems are represented by a grid of mass points connected by different types of spring. Usually three arrays are used for storing the forces, velocities and positions of each mass point, while the springs are stored in a separate array. Each spring is a structure with the following components: indices of the two vertices it connects, natural length, and spring stiffness.

Simulation algorithm on the CPU

One iteration of the simulation algorithm, which is in fact one integration step, is shown below.

Algorithm 1. Mass-spring integration on the CPU

```
For each spring
   Compute internal forces
End for
For each mass point
   Add external forces
   Compute velocity
   Do collision detection and response (modify velocity)
   Compute new position
End for
```

The algorithm has two stages. During the first one the forces acting at the two ends of each spring are computed, applying the Hook's law, and added to the forces of the two mass points. The second loop goes through each vertex, adds external forces (damping, gravity) and computes new velocities and new positions.

GENERAL PURPOSE COMPUTATION ON GPU

General Purpose computation on GPU (GPGPU) has become a fashionable topic among computer graphics people. There is even a web site www.gpgpu.org where history, events and news in this area are regularly published. Currently there are two main producers of graphics cards that offer programmable GPU – ATI and NVIDIA.

API for GPGPU

NVIDIA's CUDA (compute unified device architecture) is GeForce 8 Series' API for GPGPU programming. As it is not a graphic based API it has several less constraints for programs as well as cleaner code. AMD's CTM (close to the metal) is an approach that enables low-level efficient GPU programming without any graphics overhead. However, these are producer oriented.

Usually the GPGPU programmers must use graphics APIs: OpenGL and Direct3D (DirectX). OpenGL tends to be favoured in the academic community due to the platform portability it allows and due to its extension mechanism, which lets vendors add new features to the API as soon as the hardware supports those features (rather than waiting on Microsoft to release a new version of DirectX). DirectX/Direct3D, on the other hand, tends to be favoured in the computer game industry, where dependence on Windows is not a particular impediment.

A few GPGPU-friendly streaming languages, such as Brook, Sh, and Microsoft's Accelerator have been developed to insulate developers from the graphics APIs as much as possible. Brook is actively supported in the GPGPU.org Forums. Sh has evolved into a commercial effort (with a very unrestricted academic license) called RapidMind, targeting multicore CPUs, the Cell and GPUs with one programming model.

As we wanted our implementation to be more general, platform independent and work on both NVIDIA and ATI, it was developed with OpenGL. A good tutorial how to program GPUs with OpenGL is published by Göddeke [3].

Data on the GPU

One-dimensional arrays are the native CPU data layout. Higher-dimensional arrays are typically accessed by offsetting coordinates in a large 1D array. An example for this is the row-wise mapping of a two-dimensional array a[i][j] of dimensions M and N into the one-dimensional array a[i*M+j], assuming array indices start with zero.

The native data layout for the GPUs is a two-dimensional array. One- and three-dimensional arrays are also supported, but they either impose a performance penalty or cannot be used directly. Arrays in GPU memory are called **textures** or texture samplers. Texture dimensions are limited on GPUs, the maximum value in each dimension can be queried. On modern cards the maximum dimensions are 2048x2048 or 4096x4096. On the CPU, we usually use *array indices*, on the GPU, we will need *texture coordinates* to access values stored in the textures. The GPUs work on four channels of data simultaneously corresponding to red, green, blue and alpha (RGBA). However, one can use fewer channels: one or three.

Typically the graphics cards support textures with dimension constrained to powers of two, for example 512 by 256, which is usually called a texture2D. Some cards also support rectangular textures with arbitrary dimensions. In our case we wanted the system to be more general, so we use texture2D. Current graphics cards do not support double precision floating point values, so we will use 32 bit IEEE floats stored in the textures.

In order to do computations on the GPU first one has to create data in the main memory, transfer values to textures in the graphics card memory, perform computations there and read back results. The usual approach is to create a frame buffer object (FBO) in the graphics card, set it as a default render target and draw pixels there. After the computations are finished the pixels can be read back from the buffer to the computer primary memory.

Shaders

The programmes that run on a GPU are called shaders. They are either vertex or fragment (pixel) shaders depending on whether they compute vertexes or pixels. In our case we are talking about fragment shaders, which perform operations on the textures. There is a fundamental difference in the computing model between GPUs and CPUs. If we want a vector operation on the CPU we must use a loop. Since the GPUs are parallel processors, called vector (array) processors, which can perform a single instruction on multiple data, one does not need the loop at all! So, the programmer has to specify only the computational kernel inside the loop. The programmable part of the GPU is called a *fragment pipeline* and consists of several parallel processing units. The hardware and driver logic however that schedules each data item into the different pipelines is not programmable! So from a conceptual point of view, all work on the data items is performed

independently, without any influence among the various fragments moving through the pipeline.

There are two programming languages that can be used to write shaders with OpenGL. The one is Cg and the second is GL shading language (GLSL). In order to use Cg, one has to install additional libraries, while GLSL can be compiled by the driver of the graphics card, if it supports OpenGL 2.0 or later version. That is why GLSL was used in this work. The difference to the traditional way of programming is that the shaders are compiled and linked by the graphics card driver during run time. This makes them more difficult to debug and test.

Perform computation on the GPU

In order to perform computations on the graphics card, a suitable geometry has to be rendered. However, we have to first specify that the built-in rendering pipeline has to be replaced by the shader we wrote. In OpenGL this is done by a single call *glUseProgram(myProgram*) and after the program does its job, *glUseProgram(0*) restores defaults.

Then we specify an orthogonal projection and a viewport.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity(); gluOrtho2D(0.0, texSize, 0.0, texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); glViewport(0, 0, texSize, texSize);
```

and render a filled quad, which ensures that our fragment shader is executed for each data element we stored in the target texture.

```
glPolygonMode(GL_FRONT,GL_FILL);
glBegin(GL_QUADS);
   glTexCoord2f(0.0, 0.0);
   glVertex2f(0.0, 0.0);
   glTexCoord2f(1.0, 0.0);
   glVertex2f(texSize, 0.0);
   glTexCoord2f(1.0, 1.0);
   glVertex2f(texSize, texSize);
   glTexCoord2f(0.0, 1.0);
   glVertex2f(0.0, texSize);
glEnd();
```

DATA AND ALGORITHM ON THE GPU

Data structures

The mass-spring model very naturally maps into several texture2D, as described in the previous section. One very important fact has to be considered, when designing data. Usually only one rendering target is allowed. Although multiple rendering targets are possible in GLSL, this is not very efficient of all graphics cards. In addition, if a texture is set as a rendering target, it is not available for reading. So, in order to compute the new velocities and positions in equation 3, one has to store the old values in another texture, just for reading. After a computational step, the two textures are swapped, and the reading texture becomes a rendering target. This technique is know as "ping-pong" and is often used in GPGPU (Göddeke, 2007). The textures have three components corresponding to (X, Y, Z) values of the three dimensional points and vectors.

So, the textures we define are as follows:

- Two texture2Ds (read/write) for velocities;
- Two texture2Ds (read/write) for positions;
- One texture2D for normal vectors of the cloth surface at each cloth vertex.

We take advantage of the fact that positions are in the graphics memory to compute the normal vectors there, too. That is why another texture2D is defined for the normal vectors.

Now we have to make up a data structure for the springs. The main idea of this work is to store information about the springs connected to each mass point. We should also impose a suitable constraint on the maximum number of other vertices to which a given mass point is connected. For the cloth model this number is 12, while for the deformable solid it is 10. To make things more general we suppose that each point is connected to a maximum of 16 other points. So, if the textures for velocities, positions and normals have a size of (texSize x texSize), we use one more texture, which we call "connectivity texture", which is of a size (4*texSize)x(4*texSize). This connectivity matrix consists of 16 smaller matrices. Each entry in these 16 matrices has 4 channels (RGBA) and keeps the following information of a spring connected to the corresponding vertex: entry.rg – texture coordinates of the other spring end point, entry.b – natural length, entry.a – spring stiffness. If all channels are equal to -1.0, this means that the entry represents no connection.

Algorithm

The idea of the simulation algorithm on the GPU is that there is no loop that goes for each spring. The computation of internal forces is included in the loop for each mass point. In this way the forces due to each spring will be computed twice, but as this is done in parallel this is more efficient than first to go for each spring. One iteration step of the algorithm is shown below.

Algorithm 2. Mass-spring integration on the GPU

```
Rendering 1:
For each mass point
   Compute internal forces
   Add external forces
   Compute velocity
   Do collision detection and response (modify velocity)
End for
Rendering 2:
For each mass point
   Compute new position
End for
```

In fact the "for loops" are not specified in the shader, the operations inside are performed in parallel and are scheduled by GPU control unit, which is not programmable. As shown in the algorithm each iteration step performs two renderings on the graphics card. First the velocity write texture is set as a rendering target. After it is computed the read and write textures are swapped. Then the position write texture is set as the current rendering target, after that the position textures are swapped.

The internal forces are computed with the help of the connectivity texture (texCon) as shown in algorithm 3 below.

Algorithm 3. Internal forces computation

```
ff=vec3(0.0,0.0,0.0);
vec2 curr=get_current_texcoors();
vec3 v = sample_texture2D(texVel, curr).rgb;
vec3 p = sample_texture2D(texPos, curr).rgb;
curr*=0.25;
for (x=0.0;x<1.0;x+=0.25)
for (y=0.0;y<1.0;y+=0.25){
  vec4 con=sample_texture2D(texCon, curr);
```

```
  if (con.r<-0.5) break;
  ff+=compute1Force(p,con.rg,con.b,con.a);
}
```

The operation get_current_texcoors() is built-in and retrieves the current texture coordinates to be processed, scheduled by the GPU control unit. The operation sample_texture2D has the same functionality as array indexing. It retrieves the 4-coordinate value stored in the specified texture at coordinates (curr.x, curr.y).

RESULTS

The algorithm was implemented and tested on two machines:

1) Toshiba Satellite Pro laptop, 1.86 GHz Pentium M70 processor, 1 GB RAM and an ATI Mobility Radeon X700 graphics card.

2) Desktop with 2.8 GHz AMD Athlon 64 Dual processor, 3.5 GB RAM and NVIDIA GeForce GTX 260 graphics card.

The simulation of a square tablecloth on a round table, as shown in Fig. 2, was implemented on both CPU and GPU under Windows XP using Microsoft Visual C++, OpenGL library and GLSL for programming the GPU. The fact that the sizes of the textures have to be a power of two does not limit the generality of the approach and arbitrary resolutions of tablecloth vertices are possible. For example, if we set the resolution to 48x48 vertices, the selected textures are 64x64, but the vertices are places in the upper left corner of the texture. When computing the results, only that part of the texture is rendered, which corresponds to the active pixels. The collision detection in this particular case is trivial and is done in the object space, but in more complex scenes it can be performed in the image space, taking advantage of the GPU as described by Vassilev *et al.*[7].

The performance of the two techniques was compared for different resolutions of the mass points grid on the tablecloth. The results for the two machines are shown in Table 1. As expected the advantage of the GPU is much more evident for higher grid (texture) resolutions, because of the SIMD parallel character of the GPU.

Fig. 2. Simulation of a tablecloth on a round table

CONCLUSIONS AND FUTURE WORK

The current paper presented an algorithm and data structures for implementing a general mass-spring system on the modern GPU. A cloth model, based on a mass-spring system with improved elasticity properties achieved by modification of velocities of cloth vertices [7], was implemented on the graphics processor. The rectangular piece of cloth is discretised in a two dimensional array of mass points which naturally maps to textures of the GPU. The results show that significant increase of the performance is achieved.
In the future the approach will be also implemented and tested for a more complex simulation of garments on virtual characters.

Table 1. Comparison of the CPU and GPU performance

Grid Resolution	CPU				GPU				Advantage of GPU, %	
	Time for 2000 iterations, s		Iterations per second		Time for 2000 iterations, s		Iterations per second			
	1	2	1	2	1	2	1	2	1	2
48 × 48	4.39	3.05	455	656	1.68	0.67	1190	2985	261	455
64 × 64	7.70	5.53	260	362	2.20	0.67	909	2985	350	825
96 × 96	18.65	12.30	107	163	4.25	0.67	470	2985	438	1835
128 × 128	32.86	21.87	61	91	6.90	0.67	290	2985	476	3264

REFERENCES

[1] Baraff, D, A. Witkin. Large steps in cloth simulation. Computer Graphics, Proceedings of SIGRAPH'98, Annual Conference Series, 1998, pp. 43–54.

[2] Desbrun, M., P. Schroeder, A. Barr. Interactive animation of structured deformable objects. Proceedings of Graphics Interface (1999), pp. 1–8. Canadian Computer-Human Communications Society.

[3] Göddeke, D. GPGPU::Basic Math Tutorial, 2007 http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html

[4] Houston, M. General-Purpose Computation on Graphics Hardware. SIGGRAPH 2007 GPGPU Course, http://www.gpgpu.org/s2007/

[5] Provot, X. Deformation constraints in a mass-spring model to describe rigid cloth behaviour. Proceedings of Graphics Interface'95, 1995, pp. 141-155.

[6] Vassilev, T. I., Spanlang, B. A Mass-Spring Model For Real Time Deformable Solids, Proceedings of East-West Vision 2002, pp. 149-154, Graz, Austria, September 12-13, 2002

[7] Vassilev, T., Spanlang, B., Chrysanthou Y. Fast Cloth Animation on Walking Avatars, Computer Graphics Forum, 2001, 3 (20), 260-267.

[8] Volino, P., M. Courchesne, N. Magnenat-Thalmann. Versatile and efficient techniques for simulating cloth and other deformable objects. Proceedings of SIGGRAPH'95, 1995, pp. 137–144.

[9] Volino, P., N. Magnenat-Thalmann. Comparing efficiency of integration methods for cloth simulation. Computer Graphics International 2001, pp. 265–272.

ABOUT THE AUTHORS

Assoc. Prof. Tzvetomir I Vassilev, PhD, Department of Informatics and Information Technologies, University of Rousse, Bulgaria, Phone: +359 82 888475, E-mail: TVassilev@ru.acad.bg.

Mr. Roumen I Rousev, MSc, Department of Informatics and Information Technologies, University of Rousse, Bulgaria, Phone: +359 82 888326, E-mail: rir@ami.ru.acad.bg.

Докладът е рецензиран.