Fast GPU Garment Simulation and Collision Detection

Tzvetomir I. Vassilev
Dept. of IIT, University of Ruse
8 Studentska St
Bulgaria 7017, Ruse
tvassilev@uni-ruse.bg

Bernhard Spanlang
EventLab, Universitat de Barcelona
Campus de Mundet - Edifici Teatre
Passeig de la Vall d'Hebron 171,
Spain 08035, Barcelona
bspanlang@ub.edu

ABSTRACT

This paper describes a technique for garment simulation and collision detection implemented on modern Graphics Processors (GPU). It exploits a mass-spring cloth model with velocity modification approach to overcome the super-elasticity. Our novel algorithms for cloth-body and cloth-cloth collision detection and response are based on image-space interference tests. For collision detection a 3D texture is generated, in which each slice represents depth and normal information for collision detection and response. These algorithms were implemented to build a fast web-based virtual try-on system. Our simulation starts from flat garment pattern meshes and performs the entire seaming and cloth draping simulation on the GPU. By mapping cloth properties of real fabric measurements we are able to approximate the real drape behaviour of different types of fabric, taking into account different warp and weft fabric drape properties. As the results section shows the average time of dressing a virtual body with a garment on state of the art graphics hardware is 0.2 seconds.

Keywords

Cloth Simulation, GPU programming, Collision detection.

1. INTRODUCTION

Physical simulation and elastic deformable objects have been widely used by researchers in computer graphics. The main applications of garment simulation are in the entertainment industries, in the fashion design industry and in electronic commerce when customers shop for garments on the web and try them on in a virtual booth.

The graphics processing unit (GPU) on today's commodity video cards has evolved into an extremely powerful and flexible processor [LHG*06]. The latest graphics architectures provide huge memory bandwidth and computational power, with fully programmable vertex and pixel processing units that support vector operations up to full IEEE floating point precision. Architecturally, GPUs are highly parallel streaming processors optimized for vector operations, with single instruction on multiple data (SIMD) pipelines. Not surprisingly, these processors are capable of general-purpose computation beyond the graphics applications for which they were designed and many researchers have utilized them in cloth modelling [Zel05], [GW05].

2. BACKGROUND

Previous work in cloth simulation

Physically based cloth modelling has been a problem of interest to computer graphics researchers for more than two decades. First steps, initiated by Terzopoulos et al. [TPBF87], characterised cloth simulation as a problem of deformable surfaces and used the finite element method and energy minimisation techniques borrowed from mechanical engineering. Since then other groups have been formed [BHW94], [EWS96], [CYTT92] challenging the cloth simulation using energy or particle based methods.

Provot [Pro95] used a mass-spring model to describe rigid cloth behaviour, which proved to be faster than the techniques described above. Its major drawback is the super-elasticity. In order to overcome this problem he applied a position modification algorithm to the ends of the over-elongated springs. However, if this operation modifies the positions of many vertices, it may elongate other springs. Vassilev et al. [VSC01] used a velocity modification approach to solve the super-elasticity problem.

The nature of the mass-spring system is suitable for implementation on the GPU. NVIDIA [Zel05] have provided a free sample demo of a mass-spring cloth simulation on their graphics processors. Their cloth model is quite simple and does not simulate resistance to bending. Rodriguez-Navarro et al. have published two implementations of cloth model on the GPU. The first is based on a mass-spring system [RNSS05] and the second on the finite element method [RNS06]. Georgii and Westermann [GW05] compared two possible implementations of mass-spring systems on the GPU and tested them with cloth simulation. A GPU accelerated mass-spring system has been used in other fields like surgical simulation [MHS05]. The advantage of the mass-spring system, described in this paper, is that it implements on the GPU a velocity modification approach for overcoming the super-elasticity, which results in a faster and more realistic simulation. Unlike other GPU implementations we perform also the garment seaming process on the GPU and we use fabric property measurements in order to approximate the behaviour of real fabric to a good degree.

Mass-spring model of cloth

The method, described in this work is based on the cloth model described in [VSC01]. The elastic model of cloth is a mesh of $l \times n$ mass points, each of them linked to its neighbours by massless springs of natural length greater than zero. There are three different types of springs: structural, shear, and flexion, which implement resistance to stretching, shearing and bending, correspondingly.

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

$$\mathbf{f}_{ij} = m\,\mathbf{a}_{ij}, \qquad (1)$$

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

The **internal forces** are due to the tensions of the springs. The overall internal force applied at the point $\mathbf{p}_{ij}$ is a result of the stiffness of all springs linking this point to its neighbours:

$$f_{int}(p_{ij}) = -\sum_{k,l} k_{ijkl}\left(\overrightarrow{p_{kl}p_{ij}} - l^0_{ijkl}\frac{\overrightarrow{p_{kl}p_{ij}}}{\left\|\overrightarrow{p_{kl}p_{ij}}\right\|}\right), \qquad (2)$$

where k_{ijkl} is the stiffness of the spring linking $\mathbf{p}_{ij}$ and $\mathbf{p}_{kl}$ and $l$is the natural length of the same spring.

The **external forces** can differ in nature depending on what type of simulation we wish to model. The most frequent ones are gravity and viscous damping.

All the above formulations make it possible to compute the force $\mathbf{f}_{ij}(t)$ applied on cloth vertex $\mathbf{p}_{ij}$ at any time t. The fundamental equations of Newtonian dynamics can be integrated over time by a simple Euler, Verlet or Runge-Kutta method [PTVF92].

Collision detection

Collision detection (CD) and response prove to be the bottleneck of dynamic simulation algorithms that use highly discretised surfaces. Most CD algorithms between cloth and other objects in the scene are based on geometrical object-space (OS) interference tests. Some apply a prohibitive energy field around the colliding objects [TPBF87], but most of them use geometric calculations to detect penetration between a cloth particle and a triangle of the object together with techniques that reduce the number of tests.

Most common approaches are voxel or octree subdivision [Gla98]. Another solution is to use a bounding box (BB) hierarchy [BW98], [Pro97]. Objects are grouped hierarchically according to proximity rules and a BB is pre-computed for each object. The collision detection is then performed by analysing BB intersections in the hierarchy. Other techniques exploit proximity tracking [VM95] or curvature computation [Pro97] to reduce the large number of collision checks, excluding objects or parts which are impossible to collide.

Another approach to CD is based on image-space (IS) tests [SF91], [MOK95], [BWS99]. These algorithms use the graphics hardware to render the scene and then perform checks for interference between objects based on the depth information of the rendered image. In this way the 3D problem is reduced to 2.5D. Vassilev et al. [VSC01] applied this technique for detecting collisions between cloth and body when dressing virtual characters. They created depth, normal and velocity maps using two orthogonal cameras that were placed at the centre of the front and the back face of the body's BB. The depth map was used for detecting collisions, while the normal and velocity maps were used for collision response. Since they perform cloth simulation on the CPU they have to read back the frame buffers from the GPU which is time consuming.

Heidelberger at al [HTG04] extended the image space based approach to also deal with self-collisions by creating separate layered depth images (LDIs) on the GPU for front and back facing polygons. Their approach only works with water tight volumes and also requires the reading back of the frame buffer to the CPU for analysis.

Allard et al [AFC*10] recently built on the LDI approach but moved the whole simulation of deformable volumes to the GPU, avoiding the bottleneck of framebuffer readback. However, their approach also relies on watertight volume geometry.

In Govindaraju et al. [GKLM07] collision detection is regarded as a visibility problem and they use occlusion queries on the graphics processor to detect collisions at fast rates. Their performance tests show collision detection at rates over 100ms though.

Another approach to perform collision detection on the GPU was introduced by Sud et al [SGG*06]. They create a discrete Voronoi diagram (DVD) of the scene on the GPU and can therefore access proximity and collision information. This is useful if the topology of the geometry can change, as for example in fractures, etc. However, they report rates of several hundred milliseconds just for creating the discrete Voronoi diagrams.

The method described in this paper exploits the idea of the image-space approach of Vassilev et al. [VSC01] but we implement it entirely on the GPU including the cloth simulation and therefore we eliminate the bottleneck of frame buffer readback and the analysis of the framebuffer for collision detection and response on the CPU. Moreover, we extend the algorithm to test not only for cloth-body collisions, but also to cloth-cloth collisions. Unlike [HTG04] and [AFC*10] our self-collision tests are not based on LDIs but our method exploits the information we have on the GPU about the separate layers of cloth. Our method therefore does not rely on watertight volume geometry and we are able to perform collision detection and cloth-cloth collision detection at rates much higher than previously reported.

3. MASS-SPRING CLOTH ON THE GPU

Algorithm

Implementation of the mass-spring cloth model [VSC01] on the CPU requires an algorithm similar to the following pseudo code:

```
For each spring
  Compute internal forces
  Add forces to 2 end mass points
Endfor
For each mass point
  Add external forces
  Compute velocity
  Do collision detection and response
  Correct velocities for over-elongated
     springs
  Compute new positions
Endfor
```

Two implementations of the mass-spring system are possible on the GPU, which were compared by Georgii and Westermann [GW05]. The first one is to directly follow the CPU implementation, which they call edge-centred approach (ECA). The main difficulty here is to distribute the spring forces to the correct mass points with the correct sign. To solve this they use vertex shaders, but two additional render passes are required. The advantage of the ECA is that spring forces are computed only once, but it has several drawbacks:

- it requires more graphics memory for spring and force textures;
- it requires at least four rendering passes;
- additive blending in the render target is used to accumulate the force contributions, which has precision problems on some cards.

The second implementation uses only one for loop (for each mass point) and the computation of the spring forces is the first step inside. As a result it can be implemented in only one or two render passes, requires less graphics memory and is more straightforward to implement. Its only disadvantage is that each spring force is computed twice, but considering the parallel nature and tremendous power of recent GPUs this is negligible. Therefore the work in this paper is based on the second method. The texture units needed for our algorithm are described in the next sections.

GPU data structures for the cloth model

On the GPU the mass-spring model naturally maps into several texture2Ds. Several important facts have to be considered, when organising the data. 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 of cloth vertices, the old values have to be stored in another texture, just for reading. We use the so called "ping-pong" technique [Göd07]: after a computational step, the two textures are swapped, and the texture from which was read before becomes the new rendering target.

Therefore, we need two textures (read/write) for velocities, two textures (read/write) for positions and one texture for normal vectors of the cloth surface at each cloth vertex, which are also computed on the GPU.

The main idea of this work is to store information about the springs connected to each mass point in an additional texture, which we call "spring connectivity texture". A suitable constraint on the maximum number of other vertices connected to a given mass point has to be imposed. For our cloth model this number is 12, owing to GPU architecture it is simpler to reserve 16 values, therefore we have 4 values for future extensions. If the textures for velocities, positions and

normals have size (texSize × texSize), then the spring connectivity texture is of size (4*texSize)×(4*texSize). This spring 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: texture coordinates of the other spring end point, natural length and spring stiffness. If all channels are equal to -1.0, this means that the entry represents no connection.

Seaming of garment pieces

Our Virtual Try-On system reads a body file and a garment file and dresses the body. The garment file holds information about the geometry of the cutting patterns, physics parameters of the cloth and seaming information. The patterns are automatically positioned around the body and external forces are applied along the seaming lines. The seaming lines are discretized into groups of two or three cloth vertices to be sewn together.

The connectivity of cloth vertices into seams is stored in a similar texture as for the springs, which we call "seam connectivity texture". Each entry of the texture keeps information about the other mass points to which the current cloth vertex has to be sewn. During the simulation forces are applied which pull together the corresponding vertices. When the vertices are closer than a certain threshold, they are marked as sewn and are attached to each other. The simulation ends when all seams are done, or after a pre-defined number of iterations which means the garment is too small for the body.

Occlusion queries for counting sewn vertices

In order to identify when the garment is sewn, the number of vertices still to be sewn have to be counted. Counting on the GPU can be performed using a reduction approach similar to the max reduction, described by Harris [Har05]. However, it cannot be applied directly. First, a 2D buffer has to be built, which contains ones for the not sewn vertices and zeros for the sewn ones. Then a sum reduction should be applied to the buffer, which will perform the required count.

In our system we utilize GPU occlusion queries for counting. Occlusion queries are implemented in hardware on most of the recent graphics cards and they allow the programmer to render a 3D scene and to obtain the number of fragments that passed the depth test. There is an internal counter, which is initially set to zero, and during the rendering it is increased by one for each fragment that passes the depth test.

In order to use occlusion query in our case, the following steps have to be carried out:

- Allocate a depth texture;
- Set this texture as a depth buffer for rendering;
- Render a suitable geometry and perform an occlusion query to retrieve the number of fragments that pass the depth test.

After the new positions were computed by the mass spring simulation we call a shader which holds the sewn vertices together and also builds a seam depth texture with the following values: 0 if the vertex is not involved in a seam, 1 if the vertex is part of a seam but is not sewn yet and 0.5 if the vertex is part of a seam, which is sewn. Then we set this texture as the default depth buffer. The z-buffer is turned to read only, otherwise the depth values will be replaced with the ones of the incoming vertices. Next we render a quad with a depth value of 0.8 that covers the whole draw buffer. In this way the occlusion query counts all fragments with a depth value greater than 0.8, that is the number of unsewn vertices. In fact we can render a quad with any depth value in the interval (0.5, 1).

The function for counting unsewn vertices does not need to be called after every integration step. To speed the simulation up it could be called after every 10 or 15 iterations.

Cloth Modelling Shaders

The following shaders are used in the system:

Velocity shader. This is the main cloth simulation shader. It computes the forces applied to each cloth vertex due to springs' tension, gravity, damping and seaming, then integrates over time to compute velocity and writes the result in the velocity texture. It also checks for collisions, as described below and if there is a collision it applies a force and also modifies the velocity to resolve the collision.

Position shader. It reads from the velocity texture and computes the new cloth vertex positions.

Seam shader. It checks if the cloth vertices that participate in a seaming line are close enough to be considered sewn, holds the sewn vertices together and builds the seam depth texture, as described in the above section.

4. COLLISION DETECTION BODY-CLOTH AND CLOTH-CLOTH

Cloth-body collision detection

As explained in Section 2 this paper exploits the idea of Vassilev et al. [VSC01] for collision detection. However, we do not build velocity maps, because the current system does not animate the virtual body representation. The garment is dressed and simulated on a static body. To build the normal maps more effi-

ciently we use a simple vertex shader, which replaces the vertex colour RGB values with the XYZ coordinates of the normal. In addition, to reduce the number of texture units, the front and back maps are placed in a single texture, as shown in Figure 1.

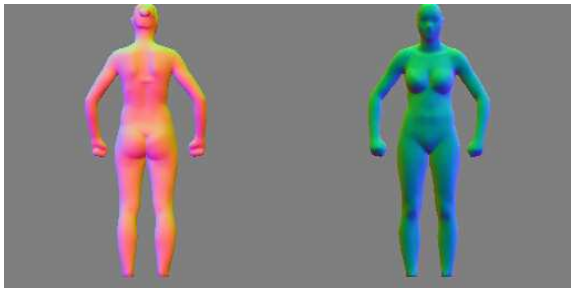

Figure 1: *Front and back normal maps in one texture*

And finally the normal and depth maps are placed in the same texture unit; RGB representing the normal coordinates and the alpha channel contains the depth. This speeds up the simulation, because when testing for collisions the velocity shader samples the collision texture only once to get depth and normal values of the front and the back of the body.

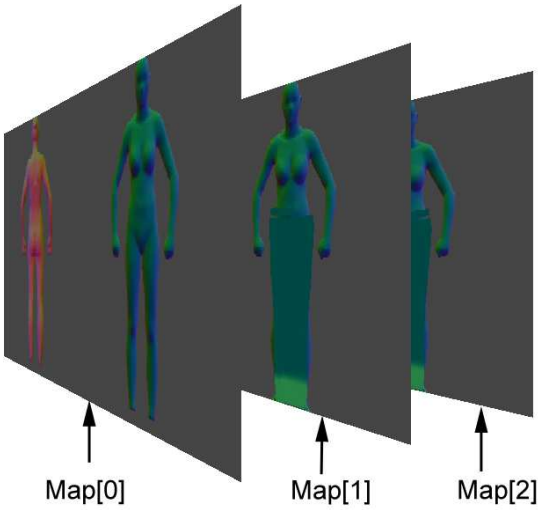

Figure 2: *Normal maps generated for collision detection*

Cloth-cloth collision detection

Our system does not aim at detecting all cloth-cloth collisions. It does not test for collisions in one piece of cloth as such are less likely to happen in tight garments on static body we simulate. When constructing garments some pieces of cloth have to be placed on top of others, for example pockets, belt loops, etc. When a person tries on two garments one is always on top of the other for example a T-shirt and a pair of jeans. The cloth pieces are grouped into layers, to know which layer is on top of other layers and we can assign a layer number to each cloth piece, starting from zero. Our system only tests for collisions between these different cloth layers.

The idea of this work is to use the same image-space approach for detecting collisions between layers of cloth. For this purpose several maps are generated (Fig. 2):

- Map[0]: body depth and normals
- Map[1]: body and cloth-layer 0 depth and normals
- …
- Map[*n*]: body and cloth-layers 0 to *n*-1 depth and normals

The number of maps, *n*, depends on the number of cloth layers we wish to simulate. Map[0] is generated only once at the beginning of the simulation, because the body does not move in our case. All other maps have to be generated at each iteration step. If we want to simulate garments on a body in motion, map[0] has to be generated at each iteration as well.

Collision checking

The computation of internal, external forces and velocities, as well as collision detection and response is performed in the velocity shader, as described above. This shader is called only once per integration step. When testing for collisions of a particular cloth vertex, which belongs to cloth layer *i*, we have to check if it collides with the body and all layers beneath it, which means that we have to use map[*i*] for CD and response. All maps are stored in a 3D texture in which each slice corresponds to a depth/normal map, as described above. The properties of each cloth vertex, such as mass, elasticity, cloth layer number, etc., are stored in another texture. The velocity shader uses the layer number to sample the appropriate slice of the 3D texture, for example cloth layer with number *i* samples slice *i* and uses it for collision detection and response as described in [VSC01]. The depth value of the current mass point is compared to the depth value read from the depth map. If a collision is detected a repulsive force is computed and applied to the cloth vertex using the normal vector, retrieved from the normal map. The velocity of the cloth vertex is also modified. The collision response enables us to simulate friction between layers, too.

Applying this approach allows us to simulate one-way interactions only. The lower layers push the upper layers back to prohibit penetration, but the upper layers do not affect the layers below. So, if a pair of jeans is dressed on top of a loose shirt, the jeans will not push the shirt towards the body.

In order to model interaction in both directions we do the following. The faces on the cloth surface are numbered from one to the number of faces and each number is encoded as vertex colour. When generating

the depth and normal maps the cloth surface is rendered using flat shading, so that these colours are not interpolated and the face colour encodes the face number. The fragment shader, used for the generation of the maps, stores the XYZ values of the normal vector in the RGB values of the map and the alpha value is computed as follows:

$$map\_alpha = depth + face\_number, \quad (3)$$

where face_number is decoded from the colour values. As the depth value is from 0 to 1 and the face number is greater than or equal to one, the two values can be easily separated. If a collision is detected the velocity shader, in addition to applying repulsive forces and modifying velocities, also writes the following alpha in the velocity texture:

$$velo\_alpha = face\_number. \quad (4)$$

If there is no collision the alpha is set to zero.

Another pair of vertex/fragment shaders is used to apply forces to lower layers of cloth. If a lower layer cloth face has collided with an upper layer vertex, we apply forces to each of the three vertices of this face, which are opposite to the face normal. These forces have to be summed up for each vertex, as a vertex can be part of several adjacent faces that have collided. In fact we integrate the forces over time and add them to the velocities. One of the velocity textures is set as a rendering target and the velocities are rendered three times as points with additive blending, once for each vertex of a triangular face. A uniform variable is set to 0, 1 or 2 before the rendering to define which face vertex is targeted. The vertex shader checks the forth coordinate of the velocity. If it is greater than zero, then this is a face which has collided to an upper layer cloth vertex. The indices of the cloth vertices of that face are read from an index texture. Knowing the size of the velocity texture and the index, the output position of the vertex shader is computed so that it projects to the targeted cloth vertex (number 0, 1 or 2 of the face) in the rendering target. As a result the fragment shader is executed for this cloth vertex and it applies a constant force opposite to the cloth normal, multiplied by the time step, and in this way pushes the cloth back. The magnitude of the force is determined experimentally. If there was no collision, the new position is computed so that it is outside the viewing volume and the fragment shader is not executed for this vertex.

Maps generation

As mentioned above map[0] is generated only once at the beginning. After each integration step we have to render maps from 1 to *n* in each slice of the 3D texture. In order to speed the simulation up, when generating map[i], we first copy map[i-1] to map[i], set it as the default colour and depth buffers and render cloth layer number (i-1). In this way the body and all cloth layers from 0 to (i-2) are already present in the frame buffer and do not have to be rendered again. So we have to render only the pieces with a layer number (i-1).

5. RESULTS

The system was implemented in OpenGL and GLSL. Figure 3 shows an example of the simulation of a pair of jeans and gives a closer look of pockets and belt loops.

In order to check the system performance, it was tested on several configurations. Two implementations of the Virtual-Try-On system were compared: 1) programs running on the CPU and 2) GPU-based as described in this paper. For the second one the textures used to store the cloth vertices positions and velocities were of a size 64 × 64. The influence of the upper to lower layers was not simulated, as for a single garment it is not significant. The performance results of 3 GPUs and the fastest CPU are given in Table 1 and Figure 4. They show that the virtual try-on runs very well on a modern laptop GPU, which is about 20 times faster than the fastest tested CPU. One iteration includes integration, collision detection and response.

Figure 3: Layers of cloth: pockets and belt loops

The average time of putting a garment on a virtual body using the NVIDIA GTX 460 GPU is about 0.2 seconds depending on the garment complexity and size.

Figure 6 shows results of the simulation of two garments, jeans dressed on top of a shirt. The jeans were discretised with 3600 mass points and 2100 vertexes were used for the shirt. Two modifications of the cloth-cloth collision detection algorithm are depicted:

left – no impact of upper to lower layers. The simulation is faster (0.20 sec for dressing the jeans), but not satisfactory.

right – with impact of upper to lower layers. The simulation is slower (0.34 sec for dressing the jeans), but of much better quality. The time spent only on collision detection and response is 0.21 seconds for the whole cycle of dressing a pair of jeans, which requires 675 iterations. This means that for each iteration 0.31 ms is spent on cloth-cloth collision detection and response.

	Time for 1000 iterations, s	Iterations per second
Intel core i7, 2.2 GHz	8.65	116
ATI Radeon HD4850	0.57	1769
NVIDIA GeForce GTX 560M laptop	0.40	2491
NVIDIA GeForce GTX 460	0.30	3300

Table 1: **Performance of the system, measured on 3 GPUs and a CPU**

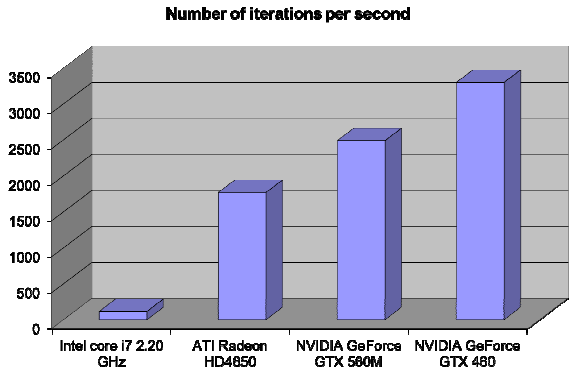

Figure 4: Performance of CPU and 3 GPUs

The algorithms for cloth simulation and collision detection and response were also implemented using NVidia CUDA and OpenCL. The comparison with the GLSL implementation [Vas10] showed that GLSL outperforms CUDA (OpenCL). One of the main reasons is that CUDA (OpenCL) and OpenGL have to share buffers for the rendering and this buffers should be mapped and later unmapped when used in CUDA (OpenCL), which slows down the simulation.

Figure 6: **Jeans dressed on a shirt; left: no impact of upper to lower layers; right: with impact of upper to lower layers**

6. CONCLUSIONS

An efficient technique for dynamic garment simulation entirely on the GPU has been presented. It implements a mass-spring system with velocity modification to overcome super elasticity and exploits an image-space approach for collision detection and response. The following more important conclusions can be drawn:

- A general mass-spring system can be implemented on the GPU using several textures for storing data and several connectivity textures for keeping spring and seaming information.
- Hardware assisted occlusion queries can be utilised for counting unsewn cloth vertices, which speeds simulation up.
- The same image-space based approach can be applied for detecting collisions cloth-body and cloth-cloth when layers of cloth are simulated. Multiple collision maps can be stored in a 3D texture.

The system can simulate approximately 20 garments per second on a PC with 2 dual GPU NVidia GeForce graphics cards. This is currently sufficient for our web based Virtual Try On service.

The system can be extended to simulating garments on animated virtual characters. For this purpose velocity maps will also have to be generated and stored in another 3D texture.

7. ACKNOWLEDGEMENTS

Tzvetomir Vassilev's work is partly supported by a National Research Fund project at the University of

Ruse, Bulgaria. Bernhard Spanlang's work is partially supported by the ERC project TRAVERSE.

8. REFERENCES

[AFC*10] Allard J., Faure F., Courtecuisse H., Falipou F., Duriez C., Kry P. G.: Volume contact constraints at arbitrary resolution. ACM Trans. Graph. 29, 4 (2010), 1–10.

[BHW94] Breen D., House D., Wozny M.: Predicting the drape of woven cloth using interacting particles. In Computer Graphics Proceedings, Annual Conference Series (1994), vol. 94, pp. 365–372.

[BW98] Baraff D., Witkin A.: Large steps in cloth simulation. In Computer Graphics Proceedings, Annual Conference Series (1998), SIGGRAPH, pp. 43–54.

[BWS99] Baciu G., Wong W. S., Sun H.: Recode: an image-based collision detection algorithm. The Journal of Visualization and Computer Animation 10, 4 (1999), 181–192.

[CYTT92] Carignan M., Yang Y., Thalmann N. M., Thalmann D.: Dressing animated synthetic actors with complex deformable clothes. In Computer Graphics Proceedings, Annual Conference Series (1992), vol. 92, pp. 99–104.

[EWS96] Eberhardt B., Weber A., Strasser W.: A fast, flexible, particle-system model for cloth draping. j-IEEE-CGA 16, 5 (Sept. 1996), 52–59.

[GKLM07] Govindaraju N. K., Kabul I., Lin M. C., Manocha D.: Fast continuous collision detection among deformable models using graphics processors. Comput. Graph. 31, 1 (2007), 5–14.

[Gla98] Glassner N. I. B. A. S.: 3d object modelling. SIGGRAPH 12, 4 (1998), 1–14.

[Göd07] Göddeke D.: Gpgpu::basic math tutorial, 2007.

[GW05] Georgii J., Westermann R.: Mass-spring systems on the gpu. Simulation Modelling Practice and Theory 13 (2005), 693–702.

[Har05] Harris M.: Mapping computational concepts to gpus. In SIGGRAPH '05: ACM SIGGRAPH 2005 Courses (New York, NY, USA, 2005), ACM, p. 50.

[HTG04] Heidelberger B., Teschner M., Gross M.: Detection of collisions and self-collisions using image-space techniques. In Journal of WSCG (2004), pp. 145–152.

[LHG*06] Luebke D. P., Harris M., Govindaraju N. K., Lefohn A. E., Houston M., Owens J. D., Segal M., Papakipos M., Buck I.: S07 - gpgpu: general-purpose computation on graphics hardware. In SC (2006), ACM Press, p. 208.

[MHS05] Mosegaard J., Herborg P., Sørensen T. S.: A gpu accelerated spring mass system for surgical simulation. Studies in health technology and informatics 111 (2005), 342–348.

[MOK95] Myszkowski K., Okunev O. G., Kunii T. L.: Fast collision detection between complex solids using rasterizing graphics hardware. The Visual Computer 11, 9 (1995), 497–512.

[Pro95] Provot X.: Deformation constraints in a mass-spring model to describe rigid cloth behaviour. In Proceedings of Graphics Interface (1995), pp. 141–155.

[Pro97] Provot X.: Collision and self-collision detection handling in cloth model dedicated to design garments. In Proceedings of Graphics Interface (1997), pp. 177–189.

[PTVF92] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.: Numerical Recipes in C, 2nd. edition. Cambridge University Press, 1992.

[RNS06] Rodriguez-Navarro X., Susín A.: Non structured meshes for cloth gpu simulation using fem. In 3rd. Workshop in Virtual Reality, Interactions, and Physical Simulations (VRIPHYS) (2006), pp. 1–7.

[RNSS05] Rodriguez-Navarro X., Sainz M., Susin A.: Gpu based cloth simulation with moving humanoids. In Actas XV Congreso Español de Informática Gráfica (CEIG'2005) (2005), J. Regincós D. M., Thomson-Paraninfo E., (Eds.), pp. 147–155.

[SF91] Shinya M., Forgue M. C.: Interference detection through rasterization. j-J-VIS-COMP-ANIMATION 2, 4 (Oct.–Dec. 1991), 132–134.

[SGG*06] Sud A., Govindaraju N., Gayle R., Kabul I., Manocha D.: Fast proximity computation among deformable models using discrete voronoi diagrams. ACM Trans. Graph. 25, 3 (2006), 1144–1153.

[TPBF87] Terzopoulos D., Platt J., Barr A., Fleischer K.: Elastically deformable models. Computer Graphics (Proc. SIGGRAPH'87) 21, 4 (1987), 205–214.

[Vas10] Vassilev T.I.: Comparison of several parallel API for cloth modelling on modern GPUs. In Proceedings of CompSysTech (2010).

[VM95] Volino P., Magnenat Thalmann N.: Collision and self-collision detection: Efficient and robust solutions for highly deformable surfaces. In Computer Animation and Simulation '95 (1995), Terzopoulos D., Thalmann D., (Eds.), Springer-Verlag, pp. 55–65.

[VSC01] Vassilev T., Spanlang B., Chrysanthou Y.: Fast cloth animation on walking avatars. Computer Graphics Forum 20, 3 (2001), 260–267. ISSN 1067-7055.

[Zel05] Zeller C.: Cloth simulation on the gpu. In SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches (New York, NY, USA, 2005), ACM, p. 39