JOURNAL OF APPLIED
COMPUTER SCIENCE
Vol. 24 No. 2 (2016), pp. 39-54

GPU Assisted Self-Collisions of Cloths

Adam Wojciechowski[1], Tomasz Gałaj[2]

[1]*Lodz University of Technology*
Faculty of Technical Physics, Computer Science and Applied Mathematics
Wolczanska 215, 90-924 Lodz, Poland
adam.wojciechowski@p.lodz.pl

[2]*Lodz University of Technology*
Faculty of Technical Physics, Computer Science and Applied Mathematics
Wolczanska 215, 90-924 Lodz, Poland
tomasz.galaj@gmail.com

Abstract. *Nowadays, people expectations about high realism in games are very high and computers have to make a huge effort to compute every simple detail that occurs in a virtual 3D scene. Fortunately, we can use power of Graphics Processing Units (GPU) to compute some part of the most computationally heavy algorithms. In this paper, we present method to accelerate computations on GPU using Compute Shaders based on cloth simulation with self-collisions for big number of cloth's model vertices (more than 2000).*

Keywords: *computations acceleration, accuracy of collisions resolving, compute shaders, gpu, gpgpu, cloth simulation.*

1. Introduction

Very common, complex and computationally heavy physics task is cloth simulation with self-collisions. Today's games try to simulate it in a way that is very similar to real world cloth behaviour - characters in games no longer use scripted cloth animations that are played over and over, but they are simulated. What is more, these simulations need to take into account collisions with other objects so that cloth can not pass through player's avatar or a wall and with itself to prevent self-penetration. For an ordinary CPU this task is impossible to be performed in real time for a huge amount of particles [1, 2, 3, 4, 5, 6]. However, with the proper use of OpenGL, in-depth knowledge how does GPU work and with GPGPU technologies this task is by all means achievable in a real time system like a 3D game.

Most of the people focus on using OpenCL or CUDA as a mainstream GPGPU technology for fast physics calculations or they use out-of-date programmable pipeline using only Vertex and Fragment Shaders [7, 8, 9]. Though, in this article we will use OpenGL's Compute Shader which is a separate stage in a rendering pipeline and does not need any interoperation and synchronization between two APIs (like OpenGL and OpenCL would need), which cost is very high.

What is more, presented approach also shows that even naive self-collisions algorithm can be done in real time for complex and high density cloths without fancy algorithm optimizations.

2. Related work

The research in the field of cloth simulation extends back to 1980's with several authors focusing on this particular subject since then. However, many authors have contributed to this field, not many of them used power of multi-threading on CPUs or the immense power of GPUs.

In this section, we will present previous methods of simulating cloth mostly with self-collisions. Most of them uses CPU to perform calculations. The crucial aspect are self-collisions which can tremendously enhance the visual aspect of cloth simulation. Unfortunately, these calculations are very time consuming and needs a lot of computational power. In the next sections we will show our method that harnesses the power of GPU to perform cloth simulation with self-collisions for big number of vertices in real-time (16ms - it is the approximation of 60 Frames Per Second which is the frequency that guarantees smoothness of the simulation) which below methods do not guarantee.

2.1. Cloth simulation challenges

What is a bottle-neck in most cloth simulations is that time step between frame n and frame *n+1* must be very small to avoid numerical instability. Therefore, simulation have to be performed several times before rendering to be able to see effects in real time on the screen. Baraff and Witkin [5] proposed a technique that is able to handle large time steps (Figure 1). They successfully proved that using this technique - a combination of implicit integration methods with special enforcement to cloth particle was stable even though the system underwent large time steps. Unfortunately, their solution was far from being able to perform in real-time for big number of vertices. Others [10] also tackled this problem, but they did not applied their theory to cloth simulations.

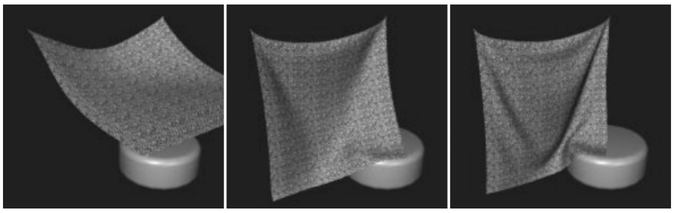


Figure 1: Cloth draping on a cylinder [5].

Another, more common bottle-neck is cloth's self-collision. It is important for cloth simulation to avoid penetration between two parts of the same cloth model, otherwise the simulation will not look realistically. The naive approach of checking these collisions is to perform polygon intersection tests for every section of the cloth which has at least $O(n^2)$ complexity and can significantly reduce the simulation's performance. To avoid the penetration, Lv et al. [6] implemented pruning method based on the strict constraints to prune most of the unwanted particles in self-collision detection. They also have applied constraints to enhance visual quality of the cloth in mass-spring model where edges of the cloth are undesirably stretched that cloth starts to resemble a rubbery material than a cloth one (Figure 2). Despite the fact that this method is very efficient, it only works on a CPU and therefore game programmers are limited to perform any other complex operations on the CPU. The limitation of this method was that it could not achieve real-time performance for more than 2000 cloth's particles.

Volino and Thalmann [4] proposed another efficient method for detecting self-collisions. They extended their former algorithm [11] in a way that they consider that self-collisions may occur within a surface region only if that region is curved enough (Figure 3). Therefore, self-collisions should not be checked for within flat surfaces. Again, this method runs only on CPU and could not perform cloth simulation in 16 ms for large number of cloth's particles.

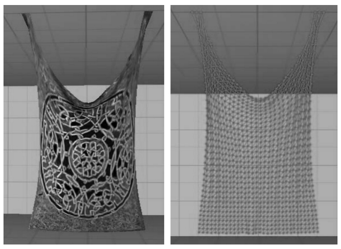


Figure 2: Cloth looking like a piece of rubbery [6].

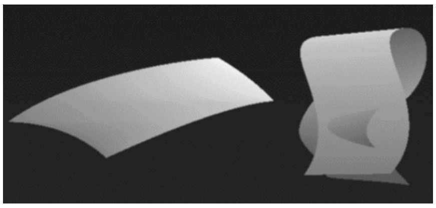


Figure 3: Self-collision may only occur in curved surfaces [4].

Another algorithm to speed up cloth self-collisions was proposed by Bridson et al. [1]. They focused on the simulation of self collisions of the cloth using an AABB tree to optimize the neighbour search. Moreover, they also put an emphasis on maintaining folds and wrinkles of the cloth when in contact with rigid objects (Figure 4). The drawback of this method is that it works only on CPU and therefore it is not applicable for modern computer games.

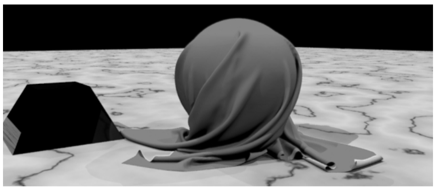


Figure 4: Cloth draped over a sphere [1].

Fuhrmann et al. [2] presented a solution of realistic cloth simulation using a combination of parameterized particle's internal and external forces. Their strategy was able to produce a robust algorithm which could handle large time steps. Also, an efficient collision avoidance was applied to enhance the realism of the simulation. Figure 5 shows several situations of cloth modelling being explored by Furhman et al. [2]. We can see that in the images a) and b) self-collision detection is on, whereas on the c) and d) is off. Although this method works really well, increasing number of particles beyond 1600 slows down the algorithm too much due to self-collisions.

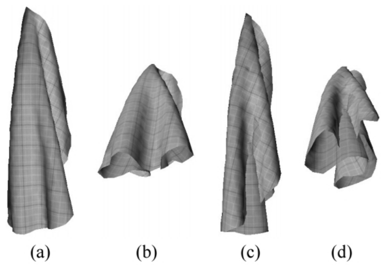


Figure 5: In (a) and (b) self-collisions are enabled. In (c) and (d) are disabled [2].

Visually plausible method was presented by Selle et al. [3] to handle high resolution cloth objects while keeping a collision history. This method allows for simulations of cloth objects with up to two million triangles (Figure 6). They used CPU parallelism to enhance the speed of their algorithm. But again, this is only CPU method which could be hard or even impossible to rewrite on GPUs and it does not perform cloth simulation in real time.

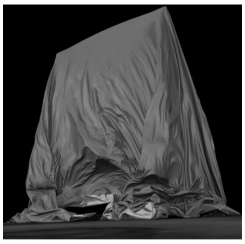
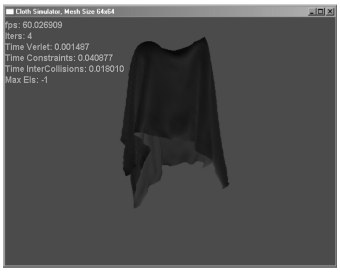


Figure 6: Cloth with two milion triangles draped over a wardrobe [3].

Another solution is presented by Rodriguez-Navarro et al. [9] which is a GPU implementation. It extends Finite Element Method (FEM) which was based on deformabled objects presented by another paper from Muller et al. [12]. This method is more physically accurate than the mass-spring one because it directly models elasticity theory and it can handle models with arbitrary structures such as triangular meshes or even volumetric models (Figure 7).



```
fps: 60.026909
Iters: 4
Time Verlet: 0.001487
Time Constraints: 0.040877
Time InterCollisions: 0.018010
Max Els: -1
```

Figure 7: Cloth self-collision response [9].

For cloth self-collisions they decided to implement on GPU an optimal spatial hashing method [13] with difference that they make a local voxelisation for each time step only in the space covered by the cloth. However, this method is efficient (171 FPS for 64x64 cloth grid) it uses old fashioned GPU programmable pipeline and a lot of tricks that makes this method cumbersome to implement. What is more it uses a lot of GPU memory because it stores some results in texture buffers, where in today's games this memory could be spent for other effects (e.g. fancy post-processing).

Nvidia was also working on GPU based cloth simulations [8] for square pieces of cloth. They also used a lot of additional memory (three texture buffers - one stores previous timestep's positions, one current position and one is a temporary texture used for calculations). Moreover, they did not support self-collisions. Similiar issue is with Ken Arthur's solution [7]. He used only Vertex and Fragment shaders and therefore he had to used a lot of additional memory to perform cloth simulation.

3. Method

To harness the full power of the GPU to perform cloth simulation with self collisions Compute Shader was used. During the first step of the above algorithm, the compute shader is dispatched several times since time step of the simulation is very small. It of course, slows down the simulation a bit cause the cloth simulation has to be performed several times per frame, but it is a price that needs to be paid if we want to observe the simulation in a reasonable time.

In order to make algorithm as easy as possible, we decided not to use shared memory in compute shader. Therefore, we had to create 4 shader storage buffers objects - two for positions and two for velocities. During multiple times when compute shader is being invoked we read the initial positions and velocities from the first position and velocity buffers, modify these data and save them to the other two position and velocity buffers (so the newest position and velocity data are in second buffers) via compute shader and after calling *glDispatchCompute()*, we issue a *glMemoryBarrier()* call to make sure that all shader writes have completed. Then, we swap buffers so in the next compute shader dispatch we will read data from second pair of the buffers and write to the first pair and so on.

The pseudo-code of this algorithm is presented in the Listing 1.

Listing 1: Final algorithm pseudo-code.

```
void main()
{
    input: position p
           velocity v
           index idx of the global invocation ID that is responsible for processing
           this particle

    output: position out p
             velocity out v

    set force to gravity * particle mass

    for each neighbour of the particle
        set r to neighbour pos − p
        set force to force + norm(r) * spring k * (length(r) − rest spring length)

        set force to force − damping * v
            set acceleration a to force * 1.0 / particle mass

        // Apply simple Euler integrator
        set p to p + v * delta t + 0.5 * a * delta t * delta t
            set v to v + a * delta t

        check if there was a self−collision
                adjust p and v using algorithm in Listing 2

        check if there was a collision with a plane
                adjust p and v

30         check if there was a collision with a sphere
31                 adjust p and v
32
33         out p = p
34         out v = v
35 }
```

In lines 3-9 we define our inputs and outputs. For each particle we provide it's position, velocity and index of the global invocation ID that is responsible for processing this particle.

In lines 13-15 we calculate forces that neighbouring particles are adding to the particle we currently processing. This is strictly based on a well-known mass-spring model [1].

Then, in lines 20-22 we are applying simple Euler integrator [14] to particle's position and velocity based on Δt.

Next, in lines 24-31 we are checking if our particle collides with any other particle in cloth mesh (self-collisions described in subsection 3.1) or a plane or a sphere. If it collides we calculate the adjusted position and velocity for this particle. In other words, we are taking backward the currently processed particle to avoid the collision.

Finally, in lines 33-34 we are returning the calculated new position and velocity for a currently processed particle.

3.1. Self-collisions

Cloth self collisions are much more complicated and much more time consuming than collisions with simple objects (like sphere or plane) that can be described with simple mathematics. However, they highly increase the realistic factor of cloth simulation since cloth will not pass through itself.

A simple solution to this problem is check if two regions of the cloth are too close to each other, apply repulsive force to the vertices around this region to encourage separation as shown in [5]. However, this method is simple it can lead to some problems because of the definition of "too close", the choice of time step, the choice of repulsive forces and can further contribute to numerical instability.

Another solution is to check any intersections that happened. If there was a collision, modify position and velocity of the particle in such a way that collision no longer occurs. This method is not very hard to implement, it is effective and it guarantees that no collision will occur. However, it is very time consuming since we have to check triangle-point intersections for every particle and triangle.

The inspiration for the algorithm that we use comes from [14], where there is only a brief description of the solution to this problem without any pseudo-code or implementation details. We can imagine that every particle is surrounded with a virtual marble and what we get is a set of connected marbles. These marbles are not considered to be touching if their associated particles are connected by a spring. Therefore, we can set the radius of a marble larger than the distance between two particles. Then the algorithm looks as follows: if the distance between the particles is less than twice the marble radius then we know that collision has occurred. Then we need to back the particles up such that this collision has not occurred and slightly decrease their velocity. The pseudo-code of this algorithm is presented in the Listing 2.

Listing 2: Cloth self-collision pseudocode.

```
void clothConstraint(ref particle in p, ref velocity in v)
{
    set marble radius to min(restLengthVert, RestLengthHoriz) * 0.45f;

        for each particle p in particles do
        if distance(in p, p) < marble radius * 2 AND in p is not the same as p
            back in p up such that collision has not occurred
            slightly decrease in v velocity
}
```

4. Results

The demo application has been tested on computer with the following specification:

- CPU: **Intel Core i5-4690K @ 3.50GHz**
- RAM: **16GB**
- GPU: **Nvidia GeForce GTX 970**
- Operating system: **Windows 10 Pro (64 bits)**

Within the application testing collisions with other objects (sphere, plane) and cloth self-collisions were implemented. This method has been tested under visual plausibility and efficiency under different circumstances. The example cloth model's topology that was used in the experiments is presented in Figure 8 (this model has 3600 particles).

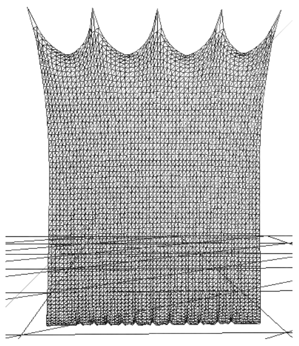


Figure 8: The example of cloth model's topology with 3600 particles.

4.1. Visual results

The visual results of the implemented cloth simulation using OpenGL Compute Shader are presented in the below Figure 9. We can see that cloth without self-collision is far from reality whereas cloth with self-collisions looks more real and its visual results are satisfactory.

Figure 10 presents visual comparisons between cloth simulations on CPU and GPU. As it can be seen on the below figures, the visual results of both versions are very similar. However, in the Figure 10 in images **b)** and **c)** row (from top to bottom) GPU version gives more visually plausible results.

4.2. Performance results

Efficiency comparison of cloth simulated using Compute Shader is presented in Table 1. On the other hand, Table 2 presents efficiency comparison between cloth with self-collisions and without self-collisions simulated on CPU. The results were made using different cloth mesh densities and with self-collisions turned on a)

b)

c)

d)

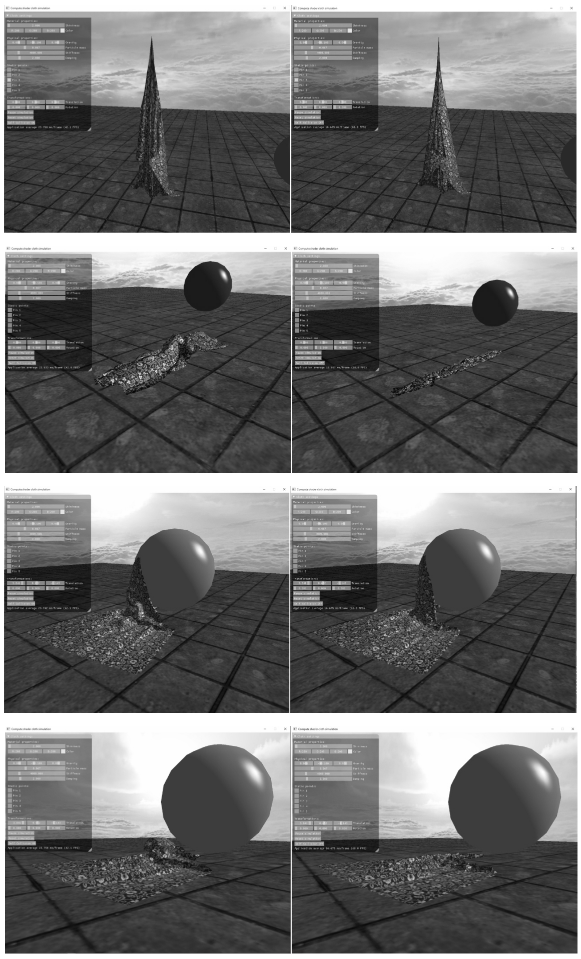
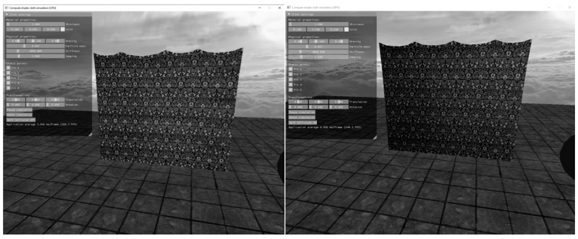
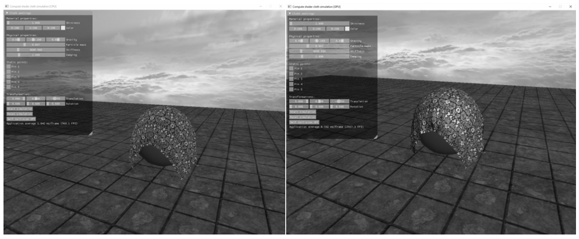


Figure 9: Comparison between cloth with self-collisions (left column) and without self-collisions (right column). Model contains 50x50 particles.

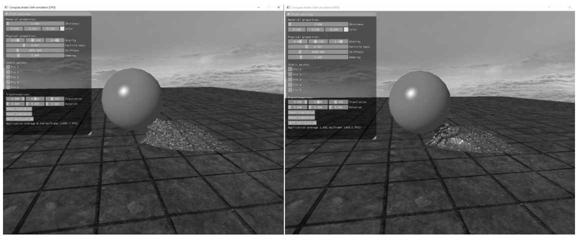
a)



b)



c)



d)

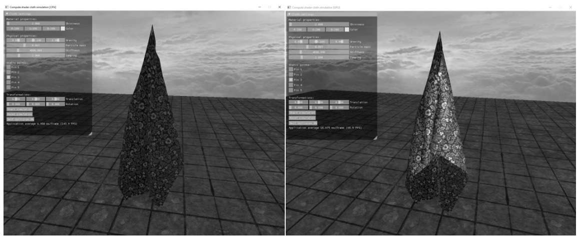


Figure 10: Comparison between CPU version (left column) and GPU version (right column). Both models contain 20x20 particles (Number of particles is low because of CPU limitations.

and off. Collisions with other objects like sphere and plane are so easy and efficient that they do not contribute to the final performance result.

Table 1: Efficiency comparison between cloth with self-collisions and without self-collisions on GPU.

Number of particles	Self-collisions ON [ms]	Self-collisions OFF [ms]	Self-collisions ON [FPS]	Self-collisions OFF [FPS]
100	1.967	0.517	502	1935
400	6.408	0.517	156	1935
900	13.700	0.542	73	1846
1600	23.950	0.542	42	1846
2500	36.900	0.542	27	1846
3600	53.108	0.542	19	1846
4900	72.308	0.550	14	1818
6400	94.575	0.575	11	1714
8100	121.400	0.608	8	1621
10000	151.325	0.658	7	1519

Table 2: Efficiency comparison between cloth with self-collisions and without self-collisions on CPU.

Number of particles	Self-collisions ON [ms]	Self-collisions OFF [ms]	Self-collisions ON [FPS]	Self-collisions OFF [FPS]
100	3.708	1.033	269	962
400	45.358	3.942	22	254
900	216.533	8.942	5	112
1600	673.783	15.708	2	64
2500	1625.892	24.458	1	41
3600	3350.125	35.254	~0	28
4900	$\gtrsim$4000.000	48.208	~0	21
6400	$\gtrsim$4000.000	62.908	~0	16
8100	$\gtrsim$4000.000	80.016	~0	13
10000	$\gtrsim$4000.000	98.508	~0	10

5. Conclusions

As the results show, cloth simulation with self-collisions simulated on GPU performs really well despite the use of the naive algorithm for checking this constraints and integration method that forces us to evaluate the same compute shader 50 times. Moreover, cloth simulation on GPU even for high cloth mesh densities without self-collisions simulates very fast. On the other hand, simulation performed on CPU has not so great performance. From 70x70 mesh dimension with self-collisions, there was impossible to measure the performance because of the lack of the computational power of CPU.

Furthermore, in the Figure 9 we can see that proposed naive algorithm for checking cloth self-collisions gives visually plausible results and it makes cloth "more real". When comparing visual results from CPU and GPU versions, it can be seen that results are very similar. However, in the Figure 10 in 2nd and 3rd row (from top to bottom) cloth simulated on GPU looks more realistically.

Above results show that optimization of rendering complex scenes using OpenGL Compute Shaders can speed up a lot of things that an ordinary CPU had problems with and in most cases it was a bottle-neck. The use of the new shader stage is something new and not seen in previous papers and has a lot of potential and does not have drawbacks of OpenCL/CUDA that has to interoperate with OpenGL to exchange data (synchronization) which has a huge impact on performance.

To conclude, the GPU based cloth simulation method presented in this article works perfectly for real time applications like computer games. It is really fast and it gives a lot of room to improvement and in our humble opinion it will be the way in which modern simulations or algorithms will be developed and improved. Presented method has a huge potential in further development. It is very good and computationally efficient to simulate cloth meshes with huge number of particles/vertices.

References

[1] Bridson, R., Fedkiw, R., and Anderson, J., *Robust Treatment of Collisions, Contact and Friction for Cloth Animation*, ACM Trans. Graph., Vol. 21, No. 3, July 2002, pp. 594–603.

[2] Fuhrmann, A., Groß, C., and Luckas, V., *Interactive Animation of Cloth Including Self Collision Detection.* In: WSCG, 2003.

[3] Selle, A., Su, J., Irving, G., and Fedkiw, R., *Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction*, IEEE Transactions on Visualization and Computer Graphics, Vol. 15, No. 2, 2009, pp. 339–350.

[4] Volino, P. and Magnenat-Thalmann, N., *Interactive Cloth Simulation: Problems and Solutions*, In: Virtual Worlds on the Internet, IEEE Publisher, 1998, pp. 175–192.

[5] Baraff, D. and Witkin, A., *Large Steps in Cloth Simulation*, In: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '98, ACM, New York, NY, USA, 1998, pp. 43–54.

[6] Lv, M., Li, F., Tang, Y., and Bi, W., *A fast self-collision detection method for cloth animation based on constrained particle-based model*, In: Digital Media and its Application in Museum & Heritages, Second Workshop on, IEEE, 2007, pp. 140–145.

[7] Arthur, K., *GPU-Based Cloth Simulation and Rendering*, Retrieved from http://kenarthur.bol.ucla.edu/, 2008.

[8] Zeller, C., *Cloth simulation*, White paper NVIDIA, 2007.

[9] Rodriguez-Navarro, J. and Susin, A., *Non structured meshes for Cloth GPU simulation using FEM*, "3rd Workshop in Virtual Reality Interactions and Physical Simulation". Madrid: EUROGRAPHICS, 2006, p. 1-7, 2006.

[10] Gast, T. F., Schroeder, C., Stomakhin, A., Jiang, C., and Teran, J. M., *Optimization integrator for large time steps*, IEEE transactions on visualization and computer graphics, Vol. 21, No. 10, 2015, pp. 1103–1115.

[11] Volino, P., Courchesne, M., and Magnenat Thalmann, N., *Versatile and Efficient Techniques for Simulating Cloth and Other Deformable Objects*, In: Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '95, ACM, New York, NY, USA, 1995, pp. 137–144.

[12] Müller, M., Dorsey, J., McMillan, L., Jagnow, R., and Cutler, B., *Stable Realtime Deformations*, In: Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '02, ACM, New York, NY, USA, 2002, pp. 49–54.

[13] Teschner, M., Heidelberger, B., Müller, M., Pomerantes, D., and Gross, M. H., *Optimized Spatial Hashing for Collision Detection of Deformable Objects.* In: VMV, Vol. 3, 2003, pp. 47–54.

[14] Fisher, M., *Cloth*, Retrieved from https://graphics.stanford.edu/~mdfisher, 2014.