Physics Simulation Project in EDAN35 - High Performance Computer Graphics

Johannes Blühdorn Erik Molin

Lund University
Sweden

Abstract

In this paper, we describe a project for the course EDAN35 - High Performance Computer Graphics, given in the fall of 2016 at Lund University. We combine a simulation of the physical behaviour of a piece of fabric, both with respect to itself and with respect to other objects in the environment, with methods to enhance its graphical appearance. The physics simulation is parallelized and implemented in OpenCL. The fabric is rendered using OpenGL, and hardware tesselation and procedurally generated textures are deployed to achieve a realistic look.

1 Introduction

This report concerns a project in the course EDAN35 - High Performance Computer Graphics, given in the fall of 2016 at Lund University. Our project addresses several issues that arise in connection with simulating fabric, both from a physical and a graphical point of view. A realistic simulation of fabric is of interest when it comes to rendering cloth, curtains, or other planar, non-rigid materials in computer generated environments like games. Physically plausible behaviour of the material and physically plausible interaction with the environment add great levels of realism here. Note that the focus of this paper is a plausible visualization of the underlying physical process, as of interest in computer graphics, rather than an accurate simulation of the physical process itself. The general framework and some solution techniques can also be used as a starting point towards a more physics focused approach, but as for now, the methods deployed here aim at a feasible real-time simulation of fabric. This means there is a trade-off between both accurate physics and photo-realistic fabric on the one hand while on the other hand, the time constraints have to be met. Physical exactness is dropped in favor of computational feasability. By parallelizing the entire simulation, we address this trade-off. The goal of parallelism is fully using available computational resources, which speeds things up and allows for a low-scale simulation of the fabric. We are also able to populate the scene with additional objects the fabric can interact with.

2 Algorithms

2.1 Physics Simulation

We model the piece of fabric as an $n_1 \times n_2$ equidistant grid of particles. All particles have the same mass m and the same radius r, in a way that along each dimension of the grid, neighbouring particles touch each other. The distance constraint between neighbouring particles is modelled by a spring of length $2r$ that is spanned between them. Consider Figure 1 for an example.

In the following we explain the structure of the physics simulation and its basic ideas. Alone, they don't make the simulation work well yet — further details are explained later in Section 4.1. The physics simulation is based on two primary parts. First, the particles move according to Newton's law of motion, i. e. each particle's movement is described by the ordinary differential equation

$$F = m\ddot{x}. \tag{1}$$

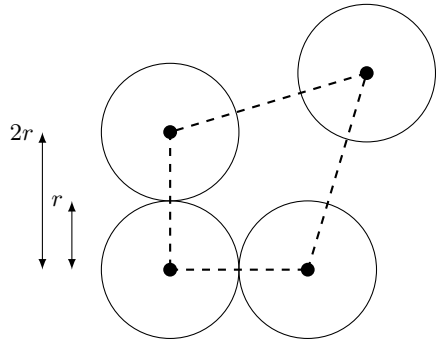

Figure 1: 2×2 fabric with two violated constraints. Spring connections dashed.

The force F is computed as the gravitational force on the particle plus all forces caused by springs that directly connect to that particle. Starting from there, the update rules over one timestep of length Δt for the numerical integration of (1) are

$$\begin{aligned} \dot{x}_{\text{new}} &= \dot{x}_{\text{old}} + \Delta t \cdot F/m, \\ x_{\text{new}} &= x_{\text{old}} + \Delta t \cdot \dot{x}_{\text{new}}. \end{aligned} \tag{2}$$

This integration scheme is known as the semi-implicit Euler method.[1] To summarize, this approach is closely inspired by the laws of mechanics together with an established scheme of numerical integration, so we have good reason to expect results with a certain level of quality and believability.

Second, we test for collisions, both between pairs of particles and between particles and fixed parts of the environment. We check for particle-particle collisions by comparing the distance between two particle centers to $2r$. In the case of a collision, the component of the particles' speed along the line connecting them is averaged so that the speed difference in collision direction is zero, and the position of both particles is corrected along the line connecting them so that they no longer collide. Let x_1 and x_2 be the positions of two colliding particles. Let

$$\begin{aligned} \Delta v &= \dot{x}_1 - \dot{x}_2, \\ \Delta x &= x_1 - x_2. \end{aligned} \tag{3}$$

[1] https://en.wikipedia.org/wiki/Semi-implicit_Euler_method

The update is then given by

$$\dot{x}_1 \leftarrow \dot{x}_1 - 0.5 \cdot \left(\Delta v \circ \frac{\Delta x}{|\Delta x|}\right) \cdot \frac{\Delta x}{|\Delta x|}, \tag{4}$$

$$\dot{x}_2 \leftarrow \dot{x}_2 + 0.5 \cdot \left(\Delta v \circ \frac{\Delta x}{|\Delta x|}\right) \cdot \frac{\Delta x}{|\Delta x|}, \tag{5}$$

$$x_1 \leftarrow x_1 + 0.5 \cdot (2r - |\Delta x|) \cdot \frac{\Delta x}{|\Delta x|}, \tag{6}$$

$$x_2 \leftarrow x_2 - 0.5 \cdot (2r - |\Delta x|) \cdot \frac{\Delta x}{|\Delta x|}. \tag{7}$$

The collision with fixed parts of the environment, e. g. the floor, works analogously to the particle-particle collision except that such fixed parts do not start moving due to collision. What was before the direction connecting the particles' centers becomes now the normal vector of the fixed surface we collide with, and the averages in Equations (4) to (7) that equally distribute the velocity and position change between the two involved particles now fully affect the single involved particle instead, so 0.5 has to be replaced by 1.

The collisions computations are structured as follows: Before the simulation starts, we choose a fixed discretization of the subset of the three-dimensional space that is relevant to the simulation. That means, the part of space where most of our interesting physics will happen, is subdivided into several axis-aligned cubes of smaller size. Everything else around that finely discretized area is just considered to be the "rest". During movement, we determine via simple comparisons on the components of the particles' positions the cube that each particle ends up in after the movement step, or assign it to the "rest" if it is not contained in any cube. We keep track of this via a list for each of the partition's subsets. Then, pairwise particle collisions are computed on each of the subsets of the space separately. That is, for each subset, we iterate through its list of particles. For each particle in the list, we carry out the above pairwise collision procedure for this particle with all other particles that follow it in the list.

We now discuss how this simulation can be parallelized. The particles' forces for the next simulation step can be accumulated in parallel. The gravity component only depends on the particle itself, and can be computed for all particles in parallel. The computation of the spring forces affecting a fabric particle depends on the springs it is connected to, and in that indirectly on the distance to its neighbor particles after the last step. In the end, we parallelized along the rows and columns of the fabric grid, that is, there is one thread for each row that accumulates the forces caused by all springs along that row, and after all rows are done, there is analogously a thread for each column in the fabric. An alternative approach, thinking in terms of particles and not in terms of springs, would be setting up one thread for each particle. That thread would accumulate the forces for that respective particle, which would work because particle positions and can be read by multiple threads at the same time. The numerical solver given by Equations (2) can be computed for all particles in parallel. The particle-particle collisions are inherently sequential within each subset of the space, but the computations for all subsets can be done in parallel since the subsets are independent. The collisions with fixed objects in the scene can be carried out for all particles in parallel.

Apart from the particles that belong to the piece of fabric, additional unconstrained particles can be included in the simulation easily. They are subject to the very same computations, but in the computation of the force, there are no contributing springs.

2.2 Tessellation

Due to the high computational cost of performing physics simulation on a vertex, the number of simulated vertices are preferably kept to a minimum. However, for rendering of edges and bumps in the fabric, a maximum amount of vertices is preferred. Our proposed solution is to use the vertex attributes from the simulated vertices to tessellate the surface.

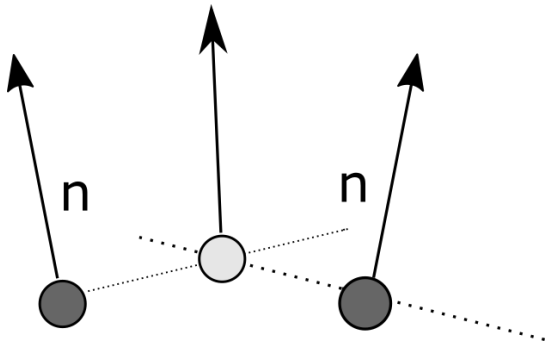

Figure 2: Schematic of the interpolation mechanism - an interpolated vertex is positioned in the middle of two vertices, and then shifted by the sine of the angle between the normals. The normal of the interpolated vertex is the average of the original normals.

For our algoritm, each simulated vertex v carries two attributes of use: Their position p_v and their normal n_v. Tessellated vertices $\hat{v}$ are interpolated by a two step process. First, their position on the surface is linearly interpolated from the simulated vertices. Second, their height above the surface is determined by comparing the simulated vertices' normal vectors:

- If they are parallel, the tessellated vertex should be on the surface.
- If they diverge, the tessellated vertex should be above the surface.
- If they converge, the tessellated vertex should be below the surface.

This is implemented as follows:

$$n_{\hat{v}} = \frac{n_{v_1} + n_{v_2}}{2},$$
$$p_{\hat{v}} = \frac{p_{v_1} + p_{v_2}}{2} + n_{\hat{v}} \cdot |n_{v_1} \times n_{v_2}|.$$

In this implementation, one interpolated vertex per edge is used. The method can be generalized to include more interpolation points, using barycentric interpolation of normal vectors and displacements.

2.3 Shading

The shading for the fabric is quite straightforward. As a base, it uses a Blinn-Phong shader. A gloss is added, where the gloss is calculated according to

$$\text{gloss} = \max(\text{camera\_position} \circ \text{normal}, 0).$$

In addition to this, normal and shadow mapping is used. Textures are procedurally generated beforehand, and consist of three components,

1. a texture of white noise,
2. a texture of $\sin^2$ functions.
3. a normal map that is the numerical derivative of 2.

The textures and light components are then blended in the fragment shader, using experimentally determined coefficients and texture indexing parameters.

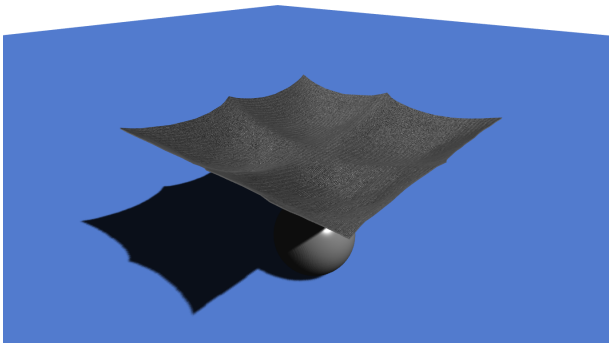

Figure 3: Piece of fabric, fixed at 8 points along its edge in space, hanging above a sphere.

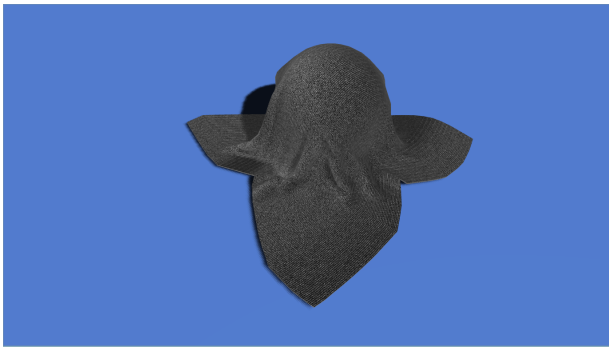

Figure 4: Piece of fabric, dropped onto the sphere.

3 Results

Consider Figures 3, 4, 5 and 6 for some examples of how the fabric interacts with its environment. In the configuration of Figure 5, there is a lot of tension in the fabric. This causes parts of it to oscillate within small bounds, as can actually be seen in Figure 5 at both the right and leftmost corner of the piece of fabric. These oscillations can probably be addressed by incorporating some damping in the computation of spring forces. In this configuration, particles also tend to leak through the fabric. On first thought, this looks like sufficiently fast particles "tunnel" through the spring potentials from the fabric constraints, but it actually turns out that this is specific to the structure of the collision computations, and will be discussed in Section 4.

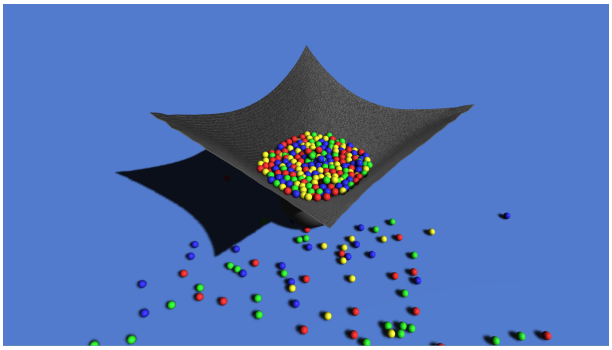

Figure 5: Several balls dropped onto the piece of fabric. Its four corners are fixed in space.

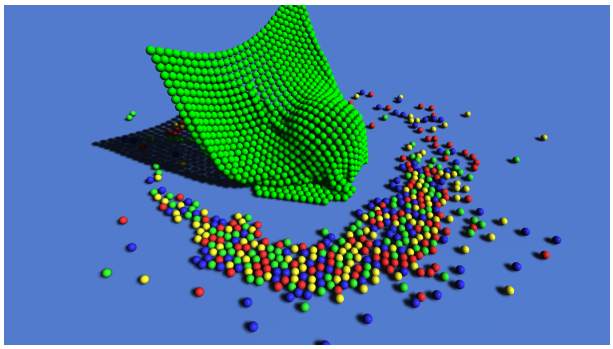

Figure 6: Visualization of the underlying particle structure. The upper two corners of the fabric are fixed in space.

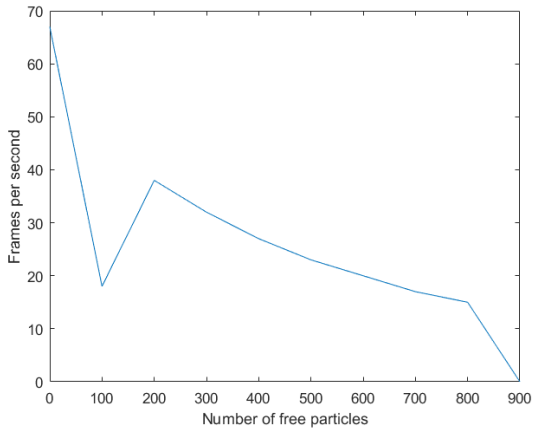

Figure 7: Framerate for several numbers of free particles.

One very interesting question is: How well does the simulation scale to larger numbers of particles? On the Intel Core i7-4510U CPU that includes an Intel HD 4400 GPU, a 29×29 fabric together with 500 free particles, giving a total of 1341 particles, has stable framerates around 20. There is quite a variance in the framerate though since the expense of the collision computations depends strongly on how densely the subsets of the partition of the space are populated with particles. Consider Section 4 for additional discussion of this. In particular, it is not trivial to obtain a setting in which a relation like particle count versus framerate actually contains useful information on the scalability. If both the number of free particles and the number of fabric particles change over a large range of values, you barely obtain comparable scenarios. To simplify things, we stick to a 29×29 fabric and vary the number of free particles. The testing scene is the one of Figure 4, but with the additional free particles dropped onto the fabric. Figure 7 shows a plot of the frames per second against the number of free particles. As of now, we do not have an explanation for the sudden and severe drop in performance as the number of free particles is increased from 0 to 100. Suspicion is on external factors like CPU energy management, but we are not sure about that.

4 Discussion

4.1 Physics Simulation

In practice, the simulation algorithm from Section 2.1 needs some small additional changes in order to yield good results. They are related to two major issues with our simulation approach which will

be discussed in all detail below. In several experiments, the following configuration was determined to work well. This is partly parameter dependent, so note that we use $r = 0.05$, $m = 0.0001$, a spring constant $K = 100$, 29×29 fabric, 500 free particles and a $20 \times 20 \times 20$ discretization of the space around the fabric with a cube sidelength of 0.4.

- In each simulation step, each velocity decays after the numerical integration and before the collision detection according to
$$\dot{x} \leftarrow (0.999)^{1000 \cdot \Delta t} \dot{x}.$$
The parameters 0.999 and 1000 yield good results, and the exponential is used to ensure that the velocity decays with a constant speed, independently of Δt. This can be thought of as a unified model of effects like friction or air resistance and helps the simulation reach equilibria.

- Each velocity component is clamped to the interval $[-20, 20]$ after velocity decay. This helps the numerical stability of the simulation and avoids a blow-up of the velocities due to numerical error. As we will see soon, this is actually an inherent problem.

- It turned out that a soft collision handling between particles yields better results. That means that the positions are not fully corrected in a single step. Instead, 0.5 in Equations (6) and (7) is replaced by 0.1. We attribute this to the fact that pairwise correction of particle collisions just takes two particles into account, but correcting their positions might actually move them straight into other particles with which they did not collide before. A more gentle position correction enables the algorithm to regard these secondary collisions as well.

- The simulation is subject to the constraint $\Delta t \leq 0.001\, s$. We will discuss this soon. For now, it means that the physics simulation in general runs several times for each frame that is rendered.

As explained in [Witkin 2011], using springs to model constraints between objects is a sloppy way of constraint handling. Springs do not forbid violated constraints, they only discourage them by imposing an energy penalty on violated constraints. First, the constraints are technically allowed to be violated if the violating force is large enough and cannot be compensated by the spring. Second, even if the spring is strong enough to correct the violation, it might still take the particles some time to move to a configuration of lower energy. The latter can be compensated by using high spring constants, which in turn leaves us with a so-called stiff ordinary differential equation. Our numerical method for solving the ordinary differential equation is a so-called semi-implicit one. This means that its first part, the velocity update, is explicit, whereas the second part, the position update that uses the already updated velocity, is implicit. Explicit solvers, and also this semi-implicit one, have the advantage that their implementation is straightforward and does not involve additional costly operations such as solving a nonlinear system of equations, something which fully implicit solvers do but we cannot afford while at the same time meeting the real-time requirement of the simulation. Explicit solvers have the disadvantage that they cannot handle stiff ordinary differential equations independently of the stepsize — there is a threshold on Δt above which the solution will not be stable, and this is exactly which we observed in our implementation.

[Witkin 2011] suggests an alternative way of constraint handling via Lagrange multipliers. The idea is maintaining a constraint function that takes all particle positions as arguments and is zero if and only if the particles are in a valid configuration. We can use the partial derivative of this constraint function to set up a linear system of equations for the particle forces. The solution are then force directions that do not violate constraints, and we can think of this as a model for the force transfer between particles that are tied together by a constraint. This approach, however, has two drawbacks. First, it is a global method that takes all particles into account at once, so both the partial derivative of the constraint function and the linear system of equations that has to be solved are very large. Nonetheless, we actually built it and managed to handle the equation solving part in an efficient way by using an OpenCL accelerated conjugate gradient solver. The main problem arised with collision handling. This setting suggests two approaches towards collissions. First, while it is possible to incorporate the collisions into the constraint function, it becomes so large in the course of that and so costly to compute that we can no longer simulate in real time. Second, we can use two interacting methods, one for movement simulation and one for collision handling, as in Section 2.1. The key here is the compatibility of the two methods, i. e. that one does not undo the work of the other in a way that things get worse and worse. In this setting, however, we did not find a collision handling scheme that is compatible with the movement computations (the one from Section 2.1 is not, for example), and we decided to drop the approach from [Witkin 2011] in the end. Nonetheless, the comparison gives us insight in some of the characteristics of our simulation method.

Second, we would like to spend some time on additional discussion of the collision handling scheme. A characteristic of our approach is that it is separate from the movement computations — a priori it is unclear if the ODE solver and the collision computations are compatible, or if both iteration schemes actually work against each other. The small tricks and choices described at the beginning of this section show, however, that the rather intuitive design of the simulation algorithm can be actually modified to yield good results.

Depending on how you implement collisions, you achieve different degrees of parallelizability. One question is: Should you check for the collision between two particles only with respect to their positions from the last simulation step or should you check for collision using particle positions that have already been updated due to other collision computations in this step? The first approach can be carried out for all particles in parallel, whereas the second approach is inherently sequential. Unfortunately, it turns out that the second approach, i. e. taking already updated particle positions into account for the remaining collisions, yields a significantly more stable simulation and helps the two iterative methods with working together well. Worse, the collision detection is the most computationally expensive part of the whole simulation. Why is that? The other steps scale linearly with respect to the number of particles or springs, but the pairwise test for collision of each particle against all other particles scales quadratically with the number of particles, so there is really need for optimization. We achieve this by moving to a more local way of collision handling. Why should you test for the collision of particles that are far away from each other? By keeping track of subsets of the space that contain only a certain number of particles, we can reduce the computational effort by testing pairwise collisions only between particles within a single subset. More important, this allows for a parallelization of the otherwise sequential algorithm since all subsets are independent of each other. A drawback is that some collisions might not be corrected at the borders of the subsets. This explains the "tunnelling" observed in Section 3. We address it by shifting the cubes of the discretization around from simulation step to simulation step so that each collision is at least handled properly every second simulation step. This does not avoid the "tunnelling" entirely, but reduces it a lot. The most general approach towards this would be an adaptive discretization of the space, that gets finer in areas of high particle density, and gets coarser in areas of low particle density. This is especially of interest if the scene is large and/or highly dynamic, and most of the physics computations are concentrated on a relatively

small subvolume.

Finally, although the physics simulation is fully parallelized, it still turned out to be faster if the OpenCL implementation is run on the CPU instead of the GPU. This observation is with respect to the Intel Core i7-4510U that comes with an Intel HD 4400. There are several possible reasons for this. One might be that in order to achieve good GPU performance, you have to design carefully around things like sizes of local caches or preferred workgroup sizes to obtain a speedup, and our implementation is not yet tuned towards that. Another might be that the GPU is already involved in OpenGL rendering and does not have the capacity to handle the simulation at the same time whereas CPU cores other than the one running the application's main thread are completely idle, and OpenCL can bring them into action.

On the Intel CPU/GPU, memory latency is less of an issue compared to other architectures with dedicated graphics cards. We can not only modify OpenGL vertex buffers at close to zero latency directly via the `cl_khr_gl_sharing` extension (even if OpenCL runs on the CPU) but also map the OpenCL buffers with close to zero latency to C pointers to edit them manually (even if they were part of the GPU's memory). This is due to the fact that on the Intel CPU/GPU, the GPU memory is contained in the system's main memory. This has to be kept in mind when switching to another architecture.

4.2 Tessellation

The proposed tessellation algorithm works well for curved edges where the divergence of normals is small. The current implementation in the geometry shader only includes one interpolation point per edge. By use of a tessellation shader, more points could be easily implemented, leading to a smoother appearance. This is, however, beyond the scope and time constraints of this project.

Sharp turns in the fabric are also a cause for concern. The proposed interpolation algorithm always places the interpolated vertex precisely between the simulated vertices. This will cause a symmetric curve between the simulated points. This is appropriate many times, but does not easily allow for sharp creases in the fabric. This could be addressed by implementing a second degree interpolation.

One approach is to include more information in the simulated vertices, including the position of adjacent simulated vertices, allowing for true quadratic interpolation. Another approach is to use the world coordinates of the simulated vertices to create a mid-vector, and comparing the two normals individually to this, thus allowing some dissymmetry in the interpolation.

The results from quadratic interpolation should model creases better, but would require a redesign of the entire system, and increase the memory footprint of the vertices. The mid-vector approach could be implemented in the geometry shader only, but the potential of the method is speculative at best.

4.3 Shading

While the shading looks satisfactory, there are several optimizations that could be made, both regarding use of resources and usability. As of the time of writing, the generated textures are 1024×1024 RGB textures. Because of the low information content in them, this is a huge waste - the wave texture can be stored in enough pixels to contain a wavelength, with the normal map only a few bytes larger. The noise texture could also be made significantly smaller, while avoiding repetition by clever sampling.

The current configuration is quite well tuned to achieve a certain fabric look. The shader components used are suitable for, or adaptable to, other fabrics. However, in the current state, the shader-intrinsic parameters (e. g. texture step size, texture blending factors, sampling directions, light blending coefficients) are quite non-intuitive. A system for mapping these intrinsic parameters to constructed shader-extrinsic parameters (for example glossiness, roughness, thread size) would significantly simplify texture authoring.

5 Conclusion

This project is an approach towards physically based rendering of fabric. Easy and intuitive methods produce results of a decent quality, but certainly there is space for further optimization. The several issues and pitfalls are discussed at length in Section 4.

Several components, for example the lightweight OpenCL accelerated linear algebra library that comes with a conjugate gradient solver, the OpenCL related classes or the OpenGL related classes, might turn out to be useful in other projects as well.

6 Acknowledgements

First of all, we would like to thank Michael Doggett for his well-taught and interesting lectures Computer Graphics and High Performance Computer Graphics.

Second, the OpenGL tutorials at `www.learnopengl.com` are extraordinarily well-written. They were our key resource in setting up our own framework for OpenGL programming. Thanks go to Joey de Vries for providing them.

Third, Blender tutorials by Senad Korjenic[2] gave some inspiration on procedural fabric texture generation and shading.

Finally, the free online version of the OpenCL programming book[3] and AMD's introduction to OpenCL[4] were helpful resources for getting familiar with OpenCL.

References

WITKIN, A., 2011. An Introduction to Physically Based Modeling: Constrained Dynamics. http://www.cs.cmu.edu/%7Ebaraff/pbm/pbm.html. Accessed: December 2016.

[2] `http://planetblender.com/makingproceduralfabricmaterials/`

[3] `https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/contents/`

[4] `http://developer.amd.com/wordpress/media/2013/01/Introduction_to_OpenCL_Programming-Training_Guide-201005.pdf`