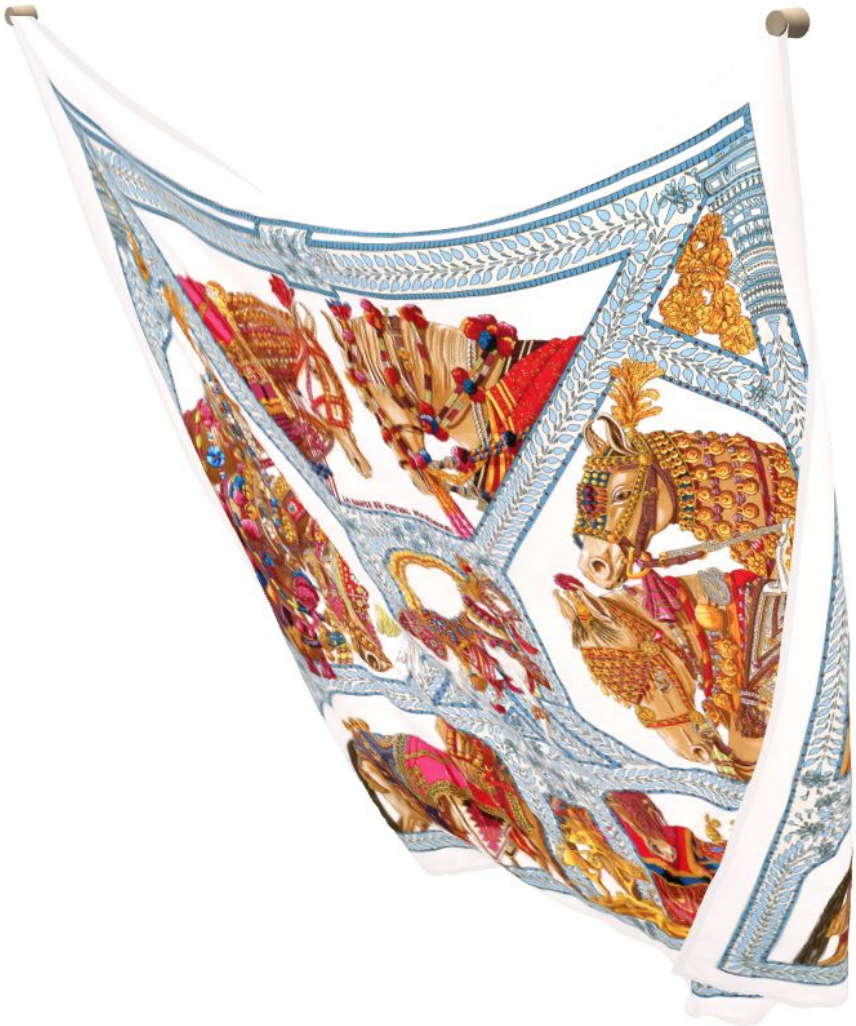
Simulating cloth in WebGL

Mass spring cloth simulation in WebGL using semi-implicit Euler and ping-pong FBO's



Name: Tim van Scherpenzeel

Student number: 326718
Course: Art & Technology
University: Saxion Hogeschool Enschede
Graduation company: Random Studio
Date: 24th of October 2016

Simulating cloth in WebGL

Mass spring cloth simulation in WebGL using semi-implicit Euler and ping-pong FBO's

Full name: Tim Wessel van Scherpenzeel
Student number: 326718
Email: timvanscherpenzeel@gmail.com
Website: www.timvanscherpenzeel.com

Graduation supervisor:
- Matthijs van Veen

Company head:
- Daan Lucas

Company supervisors:
- Ralph Kok - Senior developer
- Mark Hinch - Senior back-end developer
- Evert van der Horst - Junior front-end developer and mentor

Course: Bachelor of Art & Technology
University: Saxion University of Applied Science (Enschede, The Netherlands)
Graduation company: Random Studio (Amsterdam, The Netherlands)
Date: 24th of October 2016

Preface

This bachelor's thesis has been written during my graduation internship as front end developer at Random Studio in Amsterdam from the 6th of June 2016 until the 21st of October 2016.

First and foremost I would like to thank Random Studio for giving me the opportunity to graduate and truly become a part of the studio team. I would like to thank all that have been involved and supported me specifically Evert van der Horst, mentor and front end developer at Random Studio and Ralph Kok, senior creative developer for guiding me in the process. I would like to thank Coen Grift, creative developer, for stating the problems he has been facing with cloth simulations on the CPU allowing me to define a concrete research question.

I would like to thank Matthijs van Veen and Ruben Sinkeldam for coaching me during my graduation internship.

I have been offered a job as junior creative developer at Random Studio and will be starting on the 5th of December 2016.

Amsterdam, 23 October 2016

Tim van Scherpenzeel

Table of contents

Summary

This bachelor's thesis exists as a part of the ongoing research and development at Random Studio, a digital production and development company from Amsterdam focusing on websites and interactive installations. The studio works for international commercial high-end brands like Tommy Hilfiger, Hermès, Remy Martin and Nike as well as public cultural institutions like Het Nieuwe Instituut and Andere Tijden.

The preliminary research consists out of three parts. I'll first of all introduce what cloth is, how a cloth simulation is constructed in a mathematical and physical sense, what forces play a role when simulating cloth and how they are applied. Next I'll describe the components of the WebGL rendering pipeline necessary to do computations on the GPU that require having multiple states of the program available. Finally I'll touch on the subject of framebuffer objects and how they play a role when doing computations on the GPU.

After the preliminary research the research questions are redefined in detail and the scope of the research is declared. The main research question is as follows:

How does one realistically simulate a Hermès silk scarf using a Newtonian mass-spring system at 60 frames per second utilizing the GPU through WebGL in modern web browsers running on modern (mobile) devices?

The main question is divided into three questions with each multiple sub-questions. Sub question one deals with how one effectively moves calculations from the CPU over to the GPU. Secondly the development process of simulating cloth on the GPU using FBO's. The third question tests and documents the limitations and bottlenecks of my implementation of a cloth simulation in WebGL.

The graduation product consists out of a working prototype of a cloth simulation implemented in WebGL running at 45 - 60 frames per second with 10.000 particles (100 x 100 particle grid) on iOS devices and desktop computers. The cloth is a constructed out of a mass spring system with structural, shearing and bending properties. The positional and velocity values are stored using data textures as framebuffer objects and are extrapolated and numerically integrated using semi-implicit Euler allowing for almost all the computation necessary to simulate cloth to be done on the GPU. At the end of a single render cycle the pointers to the textures are swapped, also referred to as ping-ponging framebufferobjects.

A live demo is available on https://timvanscherpenzeel.github.io/Thesis/. One can find the source code attached or alternatively on https://github.com/TimvanScherpenzeel/Thesis

1. Introduction

1.1 Background

Random Studio is a digital production and development company located in Amsterdam, near Westerpark, focusing on websites and interactive installations. They mainly work for international commercial high-end brands like Tommy Hilfiger, Hermès, Remy Martin and Nike as well as public cultural institutions like Het Nieuwe Instituut and Andere Tijden, a television program by VPRO.

Random Studio describes itself as “a team of visual artists, developers and engineers creating experiences that blur the boundaries between art, design and technology, both physical and nonphysical”. *(Random Studio, 2015)*. They have a contemporary design style and love to experiment and prototype. The studio likes to be different and stand out from the crowd in what they do, how they approach problems, get along and work with each other and how they interact with the client.

Within Random every branch (concepting, design, development, content production, interactive installation development and construction) has its own small team of specialists that are called in by the responsible producer for the project. He or she is the one who has the control, schedules meetings and communicates directly with the client.

In this context I have been the front end developer intern between the 6th of June 2016 and the 21st of October 2016. I am called in at three phases of the project: prototyping, production and maintenance. In the prototyping phase I am responsible for the creation and early technical feasibility testing. In production front end is responsible for the technical implementation of the graphical design and assets. Projects with a longer scope often require maintenance and the implementation of feature requests from the client.

The studio, as shown in *Figure 1.1.1 and Figure 1.1.2,* is physically a very open and green environment to work in. The atmosphere is open, international and accepting to all. The lunches are lovely; the people hip, young and playful: visionaries and specialists.

Work hard. Play hard.

Figure 1.1.1: 'The pit' at Random Studio. (Random Studio, 2014)

Figure 1.1.2: The meeting room at Random Studio. (Random Studio, 2014)

1.2 Problem indication

Over the course of the last 6 years or so Random Studio has been working more and more with clients from the high-end fashion industry on websites and interactive installations. Examples of these clients are Tommy Hilfiger, Hermès and Nike but also fashion warehouse De Bijenkorf.

Some of these brands would like to display clothing, such as polo shirts, silk scarfs or new materials for shoes, in an interactive digital manner. The reason why these brands are interested in shifting from the traditional physical experience of feeling the material and seeing how a piece of clothing looks on a mannequin is because these type of experiences are very expensive and localized. Shifting these physical experiences over to digital experiences enables these brands to reach far more potential customers whilst reducing the costs of needing many physical showroom and shops.

In recent years for the most part these interactive experiences have been worked out in the form of physical interactive installations as shown in *Figure 1.2.1*.

Figure 1.2.1: Interactive installation for De Bijenkorf by Random Studio (Random Studio, 2015).

The studio has gotten (some) its fame and client base based on the interactive media work developers did using Adobe Flash Player, a multimedia plugin for the browser. An example of such an experience on the web is shown in *Figure 1.2.2*, a promotional website for Dom Pérignon.

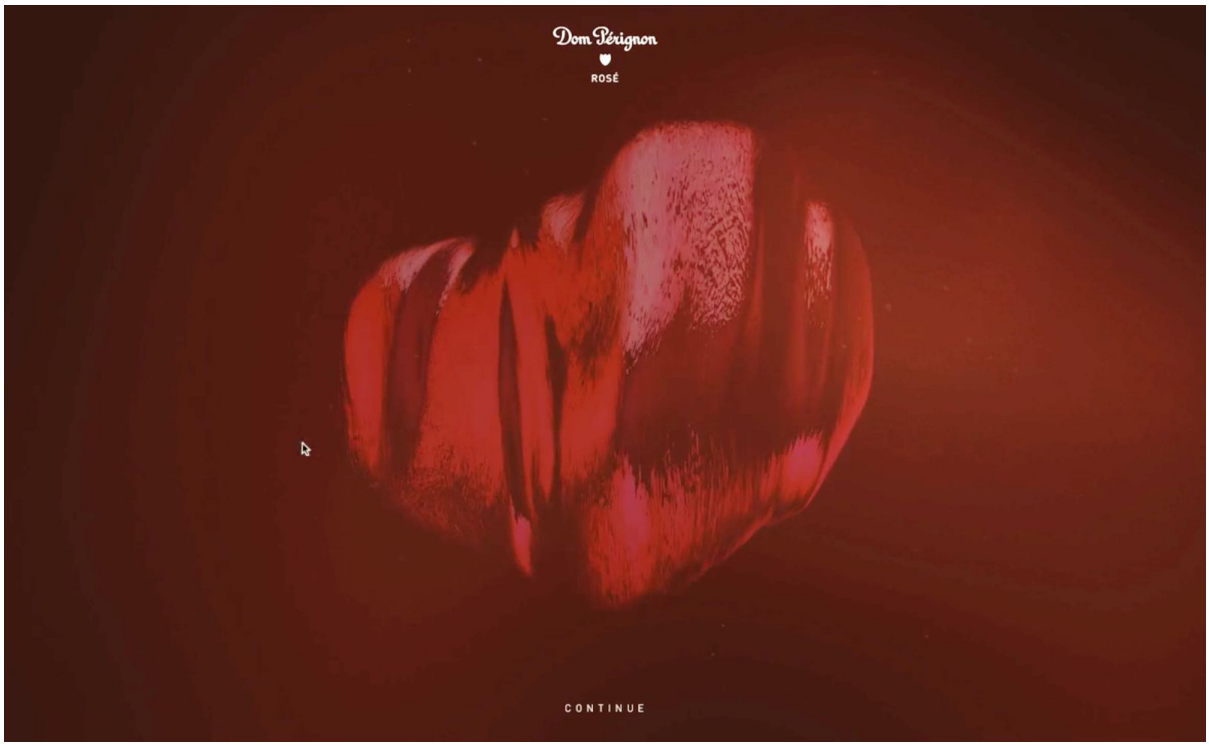


Figure 1.2.2: Interactive promotional website for Dom Pérignon Rosé 2002 using Adobe Flash Player. (Random Studio, 2013).

In the last two years Random Studio has taken a large interest in WebGL as an alternative to Adobe Flash Player as a way to embed multimedia content due to it being an external closed source plugin that is losing browser support and simply does not perform up to modern day standards. Luckily, due to major improvements in consumer hardware (especially smartphones and tablets) and feature standardization across browsers it has become possible to use a real alternative to Adobe Flash Player that is not an external plugin: WebGL. One of the most important features of WebGL is that it utilizes the parallel processing power of the GPU.

The products of high-end fashion brands are always material-quality focused: be it Hermès leather or silk scarves, Tommy Hilfigers polo- or dress shirts or the new sole of a Nike shoe. Being interactive, these materials - often cloth-like - require to be receptive to forces and responsive to user input.

Simulating the movement and physics of a physical cloth-like material in a virtual space is referred to as a cloth simulation. In the past two years Random Studio has been implementing cloth simulations on the CPU for a range of prototypes and client work. A major downside to using the CPU for cloth simulations is the limited amount of points (roughly 20 by 20, 400 particles) and forces it can compute per frame before falling below the target framerate of 60 frames per second on average modern devices. Random Studio

would like to increase the amount of points and spring constraints to create a more realistic and smooth cloth simulation whilst maintaining the target frame rate to create a more pleasant user experience.

Random Studio and I are planning to do this is by moving as much of the calculations as possible, required to simulate cloth, to the GPU and eliminate many of the communication bottlenecks between the CPU and GPU.

The research report and foundational research pieces will be used as a guideline for simulation (particles, cloth, fluids or gasses) development at Random Studio in WebGL. The research itself will form a solid starting point for further research of moving computations required to do simulations from the CPU to the GPU. It is preferred to have a working prototype or a proof of concept and therefore it will be my aim throughout the project to deliver that. My aim is to deliver a prototype of a stable cloth simulation that looks more realistic due to a higher particle count and yet have higher framerates.

Projects currently in early development that are planning to use cloth simulations are the Digital Shopping Window for Hermès and the Digital In-Shop Interfaces for Tommy Hilfiger. Attached one can find the results of visual prototyping from the designers in the folder "*Simulating cloth in WebGL - video"*. A working prototype running at a high frame rate would be a major contribution in the development process of these projects.

1.3 Preliminary problem statement

The preliminary problem is stated as follows:

Random Studio would like to know how to improve the performance of cloth simulations in the web browser. Currently developers utilize the CPU to compute the performance intensive calculations required to simulate cloth resulting in low framerates and thus a choppy and undesirable user experience. Random Studio would like to move as many calculations as possible, required to do cloth simulations, over the GPU as a possible method to improve the framerate. If successful the particle count can be increased by a large amount resulting in a smoother user experience and a more detailed cloth. The desired framerate is 60 frames per second on average modern (mobile) devices in modern browsers. By reducing the performance impact that the cloth has on the overall process developers and designers will be able to put the performance savings towards visually optimizing the scene to create a more aesthetically pleasurable user experience.

2. Theoretical framework & literature review

In the following chapter I'll discuss the preliminary research that has been done to gain a better insight into the preliminary problem as stated in the previous chapter. The theoretical research is split up into three parts:

In chapter **"2.1 Cloth simulation"** I'll introduce what cloth is, how a cloth simulation is constructed mathematically, what forces play a role in a cloth simulation and how they are applied.

"2.2 Making use of the WebGL rendering pipeline to bridge the CPU and GPU" describes the components of the WebGL rendering pipeline necessary to do computations on the GPU that require having multiple states of the program available (a requirement for virtually any particle-like simulation).

In chapter **"2.3 Implementing cloth simulation on the GPU"** I'll touch on the subject of framebuffer objects and how they play a role in when doing computations on the GPU.

After the preliminary research the research questions are redefined in detail and the scope of the research is declared.

I assume the reader is familiar with the difference between the CPU and GPU and what problems each one is optimized for. If the reader feels the need to brush up on the topic I've included a small introduction in chapter 10.1. I assume the reader has at least the level of high school mathematics and physics and general programming experience in C-like languages and JavaScript.

2.1 Cloth simulation

Cloth

Cloth is a flexible material that is constructed out of a network of fibres. A bundle of fibers is referred to as thread or yarn. Bundled fibers are interlocked using various techniques such as weaving, knitting and crocheting. The product of these interlocked bundles is referred to as a textile or cloth. Cloth has three main mechanical properties: stretching, shearing and bending. Stretching is the displacement along horizontal and vertical axis. Shearing is the displacement along two diagonal directions. Bending is the curvature of the surface of the cloth or displacement along the depth axis. *(Yalçın, M. Adil & Yıldız, C., 2009)*.

When one observes a collection of various textiles one can note that the characteristics of these textiles are very different from each other. Some are dense, others loosely woven, elastic or knitted. To cover all these different implicit cloth characteristics is beyond the scope of this paper and instead I would like to focus on a single cloth-like material with a certain set of properties.

In this research report I'll only consider the characteristics of a silk scarf, specifically a high-quality, heavy silk (65 grams), densely woven Hermès scarf as shown in *Figure 2.1.1*. A silk scarf is a flexible thin material, can drape onto objects and internal folds are formed easily.

Figure 2.1.1: Example of a silk scarf by Hermès. (Hermès, 2016).

In computer graphics cloth simulations are often simulated by replicating the fibers of cloth using particles and spring forces. A particle is a single node that has a certain position and velocity. A simulated cloth consists out of a particle grid and thus multiple nodes. These nodes are connected to each other using springs and embed the mechanical properties of cloth. A fully connected particle system by a spring ruleset is referred to as a mass spring system.

Particles

Contrary to static geometric objects, particles in a cloth simulation are moving points of mass m in two- or three dimensional space. A single particle holds information about its position r and thus velocity v and acceleration a when calculating the change over time expressed as Δt. In a three dimensional Cartesian coordinate system ($[x, y, z]$) one expresses a particle as a vector, a displacement from the origin $[i, j, k]$.

A large collection of particles affected over time is referred to as a particle system. The motion of particles, traditionally, is affected by both internal as well as external forces.

Internal forces are forces exchanged by particles within the system. External forces are any forces caused by an external agent such as wind, gravity or user interaction.

Springs

Springs in the context of computer graphics simulations simulate the ones in the physical world. One can think of a spring as a helical spring that is connected to a fixed object on one side of the spring whilst the other side is being pulled or pushed with a certain magnitude of a force. In *Figure 2.2.2* is shown a visualization of Hooke's law on springs which states the following: the extension of a spring is proportional to the force applied to the spring.

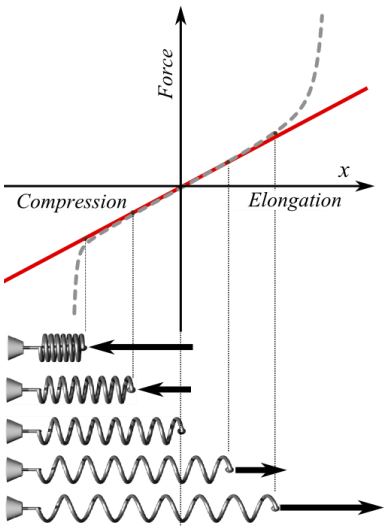


Figure 2.2.2: Visualization of Hooke's law on springs. (Svjo - Wikipedia, 2013).

The equation of Hooke's law on springs is as follows: $F = -k \cdot \Delta l$ where k is the spring constant, the stiffness of the spring, and Δl the displacement of the spring from its original length.

When comparing the knitted cloth of the poloshirt and a silk scarf one can clearly see the difference in k, the stiffness of the material. Stretching the silk scarf an x amount compared to the knitted polo shirt over the same distance will require much more force than the force required to move the polo shirt. Δl is the amount the cloth is stretched compared to the original resting length.

Mass spring system

A mass spring system is a grid of particles connected using springs. Traditional mass spring systems use three types of structural springs that hold the 'fibres' in the simulated cloth together: structural, shearing and bending springs.

When placing the traditional idea of a helical spring connected to a fixed object in the context of a mass spring system one can imagine a particle being the fixed object and the neighbouring particles as the pushing and pulling objects. The force that is expressed between the particles is then referred to as a spring.

Structural springs connect each particle in the grid vertically and horizontally represented by the green springs in *Figure 2.2.3*. Shearing springs will keep the cloth from shearing and connects each particle in the particle grid diagonally as shown in *Figure 2.2.3* by the red springs. The bending springs, represented by the blue springs in *Figure 2.2.3*, connect each group of two particles and keep the cloth from bending. The collection of these springs are also referred to as the constraints of the cloth.

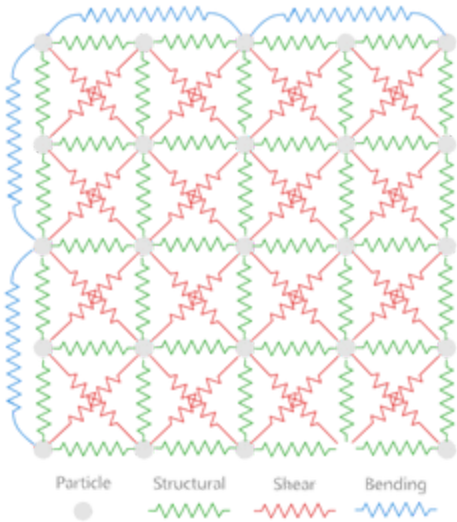


Figure 2.2.3: Diagram showing a traditional mass spring system with structural, shearing and bending springs. (Gorkin, 2009)

Forces

During each step of the simulation, internal and external forces act on the particles in the cloth system. Each new frame the forces are calculated for each particle.

The force applied to each particle is calculated using Newton's law of motion as: $\Sigma F = m * a$ or rewritten, to calculate the acceleration, as $a = \Sigma F/m$ where m is the mass of the particle, a the acceleration and ΣF the sum of the earlier mentioned internal and external forces.

In *Figure 2.2.4* one can see a representation of a single particle vector in a fixed Cartesian coordinate system where O marks the origin and A is the current position one can calculate the $[x, y, z]$ position of.

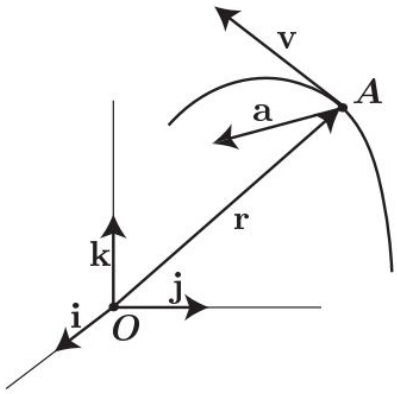


Figure 2.2.4: A diagram showing a particle as a displacement in the Cartesian coordinate system. (Widnall S., 2009).

The position vector of a particle is written as $r(t) = x(t)i + y(t)j + z(t)k$, the displacement from the origin. (*Widnall, S. & Peraire J, 2009)*

The change in position of a particle over time, velocity, can be written as
$v(t) = vx(t)i + vy(t)j + vz(t)k = x^{\cdot}(t)i + y^{\cdot}(t)j + z^{\cdot}(t)k = r^{\cdot}(t)$
The velocity of a single particle is then: $v = \sqrt{v_x^2 + v_y^2 + v_z^2}$

The change in velocity of a particle over time, acceleration, can be written as
$a(t) = ax(t)i + ay(t)j + az(t)k = v^{\cdot}x(t)i + v^{\cdot}y(t)j + v^{\cdot}z(t)k = v^{\cdot}(t)$
and the magnitude of the acceleration is then: $a = \sqrt{a_x^2 + a_y^2 + a_z^2}$

As the mass spring system is Newtonian, gravity is the base force to which all other internal and external forces are applied. For the sake of simplicity, the Earth's gravitational force is expressed as a constant: $G = 9.81\ m/s^2$.

The most important internal force to consider is damping of the internal springs in the cloth. One can think a spring damper as visualized in *Figure 2.2.5*: a second mass damps the oscillation of the first mass. The point of a damping is to smoothen the oscillation of the spring, a necessary requirement for simulating smooth cloth.

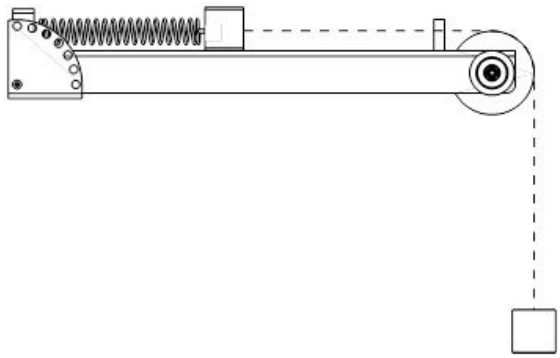


Figure 2.2.5: Diagram of a mechanical spring damper (Quanser, 2016).

The earlier mentioned structural, shearing and bending springs have different damper specifications to satisfy the balance between a too tight and a too loose cloth. As the type of cloth differs so do the damper parameters needed to keep structural integrity intact.

To simulate felt or heavy wool one would need to set the damping much higher than when one would like to simulate smooth silk. It is a process of fine tuning depending on the characteristics of the material one is trying to simulate. As my goal is to mimic a Hermès silk scarf the damping factor on the springs is set quite low. In result this means that external forces such as wind will have much more effect on it.

Numerical integration

As I am implementing a dynamic cloth simulation and not static geometry the aspect of change over time plays a large role. To calculate the position, velocity and acceleration of a particle over time as it is being affected by internal and external forces one uses a method of extrapolation: one extrapolates the position and velocity based on the previous position and velocity. This process of extrapolation has to be integrated numerically using a differential equation to accurately predict the next position and velocity.

There are a range of differential equations one can use, some are simple and inaccurate others highly complex and very accurate. As my goal is to simulate silk-like cloth in the browser, creating a visually appealing experience is of much more importance than the simulation being as physically accurate as possible. Performance over accuracy. It is not a critical system so I feel free to choose the most accurate yet easy to implement differential equation.

Other considerations when choosing what differential equation one should use for his or her specific case is the desired amount of particles and the chosen timestep. A timestep is the desired time between each computation cycle. As one is updating the position of particles in

physical systems rendered in the browser it is preferred to use a timestep that correlates with the target framerate of the application. This is because if the timestep of the physics system update is different than the current framerate, that is - being impacted positively or negatively and thus lower and higher than the target framerate -, the calculations are offset.

Browsers, as of the time of writing, limit the framerate per second and thus one can use a static but very small timestep to compensate for systematic errors.
At initialization the cloth is at its positional origin of $[0,\ 0,\ 0]$ with a velocity $[0,\ 0,\ 0]$ in the Cartesian coordinate system. From here at every simulation timestep a new position and velocity is calculated. A numerical integration method that is oftenly used due to its simplicity is Euler's method. (Rodriguez, J., 2016)
As the position and velocity is known at t_0 of the simulation one can use the old position and velocity to calculate the tangent line to the curve along which the cloth simulation moves. Once the tangent line has been calculated one moves timestep t_{0+n} when n is the timestep, along the tangent line resulting in a new position with the assumption that the existing 'trend' will be applicable in the new situation. At this new position all external and internal forces are recalculated and applied again.

Euler's method

To calculate the new velocity, as it is needed to calculate the new position, using Eulers method one uses the following formula :
$V_{n+1} = V_n + \Delta t * A_n$, where V_{n+1} is the new velocity, V_n the current velocity and A_n the current acceleration (calculated using Newton's second law of motion).
$P_{n+1} = P_n + \Delta t * V_n$, where P_{n+1} is the new position and V_n the current velocity.

In general this method of estimating new positions based on old positions works decently for extremely simple linear calculations though one must consider that the systematic error per step is proportional to the step size squared resulting in unstable and unreliable systems very quickly. The systematic error causes the particle to gain potential energy, as the Euler method is energy preserving, and does not correct this error.

To counter this issue other methods of numerical integration are available. I'll focus on the simple semi-implicit Euler's method as it is accurately feasible enough and has been found to be easy to implement. I consider more advanced and more accurate integration methods such as 4th order Runge-Kutta or time corrected Verlet integrations beyond the scope of this report and I do not find them to be required for my purpose, simulating a cloth for its aesthetic quality and specifically for physical accuracy.

Semi implicit Euler's method

The semi-implicit Euler's method corrects the systematic energy gaining error by instead of using the current velocity V_n in $P_{n+1} = P_n + \Delta t * V_n$ it uses V_{n+1}, the new velocity. The advantage of using the new velocity over the current velocity is that it adds much less

potential energy to the total energy than Euler's method making the system much more stable and accurate. (Rodriguez, J., 2016).

2.2 The WebGL rendering pipeline

OpenGL (Open Graphics Library) is a cross-language, cross-platform API (application programming interface) for rendering 2D and 3D graphics. The API is used to interact with the hardware level with the GPU (graphics processing unit) to enable hardware acceleration. OpenGL is a desktop computer centric API. OpenGL ES 2.0 builds upon OpenGL and has for the most part the same features but is largely optimized for embedded systems like smartphones and tablets.

WebGL (Web Graphics Library) is a JavaScript API that is build upon OpenGL ES 2.0. WebGL allows us to render interactive 3D and 2D graphics within all modern browsers without the use of external and/or proprietary plug-ins. WebGL is based on the specification of OpenGL ES 2.0 and for the most part retains the semantics of OpenGL ES in order to guarantee support for embedded systems. WebGL uses the HTML5 `<canvas>` element to display the rendered content.

Graphics can be rendered at high efficiency and volume by passing data from the CPU to the GPU using the so-called programmable pipeline available in OpenGL, OpenGL ES 2.0 and WebGL. All of them are quite similar though WebGL is more limited due to browser security sandboxing. The WebGL rendering pipeline has several stages that rely on each other, as shown in *Figure 2.2.1*.

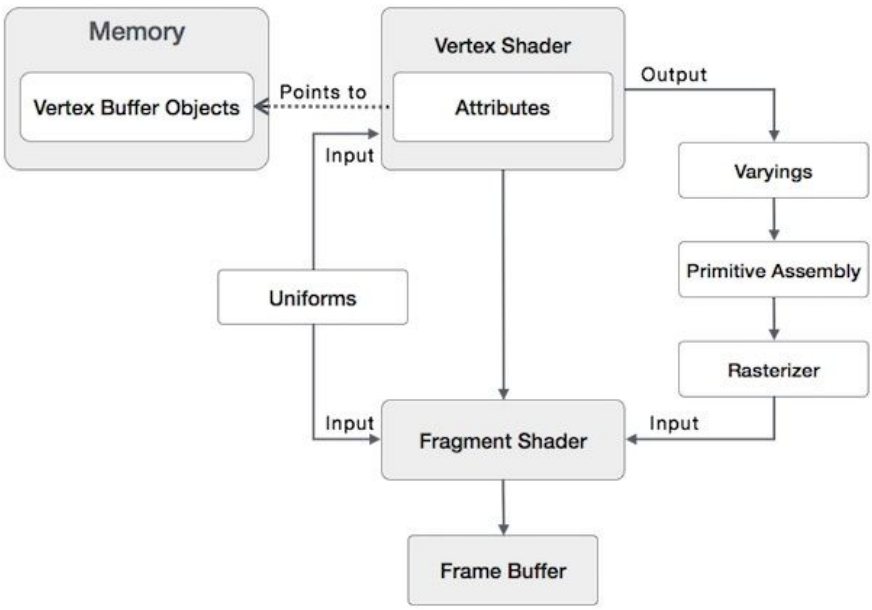


Figure 2.2.1: The programmable WebGL pipeline (Mohtashim, M, 2016)

Staging and storing positional data

One can think of position vectors as a displacement from the origin *[0, 0, 0]* in a Cartesian *[x, y, z]* coordinate system. In other words: a position vector is a point in the 3D space that has been displaced from the origin. A vertex is a position vector that holds other attributes too such as color and texture coordinates. The position of a floating point is stored in 4 homogeneous coordinates: *[x, y, z, w]* in a 4-dimensional vector (*vec4*). In most cases *w* will be a constant 1.0 and refers to a perspective scalar property. In WebGL all geometry is constructed out of triangles. Quadrilaterals are not available. This vertex data is stored in vertex buffer objects as shown in *Figure 2.2.1*.

Vertex shader

Once one has constructed a single vertex, a single particle with properties in the mass spring system, and stored it inside of a VBO the vertex is available to be passed onto the vertex shader. The vertex shader is a shader that is executed for each vertex. This allows for parallel computation as the calculations applied to a single vertex are not dependent on the calculations applied to another vertex. Because of this the GPU does not have knowledge about the location of each vertex relative to each other.

In the primitive assembly stage these triangles are assembled. Important to note is that at no point in time the GPU will treat complex geometry such as a cube as an actual cube. To the GPU complex geometry are just a large collection of vertices. Transformations like rotation and scale are done through matrix calculations.

This result is passed on to the rasterizer. In the rasterization stage the triangles are transformed to a potential pixel location on the screen. The rasterizer will determine if a triangle is in the viewport and if the triangle is visible. If this is not the case the triangle will be discarded and will not be passed on further down the pipeline. The potential pixel location is then made available as fragments to the fragment shader.

Fragment shader

On each of the fragments or “potential pixels”, passed on from the rasterizer, the fragment shader is executed. Similar to the vertex shader this allows for parallel computation as the calculation of the color of a single fragment is not dependent on the calculation of another fragment.

In the fragment shader one calculates the color of individual pixels. In the shader one is able to apply per-fragment lighting, shadows, specular and texture mapping, etc. Interestingly, after the rasterization phase the GPU does have knowledge about the screen coordinates to which the pixel is drawn. This means that, because one knows the screen coordinates, one knows all neighboring pixels relative to the pixel. This is very helpful because now one can apply for example traditional blur shading or other post-processing techniques to the pixels as it requires the knowledge of pixel colors adjacent to the target pixel to calculate the result.

The resulting colored pixel information is passed to the frame buffer and in turn rendered to the screen.

Communication

Communication between application, website in our case, happens through the HTML5 `<canvas>` element to what the WebGL API instantiation attaches. You can embed shaders in the HTML by wrapping it using `<script>` tags with an unused identifier (so that the script can be parsed as plain text) and adding an `<id>` tag that one can reference in the script.

2.3 Implementing cloth simulation on the GPU

To implement the cloth simulation on the GPU in WebGL one uses framebuffer objects or in short: FBO's. The stored and staged data (containing the position of a floating point in 4 homogeneous coordinates: *[x, y, z, w]* using the *vec4* datatype) is rendered as a texture in a *[r, g, b, a]* format.

For those unfamiliar, images in a PNG format can be expressed as in a *[r, g, b, a]* format where *r* is the red channel, *g* the green and *b* the blue, *a* the alpha. By 'misusing' this format one is able to write the staged positional and velocity data to a texture that one can express as an image. Using framebuffer objects one is able to render to an off-screen textures meaning that the presence of the texture does not have any influence on the composition but is still available to be used.
One can use these textures stored in the buffer to apply post processing effects such as motion blur. The point of storing these textures in buffers is to be able to reuse the data in further calculations.

In cloth simulations framebuffer objects are used to store the position and velocity in off-screen data textures in order to keep all calculations on the GPU. Every render pass the reference to the off-screen data textures is switched around: the old position is overwritten by the current position and the current position is overwritten by the new position. This process is referred to as ping-ponging textures.

2.4 Conclusion of theory

The preliminary research has introduced the topics necessary to, in theory, simulate cloth in WebGL. The formulas necessary to compute the internal spring forces and the external forces has been researched.

The process that follows can be divided into three distinctive parts. First of all a further study is necessary to test current implementations of WebGL simulations running on the GPU for their compatibility and research what techniques they commonly use to compute. The

second part will research the development process of simulating cloth in WebGL based on the earlier results. In the third part I'll test my implementation of the cloth simulation for its performance and compatibility with various settings. The desired framerate is 60 frames per second on average modern (mobile) devices in modern browsers at significantly higher particle counts. The lowest framerate Random Studio deems to be acceptable is 30 frames per second.

3. Problem definition

3.0 Definition of the problem

The problem definition can be defined as follows:

Random Studio wants to realistically simulate a Hermès silk scarf using a Newtonian mass-spring system at 60 frames per second utilizing the GPU through WebGL in modern web browsers running on modern (mobile) devices.

The problem consists out of three distinctive parts as follows:

The first part of the problem consists out of the question how one is able to effectively move computations traditionally done on the CPU over to the GPU and how one has to reformulate the computational problem to allow for GPU computation. The prototype should effectively do calculations on the GPU without creating a communication bottleneck.

The second part of the problem is finding what development steps are necessary to create a cloth simulation using framebuffer objects and how one adds internal and external forces. The prototype should be Newtonian mass spring system consisting out of particles, with a certain mass, connected via structural, shearing and bending springs with dampers. The prototype should be receptive to wind and be affected by gravity. It should have, at a minimum, a texture, realistic cloth physics and basic lighting.

The third part of the problem is researching the compatibility across (mobile) devices and browsers and the frame rate at different particle counts. The prototype should be available on all modern (mobile) devices and perform near the target framerate of 60 frames per second. The lowest framerate Random Studio deems to be acceptable is 30 frames per second given that the cloth simulation uses a higher particle count (over 400 in a 20 x 20 grid) than current implementations.

3.1 Main and sub questions

Main research question

The main research question is as follows:

How does one realistically simulate a Hermès silk scarf using a Newtonian mass-spring system at 60 frames per second utilizing the GPU through WebGL in modern web browsers running on modern (mobile) devices?

The main question is divided into three questions with each multiple sub-questions as follows:

Sub question one

How does one effectively move calculations from the CPU over to the GPU?

- What are commonly used techniques to do computation on the GPU in WebGL?
- How do these techniques allow the CPU and GPU communicate?

Sub question two

What is the development process of simulating cloth on GPU using FBO's?

- How does one construct a mass spring system using using particles and springs?
- How does one add internal forces such as spring constraints?
- How does one add external forces such as wind and gravity?
- How does one fixate the cloth on certain points?

Sub question three

What are the limitations and bottlenecks of my cloth simulation in WebGL?

- How compatible is my implementation of a cloth simulation using WebGL with modern browsers and modern (mobile) devices?
- What are the device specific hardware limitations regarding WebGL and what are possible solutions?

3.2 Scope

It must be noted that due to time constraints I will not be able to program a comparable cloth simulation on the CPU and will thus not be able to compare the performance gain of the WebGL implementation on the GPU compared to the traditional implementation on the CPU.

Random Studio much more desires to have a working prototype that utilizes the parallel computational power of the GPU rather than having two unfinished prototypes as a result.

Therefore I will, even though I realise it will harm the integrity of the research by not offering a comparable method, not be implementing a cloth simulation utilizing the CPU. Instead I will be testing the compatibility of the GPU across a range of modern devices and browsers. I am quite confident that the difference in performance at higher particle rates will be empirically verifiable. The assumption is that doing the computations necessary for simulating cloth (if implemented correctly) on the CPU is less efficient than on the GPU.

1. **Hardware:** I will only focus on supporting recent and popular devices such as the iPhone 5S (2013), iPad Mini 2 (2013) and an iPad Pro (2016). I have the ability to test using the Samsung Edge S7 (2016), Macbook Pro 13-inch Retina (2015), and a Lenovo W530. The main goal is cross-browser and cross-device support along this range of devices.

2. **Browsers:** I will only test on the latest versions of modern browsers on desktop (Chrome, Firefox, Safari and Edge) for compatibility as all of them offer WebGL support. I will also only test the latest versions of modern browsers (Chrome, Safari) on mobile devices and tablets.

3. **WebGL:** Due to limitations in time and complexity I won't be able to develop the prototype using the full potential of WebGL. Important to note are the limitations of WebGL compared to OpenGL and native heavily optimized API's such as Metal for Apple's iOS and OSx or CUDA optimized for NVIDIA graphics cards. It is not possible to get the same performance in the browser as on natively built alternatives.

   The current implementation of WebGL across browsers will never be able to run at the same speeds as native applications as WebGL has been build to support a wide range of device types and browsers. Because WebGL is based on the specification of OpenGL ES 2.0 one can not expect the same functionality and optimization compared to working with the latest native version of OpenGL 4.5.

   A new experimental version of WebGL called WebGL2 is based on the specification of OpenGL ES 3.0 and aims to remove many of the current browser specific sandboxing restrictions and adds extra capabilities like transform feedback, an alternate approach to framebuffer objects as a method to store data on the GPU. At the time of writing none of the browsers are supporting WebGL2 out of the box and are in development stage at both Chrome and Firefox, only available by enabling experimental flags in the browsers.

   Random Studio does not develop websites for nightly builds or other bleeding edge builds and, as any sensible business, must take cross-browser and cross-device support into account when developing.

4. Method of graduation

4.1 Sub question one

I'll be using the method of desk research and experimentation to find answers to sub question one and its sub questions. I'll be first of all conducting desk research in the form of an in-depth study about computing cloth calculations on the GPU and the fundamentals of the graphics pipeline of the WebGL API.

I'll be using the method of experimentation to explore different implementations and commonly used techniques of running complex calculations solely on the GPU.

As the first experiment I'll be testing all devices and browsers in scope for their reported WebGL support using the website WebGLReport (http://webglreport.com/). I'll document the found results in Google Sheets in order to share the tests with ease.

In the next experiment I'll find and dissect the source code of various examples on the Three.js example page known to run computations on the GPU. The technique used is often referred to as GPGPU and is used to create large complex particle systems and water simulations. I'll test the examples and document the results for their mobile and cross browser support.

I'll examine the different implementations based on the compatibility and complexity of the implementation technique used and measure the performance savings between different implementations.

The product of sub question one will be several sheets of results from the earlier mentioned experiments documenting the support for WebGL features on all devices and browsers within scope and a concrete set of examples to draw inspiration when I start programming my implementation of GPGPU cloth simulation.

4.2 Sub question two

I'll be using the method of desk research and experimentation to find answers to sub question two and its sub questions. First of all I'll be doing desk research to form an in-depth study about the development steps of cloth simulation in the browser using WebGL. Based on the result of this research I'll conduct multiple experiments and create multiple smaller iterative prototypes to construct the simulation.

During the method of desk research information is collected from various sources ranging from WebGL documentation from the Khronos.org foundation, the foundation behind WebGL, to relevant technical talks from SIGGRAPH, an annual conference on computer

graphics, OpenGL implementations of cloth simulations and technical papers on cloth simulation techniques. I'll also be referencing lectures from Carnegie Mellon University on the topic of computer graphics and parallel computing. As the publications of the earlier mentioned foundations, universities and papers are written by competent people with backgrounds in computer graphics I do not have my doubts about their credibility.

During the method of experimental research I'll build several iterations to build towards a cloth simulation in WebGL:

- The first step of the prototype will focus on implementing the Three.js library and setting up a basic scene.
- The second step of the prototype will focus on implementing the basic features of a cloth simulation such as the internal spring constraints using FBO's.
- The third and final step of the prototype will focus on adding external forces such as wind, gravity, textures, a wireframe option and internal forces such as fixation and pinning, adding mass to the particles and improving the visual quality of the scene.

The resulting prototype will serve as the final product accompanying this report. It is important to note that the final product does not consist out of the range of iterative prototypes as the git commit history is open to all and thus one is able to trace back changes in this manner.

The following toolkit will be used as a prototyping platform:

- Macbook Pro 13-inch with retina screen (2015)
- Chrome 53.0, developer tools and the WebGL inspector addon
- Sublime Text
- Npm and Gulp to compile Sass
- Three.js / WebGL API to interact with WebGL
- GLSL to write custom shaders

4.3 Sub question three

I'll be using the method of desk research and experimentation to find answers to sub question three and its sub questions. First of all I'll be doing desk research to form an in-depth study about benchmarking WebGL implementations and conduct research on the topic of WebGL debugging tools. I'll test how compatible my implementation of a cloth simulation using WebGL in modern browsers and modern (mobile) devices is and will point out bottlenecks and areas for possible improvements.

Secondly I'll create a list of existing tools to help during WebGL development and tools for testing WebGL extension support in various browsers on various devices to be shared internally.

5. Results

5.1 Sub question one results

Initial experiments

After the initial desk research I started looking for examples of WebGL particle systems that run solely on the GPU. I found several quite promising examples (water- and particle simulations) that use framebuffer objects, in order to preserve state and allow for computation on the GPU, online and tested them for browser and device compatibility.

Attached one can find the folder *"Simulating cloth in WebGL - sheets"* wherein one can find *"WebGL FBO support across devices and browsers - Initial tests"* documenting the entire process and results. In chapter 10.2 and 10.3 one can find the raw documentation from the experimentation process that was necessary to answer sub question one. It shows my thinking process, considerations and drawbacks.

As seen in the spreadsheet; none of the mobile devices seemed to support the framebuffer object technique that is often used to compute on the GPU. The two desktop devices, the Lenovo W530 and the Macbook Pro Retina 13-inch (2015) had no issues running the issues and offer cross-browser compatibility. The Samsung Galaxy S7 Edge (2016) reported to not support `OES_texture_float` and thus unable to use the framebuffer object technique.

Even though the iOS devices report to have `OES_texture_float` there is no indication of it actually being able to. Curious as to why no mobile devices seemed to be working I started researching the debugging errors from the iPad Pro, that I was able to read through the developer interface of the Safari browser and by doing so I was able to trace back the error as documented in the earlier mentioned spreadsheet.

All of the examples use the earlier mentioned framebuffer object, also referred to as render-to-texture, technique to be preserve state and thus be able to do computation on the GPU.

Continued experiments

As a response to the earlier experiments I started looking for examples of FBO's known to work with mobile devices. One example that works on all tested iOS devices is a user contributed example featured in the Three.js library examples: *'GPGPU Birds Flocking*' by Juan Espinosa (Yomboprime) and Joshua Koo (ZZ85). Attached one can find the folder *"Simulating cloth in WebGL - sheets"* wherein one can find *"WebGL FBO support across devices and browsers - Continued tests"* documenting the entire process and results.

After the initial testing and browsing through the relevant commits in the Three.js repository on Github I spotted the line that made it work as shown in *Figure 5.1.1.*

```
type: (/(iPad|iPhone|iPod)/g.test(navigator.userAgent)) ? THREE.HalfFloatType :
THREE.FloatType;
```

Figure 5.1.1: Use either half floats or full floating point textures based on device user agents

Even though the iOS devices report to support full floating point textures it seems that they don't, at least not in the same way as the tested desktop devices do. Using half floating point numbers allows FBO's to work on the tested iOS devices.

A major takeaway from both experiments is that the reported support for WebGL extensions by the browser or hardware sometimes does not reflect the actual support. The necessary requirement to use half floating point numbers is `EXT_color_buffer_half_float` is not reported as being supported even though it clearly does. Both desktop devices fail to report their support for `EXT_color_buffer_float` even though both do. Testing on physical devices is therefore highly recommended and can not be replaced by referencing to the reported support during development.

The initial research led to a series of steps one should consider when implementing a simulation using framebuffer objects in WebGL. The development process of creating ping-ponging framebuffer objects is as follows:

Initialization:

- Create a typed Float32Array buffer with the size (width * height * 4) to hold all points of the texture in a 4 dimensional floating point vector (R, G, B, A).
- Add the floating point data to the typed Float32Array.
- Write the typed Float32Array data to a data texture (easily created using THREE.DataTexture) to directly create a texture from the raw data.
- Add the data texture as a texture to a plane geometry.
- Aim an orthographic camera at the textured plane.
- Set up a render target (easily created using THREE.WebGLRenderTarget) to create a buffer to render the data texture off-screen.
- Render the data texture in the off-screen buffer.
- Clone the off-screen buffer.
- Initialize the computation (extrapolate the first time by using the same texture twice or initialize at the origin).

Updating:

- Compute the thing that needs a preserved state to extrapolate.
- Ping-pong the off-screen texture by swapping the reference pointer at the end of every render cycle allowing to store the current data to be used in the next render cycle. The current data becomes the previous data and the data in the next render cycle will become the current data.

Attached one can find the folder *"Simulating cloth in WebGL - sheets"* wherein one can find *"WebGL FBO support across devices and browsers - Overview"* providing an overview of the tests and results related to answering sub question one.

5.2 Sub question two results

To pass around data textures effectively I use a short helper function named GPGPU.js. (Jones, B., 2015) that helps with instantiating and ping-ponging multiple FBO's in an elegant manner by reusing the same geometry plane, camera and scene.

Development steps

Attached one can find the folder *"Simulating cloth in WebGL - source code"* wherein one can find the commented source code of the prototype. In *"README.md"* one can find instructions to view the prototype locally. The development process of creating a cloth simulation using FBO's in WebGL is as follows:

Initialization:

- Create a typed Float32Array buffer with the size (clothWidth * clothHeight * 4) to hold all points of the texture in a 4 dimensional floating point vector (R, G, B, A).
- Write the created floating point positional data to the typed Float32Array.
- Write the typed Float32Array data to a data texture (easily created using THREE.DataTexture) to directly create a texture from the raw data.
  (*Main.js: line 87 - 100).*

- Create a new instance of the GPGPU.js helper function and pass the data texture containing the positional data to it.
- Read the initial position from the initial data passed from the Float32Array.
- Clone and instantiate the empty velocity shader to an empty previous velocity texture with the correct dimensions.
  (*Main.js: line 102 - 104).*

- Construct the cloth mesh, apply a texture and add it to the scene.
  (*Main.js: line 123 - 175).*

- Initialize the positions and velocity based on the data texture.
  (*Main.js: line 186 - 231).*

Updating:

- Velocity texture update: write the current velocity data texture to the previous velocity data texture, compute the mass spring constraints, the new velocity using the first part of the semi-implicit Euler's method and update the current velocity data texture.
  (*SimulationShader.js: line 23 - 210)*
- Position texture update: update the positional data texture based on the new velocity using the second part of the semi-implicit Euler's method and send the new positional texture to the particle shader.
  (*SimulationShader.js: line 212 - 269)*

- Calculate the normals based on new positional data.
  (*Index.html: line 53 - 91)*

- Apply the Hermès scarf texture and basic Phong lighting based on the new normals.
  (*Index.html: line 102 - 135)*

- Swap the reference pointer to the velocity FBO.
  (*GPGPU.js: line 45 - 47)*

- Swap the reference pointer to the position FBO.
  (*Main.js: line 217 - 219)*

At the end of each render cycle `RequestAnimationFrame(render)` is called to render the next frame. This speed of this render cycle depends on the frame limiter in browsers. Google Chrome on desktop has a limit of 60 frames per second so at least calls the render cycle 60 times per second.

Mass spring system, external forces and internal forces

To create a mass spring system one first of all creates the Float32Array buffer and instantiate it. The base external force of gravity is applied first followed by other base forces such as the simplified wind (using sine/cosine over time) applied to each particle mass equally.

The sum of the external forces on each particle is then taken as a base force to which the internal forces are applied. Spring forces are constructed by calculating the forces between the particle mass and its 12 surrounding neighbors as visualized in *Figure 5.2.1*. The structure, as shown in *Figure 5.2.1,* is made up out of the totality of the structural (green), shearing (red) and bending (blue) springs connected to the particle mass.

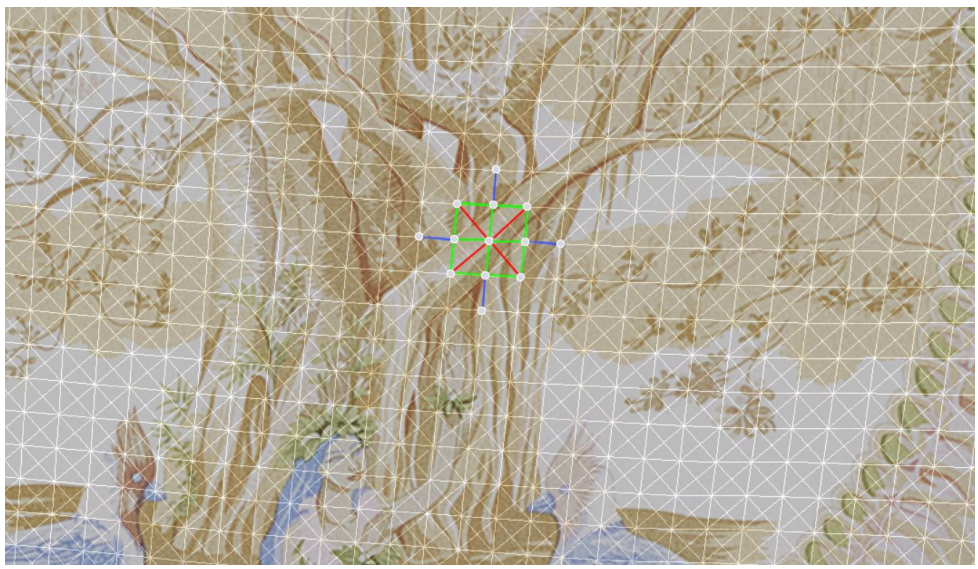


Figure 5.2.1: Diagram of the 12 neighbours of each particle mass

Fixating cloth turns out to be easy, one just sets a specific area in the velocity vector to 0.0 at every frame and exclude the point from being updated whilst the area is marked as a pin. As soon as the pin is released the area becomes susceptible to positional updates based on the velocity.

5.3 Sub question three results

Collected data from the performance and compatibility tests can be found in chapter 10.2, 10.3, 10.4 of the appendix.

Frame rate and compatibility tests

The current implementation of the cloth simulation, as described in this paper, has been tested on 6 different devices:

- iPad Pro (2016)
- iPad Mini 2 (2013)
- iPhone 5S (2013)
- Samsung Galaxy S7 Edge (2016)
- Macbook Pro Retina 13-inch (2015)
- Lenovo W530

The specifications of these devices can be found in the earlier mentioned sheet and the The results are visualized in *Figure 5.3.1*. I've tested 5 different cloth sizes and measured the performance impact. The size of the cloth is expressed in the form of the density of the particle grid that constructs the cloth.

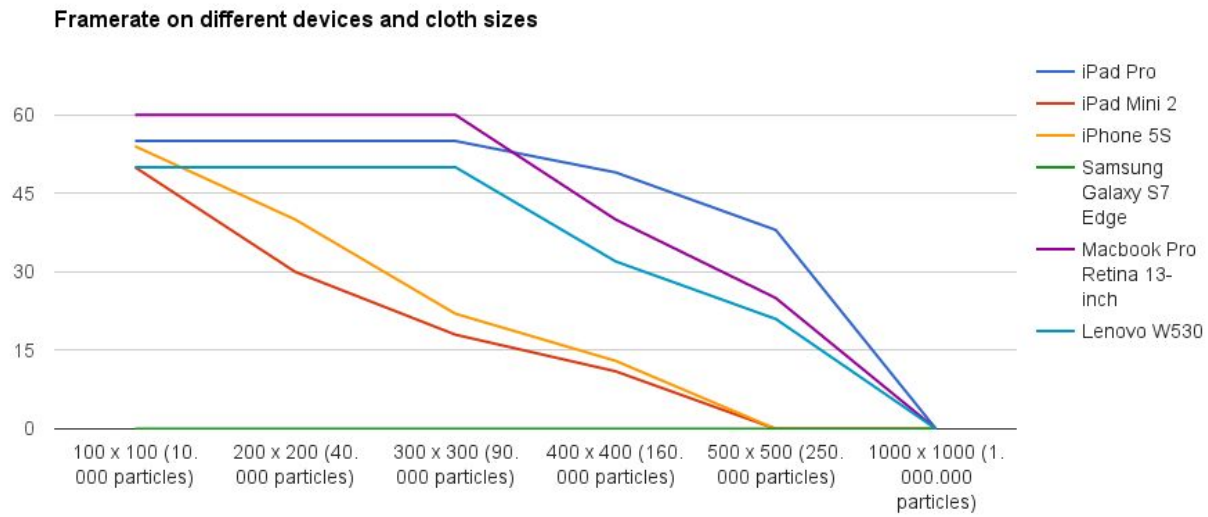


Figure 5.3.1: Visualized results from cloth performance tests

Important to note when analyzing the graph is that browsers limit the framerate to a device specific implementation. Chrome on Macbook Pro's runs smoothly at 60 frames per second whilst the Chrome implementation on the Lenovo W530 runs at a steady 50 frames per second. Framerates on both the iPad Pro and the iPhone 5S are limited to 55 frames per second. The Samsung Galaxy S7 Edge, as mentioned before, does not support off-screen rendering-to-texture in its current implementation.

If one were to remove the frame rate limiter in the browser, as one is sometimes able to force through developer settings or due to bugs (as shown in *Figure 5.3.2)*, one is able to see that the framerate is actually much higher on specific devices than reported as its limit.

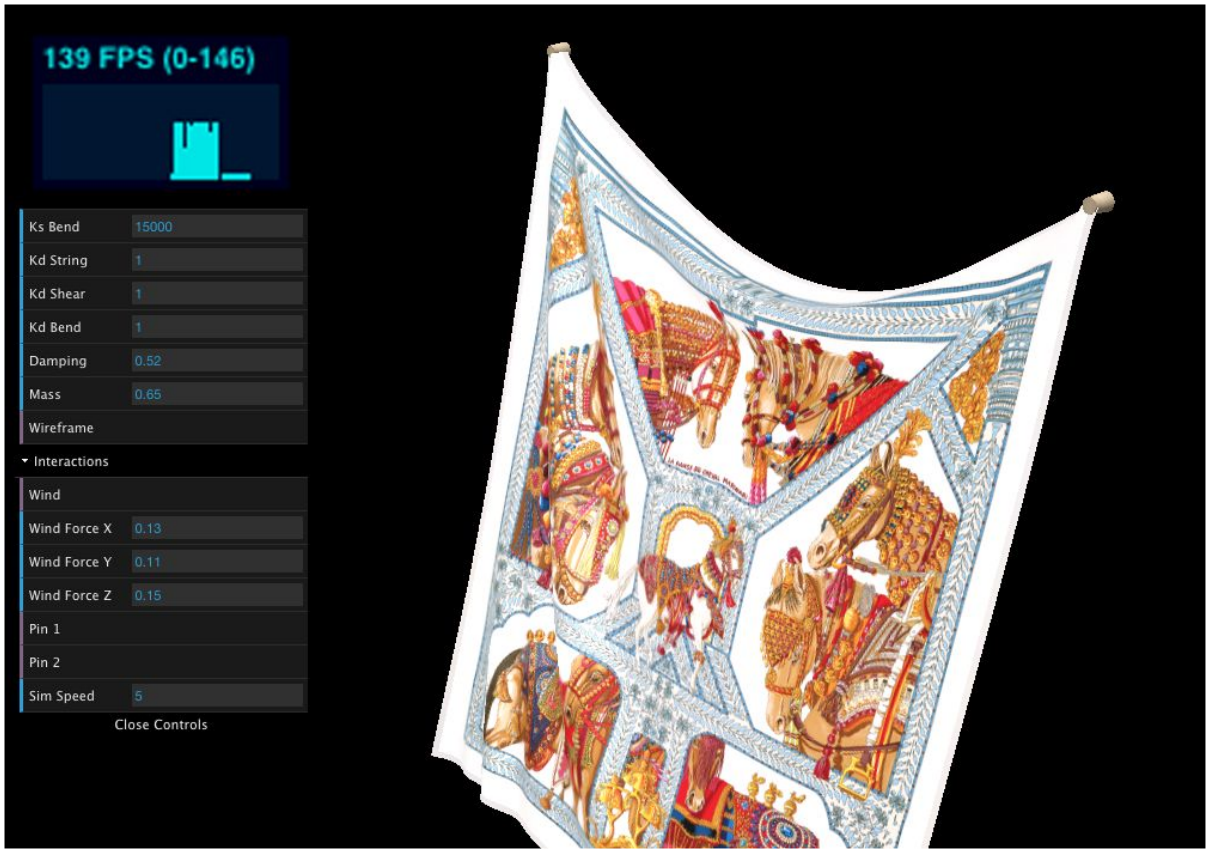


Figure 5.3.2: Delimited framerate due to external screen bug in Chrome Version 53.0.2785.116 (64-bit) - Macbook Pro shows the simulation running at 139 frames per second with a 100 x 100 particle cloth.

On all tested devices a data texture of 1 million particles crashes WebGL. I have not investigated as to why due to the frame rate being unfeasible under 30 frames per second anyways.

Indicating possible bottlenecks

As a method of analyzing I used both the developer tools available in Google Chrome and the open source and free GPL WebGL inspector extension.

When analyzing the cloth simulation in Google Chrome's developer tools using the timeline functionality one can analyze all steps the browser goes through when a page is loaded. The tool is used often by developers at Random Studio as a method of tracking down issues that could be optimized.

Figure 10.4.4 shows the initialization phase in the Google Developers timeline. During the initialization phase the HTML is parsed, scanned for any dependencies including stylesheets and scripts. After the dependencies are loaded they are parsed. The largest script that is parsed is Three.js taking 938.61 ms, a considerable time. A small bottleneck is the GUI and FPS counter that force layout changes and the recalculation of stylesheets on initialization. I would not consider this to be an issue as production websites will never have a GUI or FPS counter, it is used in development.

Figure 10.4.5 shows the texture loading phase. Image decode is render blocking meaning that if one loads a large image all rendering is blocked until the image is loaded. When one loads the site the simulation runs as soon as it is initialized whilst the image is loaded after. One can argue that it would be favorable to load the image first before starting the simulation. Once the updating cycle has started running the GPU phase runs smoothly as shown in *Figure 10.4.6 and Figure 10.4.7*.

WebGL Inspector is an addon one can use to analyze the a large amount of the communication between the graphics card and the browser. The tool allows the user to see when certain calls are made, especially useful during development and the optimization phase as shown in *Figure 10.4.8*. A useful tool within the inspector is 'trace' tab, allowing the user to see all redundant calls made in a captured frame.

From the resulting research the following list of possible ways to improve the performance and loading times was established.

Possible ways of improving performance and loading times:

- Use the minified version of Three.js

It is obvious one would use the minified version in production to reduce loading and parsing time. Combining and minifying JavaScript scripts using Require.js and loading them asynchronously would highly improves loading times on larger websites. As this is just a prototype meant to be used in development I prefer using the non-minified version as debugging is much easier.

- Create a light version of Three.js, the current full build is almost 1.1Mb of JavaScript

A light version would remove unnecessary functionality depending on the project. In this case one might be interested in removing model loaders, unused shaders and materials, plugins, animation, unused geometries and the canvas renderer from the final script.

- Load the GUI in a different way to avoid forced reflows and recalculation of styles

By iteratively adding new GUI items to a folder during initialization forces layout changes and could be avoided by bundling all before rendering.

- Optimize the file size of the textures, the larger the file size the longer it takes to initially load and decode.

File size of textures is always a matter of weighing off visual quality and file size. I decided, as the product is the central piece visual quality is of the highest priority and thus decided to use textures of 2048 by 2048 pixels and a relatively low compressed JPEG. Depending on the camera perspective and how close it can get to an object one considers the quality of the image. For showing off folds in the cloth the camera will be very close to the cloth making permits, in my opinion, the use of high resolution textures.

- Use base 2 square textures (512 x 512, 1024 x 1024, 2048 x 2048) where possible in order to avoid having to resize.

Currently the texture used for the pins is not a base 2 square and is resized by THREE.WebGLRenderer. Resizing this beforehand reduces at least one step during the initialization of the texture rendering.

- Eliminate the need to decode the image by forcing hardware acceleration.

In CSS3 the property `transform: translate3d(0,0,0);` is a well known hack to force hardware acceleration (the use of the GPU in this case). `THREE.TextureLoader()` appears to not be using hardware acceleration to load textures and is creating a major render block for around 140ms. A possible reason as to why it does not utilize the GPU is that the function is used for several renderers such as the canvas renderer. A possible fix would be to contribute an alternative hardware accelerated texture loader to the Three.js project.

- Compile, minify all scripts and load the JavaScript asynchronously

I have decided, due to the simulation being a prototype, to not use Require.js as it adds more complexity than functionality it adds in such a small prototype. In the end it would still be possible to be render blocking due to it being a requirement for initializing the GPU rendering. That said, in production one would always do this regardless.

Attached one can find the folder *“Simulating cloth in WebGL - sheets”* wherein one can find *“Cloth simulation compatibility test”* providing an overview of results from the compatibility tests.

6. Conclusion and discussion

6.1 Sub question one conclusion

To effectively move calculations from the CPU over to the GPU I would recommend the developer to use framebuffer objects as a method of ‘storing’ data textures on the GPU that one can then use to extrapolate. I recommend the developer to use the GPGPU.js helper library to support him or her when working with data textures. The development steps that one should go through are the following:

To initialize the render cycle one should at least go through:

- Create a typed Float32Array buffer with the size (width * height * 4) to hold all points of the texture in a 4 dimensional floating point vector (R, G, B, A).
- Add the floating point data to the typed Float32Array.
- Write the typed Float32Array data to a data texture (*easily created using THREE.DataTexture*) to directly create a texture from the raw data.
- Add the data texture as a texture to a plane geometry.
- Aim an orthographic camera at the textured plane.
- Set up a render target (*easily created using THREE.WebGLRenderTarget*) to create a buffer to render the data texture off-screen.
- Render the data texture in the off-screen buffer.
- Clone the off-screen buffer.
- Initialize the computation (extrapolate the first time by using the same texture twice or initialize at the origin).

After the first initialization render cycle is finished one should start the updating cycle consisting out of:

- Compute the thing that needs previous data to extrapolate.
- Ping-pong the off-screen texture by swapping the reference pointer at the end of every render cycle allowing to store the current data to be used in the next render cycle. The current data becomes the previous data and the data in the next render cycle will become the current data.

By using this general approach one is able to move the computation of various simulations (think of water, smoke, particle, fluid) that use extrapolation from the CPU over to the GPU.

6.2 Sub question two conclusion

Similar to the general approach of using framebuffer objects in WebGL I recommend using the GPGPU.js helper library in order to simplify the process of instantiating framebuffer object buffers to store the data textures in.

Development steps

The development process of creating a cloth simulation using FBO's in WebGL is as follows:

Initialization:

- Create a typed Float32Array buffer with the size (clothWidth * clothHeight * 4) to hold all points of the texture in a 4 dimensional floating point vector (R, G, B, A).
- Write the created floating point positional data to the typed Float32Array.
- Write the typed Float32Array data to a data texture (easily created using THREE.DataTexture) to directly create a texture from the raw data.
- Create a new instance of the GPGPU.js helper function and pass the data texture containing the positional data to it.
- Read the initial position from the initial data passed from the Float32Array.
- Clone and instantiate the empty velocity shader to an empty previous velocity texture with the correct dimensions.

Updating:

- Velocity texture update: write the current velocity data texture to the previous velocity data texture, compute the mass spring constraints, the new velocity using the first part of the semi-implicit Euler's method and update the current velocity data texture.
- Position texture update: update the positional data texture based on the new velocity using the second part of the semi-implicit Euler's method.
- Send the new positional texture to the particle shader.
- Calculate the normals based on new positional data.
- Apply the Hermès scarf texture and basic Phong lighting based on the new normals.
- Swap the reference pointer to the velocity FBO.
- Swap the reference pointer to the position FBO.

At the end of each render cycle `RequestAnimationFrame(render)`, a method that to run the render function before the next repaint.

To create a mass spring system one first of all creates the Float32Array buffer and instantiate it. The base external force of gravity is applied first followed by other base forces such as the simplified wind (using sine/cosine over time) applied to each particle mass equally. The sum of the external forces on each particle is then taken as a base force to which the internal forces are applied. Spring forces are constructed by calculating the forces between the particle mass and its 12 surrounding neighbors.

To fixate a cloth one sets a specific area in the velocity vector to 0.0 at every frame and exclude the point from being updated whilst the area is marked as a pin. As soon as the pin is released the area becomes susceptible to positional updates based on the velocity.

6.3 Sub question three conclusion

No major bottlenecks were discovered during testing yet minor ways to improve the performance were found. The cloth simulation is not compatible with all tested devices due to the lack of hardware support.

The following devices are reported as supporting the current implementation:

- iPad Pro (2016) - Half floating point texture
- iPad Mini 2 (2013) - Half floating point texture
- iPhone 5S (2013) - Half floating point texture
- Macbook Pro Retina 13-inch (2015) - Full floating point texture
- Lenovo W530 - Full floating point texture

The Samsung Galaxy S7 Edge (2016) is unsupported due to the lack of hardware support for full- and half floating point textures as shown in *Figure 10.4.3*.

Possible ways to improve the performance are as follows:

- Use the minified version of Three.js
- Create a light version of Three.js, the current full build is almost 1.1Mb of JavaScript
- Load the GUI in a different way to avoid forced reflows and recalculation of styles
- Optimize the file size of the textures, the larger the file size the longer it takes to initially load and decode.
- Use base 2 square textures (512 x 512, 1024 x 1024, 2048 x 2048) where possible in order to avoid having to resize.
- Eliminate the need to decode the image by forcing hardware acceleration.
- Compile, minify all scripts and load the JavaScript asynchronously

For cross-platform and cross-browser support I would highly recommend not increasing the particle amount over 10.000 in order to keep the cloth as smooth as possible. Higher particle counts does not result in smoother cloth, as one might expect, but increases the amount of micro fluctuations dramatically as shown in *Figure 10.4.1*. Cloth, in the real world, affected by forces does not act in this way and if the simulation does so it breaks the illusion.

Another reason why one does not want to increase the particle amount, in the current build, is due to the lack of support for rendering full floating point textures on mobile devices.

6.4 Conclusion

To answer the main question I've went through a process of research, testing, documenting and programming. The answers to the sub questions provide the answer to the main question stated as follows:

How does one realistically simulate a Hermès silk scarf using a Newtonian mass-spring system at 60 frames per second utilizing the GPU through WebGL in modern web browsers running on modern (mobile) devices?

One is able to realistically simulate in a Hermès silk scarf in WebGL in the form of a Newtonian mass spring system consisting out of structural, shearing and bending springs with dampers. It is highly recommended to implement particle mass and a general damper as it stabilizes the entire simulation. I've decided to, next to gravity, simulate wind.

Due to the efficient parallel computation power of the GPU the cloth simulation is able to run at 60 frames per second on modern desktop devices and roughly 50 frames (due to framerate capping in modern mobile browsers) on modern mobile devices. I would recommend, in the current build, to use a particle grid of 10.000 particles, 100 x 100 in order to limit the amount of unrealistic micro fluctuations.

One is able to maintain a state on the GPU by rendering to off-screen textures and ping-ponging at the end of the render cycle. Through the process of extrapolation and numerical integration using semi-explicit Euler one is able to accurately compute the next position and velocity based on the previous- and current position and velocity.

I would recommend using half floating point number textures on iOS devices instead of full floating point numbers as the latter is not supported. The Samsung Galaxy S7 is as of writing unsupported.

7. Graduation products

7.1 Graduation products

As a product of graduation I deliver a working prototype of a cloth simulation implemented in WebGL running at 45 - 60 frames per second with 10.000 particles (100 x 100 particle grid) on iOS devices and desktop computers. The cloth is a constructed out of a mass spring system with structural, shearing and bending properties. The positional and velocity values are stored using data textures as framebuffer objects and are extrapolated and numerically integrated using semi-implicit Euler. At the end of a single render cycle the pointers to the textures are swapped, also referred to as ping-ponging.

Users are able to control the camera on both desktop and mobile and is entirely responsive.

Users are able to influence the following parameters using a GUI:

- Control the timestep, damping, particle mass, simulation speed.
- Control the damping and strength of the structural, shearing and bending springs.
- Toggle the wireframe, wind and pins on/off.
- Control the strength of the wind X, Y, Z.

A live demo is available on https://timvanscherpenzeel.github.io/Thesis/. One can find the source code attached or alternatively on https://github.com/TimvanScherpenzeel/Thesis.

8. Discussion and recommendations

8.1 Discussion

Looking back at what has been achieved over the period of the 6th of June 2016 till the 21st of October 2016 I have to conclude that, even though almost nothing went to according to the initial planning, I am proud of what has been achieved throughout my internship.

In the first two to three months I have focused, for the most part, on working in the development team of Random Studio on client work. I've worked on developing websites for Andere Tijden (VPRO), 51Sprints (Het Nieuwe Instituut), Repossi, Remy Martin and a WebGL prototype for Hermès to be launched early next year. With 51Sprints we've won a FWA of the day award and two awards from Awwwards.

After the initial rush the realisation came that I was supposed to be writing on my thesis and not on client work even though I felt that I was learning so much, not just on the development side of things but in general - as part of the team.

After an S.O.S. meeting with the graduation supervisor Matthijs van Veen and multiple internal meetings exploring problems the research problem was defined and written down in its most concrete form.

From then on the process of writing this thesis has been an absolute joy with a concrete path in mind and a planning that I was able to communicate internally. I still did some client work for about a day or so every week but was able to put most of my time into this thesis.
The hours of reading academic papers and watching lectures and plucking apart source code followed by reconstructing the essential elements and implementing many features on top of it has been worth it.

Looking back I feel like I definitely made the right decision to focus on client work for the first two months before starting on my thesis research. This might not have been the initial plan of Saxion but I can honestly say that this is one of the main reasons why I have been offered a job as junior creative developer at Random Studio and will be starting in December 2016.

8.2 Recommendations

For future developers and contributors I would recommend researching and possibly adding the following features:

Cloth simulation:

- Add touch / mouse interaction and raycasting to add interaction with the cloth
- Implement support for area raycasting
- Add cloth tearing, rigid body support and object- and self collision and possible add a fix for z-fighting and clipping issues.
- Add interaction with lights and other objects in the scene
- Add shadow casting and receiving
- Add support for partially transparent textures (PNG's).
- Create particle location specific mass and wind dynamics
- Use higher order methods of numerical integration (Verlet, Runge-Kutta 4th order) to increase the precision

Visual quality:

- Make the pin geometry move when repinned or removed
- Construct a custom phong shader using Shaderchunks available in Three.js

Compatibility:

- Identify the specific reason why the Samsung Galaxy S7 Edge is not supported and implement a fix. The goal is to support all modern mobile devices and modern browsers.
- Identify and document issues regarding the camera controls on touch devices in combination with cloth interaction

To future graduate students:

I would recommend you to focus on client work and getting to know the company internals for the first two months before starting on your graduation research. By doing so you build up credibility in the company and other developers and producers will see what you are able to do and who you are as a person. Once you are seen as part of the team it is far easier to communicate, ask questions or help than when one is considered to be an outsider who is only there for their graduation research. Play the game: don't be the good at doing internship tasks (like preparing lunch or making coffee for clients) but show your passion in the work you want to get better at. Chances are that you are allowed to work for actual clients whilst the other interns take out the trash and complain about never being handed real work. I wish you the best of luck and please, if you want to get a head start in the field, move to Amsterdam for half a year. It will be worth it.

9. Bibliography

Imagery:

- Gorkin, K. (2009). “Physically Based Cloth Simulation”. Retrieved from http://www.gorkin.com/Cloth/cloth.html
- Hermès USA (2016). “Plumes II Vert Blue Roy”. Retrieved from http://media.hermes.com/media/catalog/product/import/S/S01/S011/item/worn/zoom/H001410S-15.jpg
- Mohtashim, M. (2016). “WebGL - Graphics Pipeline”. Retrieved from https://www.tutorialspoint.com/webgl/webgl_graphics_pipeline.htm
- Random Studio. (2014). “Studio”. Retrieved from http://random.nu/studio
- Random Studio. (2015). “Installation view of de Bijenkorf Amsterdam”. Retrieved from http://random.nu/magic/
- Random Studio. (2013). “Dom Pérignon Rosé 2002 Experience [SITE CAPTURE]”. Retrieved from https://vimeo.com/78741027
- Quanser (2016). “Diagram of a mechanical spring damper”. Retrieved from http://www.quanser.com/Content/images/Products_Overview_Pages/QNET-Physics-mass-spring-damper.jpg
- Svjo - Wikipedia. (2013). “Hooke’s law”. Retrieved from https://upload.wikimedia.org/wikipedia/commons/f/f0/HookesLawForSpring-English.png

Literature:

- ACMSIGGRAPH. (2015). “SIGGRAPH University - An Introduction to WebGL Programming". Retrieved from https://www.youtube.com/watch?v=tgVLb6fOVVc
- Arthur, K. (2009). “GPU Based Cloth Simulation”. Retrieved from http://kenarthur.bol.ucla.edu/Programs/Cloth/GPU%20Based%20Cloth%20Simulation%20-%20Ken%20Arthur.pdf
- Fisher, M. (2014). “Cloth”. Retrieved from https://graphics.stanford.edu/~mdfisher/cloth.html
- Havsvik, O. & Gerling, J. & Gräntzelius. “Cloth Simulation”. Retrieved from http://oskarhavsvik.se/projects/Cloth_Simulation/report/TSBK03_Report_Cloth_Simulation.pdf
- Lewin Greg. (2013). “Mass-spring-damper tutorial”. Retrieved from https://www.youtube.com/watch?v=_SYfzEGL7xA
- O’Connor, C. (2003) “Modeling Cloth Using Mass Spring Systems”. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.8719&rep=rep1&type=pdf
- Random Studio (2013). “Dom Pérignon Rosé 2002 Experience [SITE CAPTURE]”. Retrieved from https://vimeo.com/78741027
- Rodriguez, J. (2016). “Math for Game Developers - Particle Simulation (Numerical Integration)”. Retrieved from https://www.youtube.com/watch?v=BIz-wEu0QwE
- Rodriguez, J. (2016). “Math for Game Developers - Spaceship Orbits (Semi-Implicit Euler)”. Retrieved from https://www.youtube.com/watch?v=kxWBXd7ujx0
- Widnall, S. & Peraire J. (2009). “Lecture L4 - Curvilinear Motion. Cartesian Coordinates”. Retrieved from http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-07-dynamics-fall-2009/lecture-notes/MIT16_07F09_Lec04.pdf
- Yalçın, M. Adil & Yıldız, C. (2009). “Techniques for Animating Cloth”. Retrieved from http://www.cs.bilkent.edu.tr/~cansin/projects/cs567-animation/cloth/cloth-paper.pdf
- Zeller, C. “Cloth Simulation on the GPU”. Retrieved from http://http.download.nvidia.com/developer/presentations/2005/SIGGRAPH/ClothSimulationOnTheGPU.pdf

Source code:

- Jones, B. (2015). “GPGPU.js”. Retrieved from https://github.com/toji/webgl2-particles/blob/gh-pages/js/GPGPU.js

10. Annexes

10.1 Foundational knowledge

To understand why one would use the GPU over the CPU for calculations regarding cloth simulations I must first explain the fundamental difference in problems each one faces.

CPU's (central processing units) are microprocessors that, for the most part, are heavily optimized for calculating basic arithmetic, logic, control and do input/output operations as specified by instructions. These instructions are often written in a human readable language such as C and compiled down to a binary format that the CPU is able to understand.

GPU's are heavily optimized to run compute-intensive operations such as video processing, physics simulations and graphics processing. The unit is able to do this by using thousands of very basic CPU cores. These cores are not able to do complex calculations as the traditional CPU but rather quite simple calculations. The power of the GPU lies in the fact that the problems it is optimized for are problems that can be calculated in parallel compared to problems the CPU faces that are most of the time serial. Examples of problems that can be calculated in parallel is for example video processing and physics simulations like cloth simulations.

10.2 WebGL FBO support across devices and browsers - Initial tests

Joshua Koo (ZZ85) - GPU Particle system using framebuffer objects

http://mrdoob.com/lab/javascript/webgl/particles/particles_zz85.html

Figure 10.2.1: The source code of Joshua Koo - Macbook Pro 2015 - Chrome version 53 (64-bit) at ~ 260k particles

The reason why I am interested in using framebuffer objects is because once you have passed the initial data from the web application all further calculations are done on the GPU by copying raw data textures between multiple buffers and calculating the difference between each buffer. Not having to communicate between the CPU and GPU solves a large bottleneck and enables, in this case, to be able to render hundreds of thousands of particles and incredible frame rates on modern devices.

In JavaScript Typed Float32Arrays are used to be able to send vertex data from the CPU over to the GPU. It is necessary to work on the byte level as arrays in JavaScript are actually complicated objects compared to simple C-language-like arrays that the shader compiler expects. The array consist out of 4 bytes, typed meaning that you can reference the position of each bit in the byte using named components. When a bit contains position data the byte component is named *[x, y, z, w]*, with color *[r, g, b, a]* and with texture maps *[s, t, p, q]*. This is useful when you need to create a texture out of data and you need to be able to reference specific components.

```
var width = 512, height = 512;
data = new Float32Array( width * height * 3 );
```

Figure 10.2.2: Example of a Typed Float32Array created to hold the size of the texture.

See *Figure 10.2.3*. Joshua Koo throws an exception when `OES_texture_float` or `gl.MAX_VERTEX_TEXTURE_IMAGE_UNITS` is not supported. This is worth investigating and find out the support across multiple devices Random Studio would like to target. If this the support is problematic I should research alternative techniques used to store data in textures.

```
if( !gl.getExtension( "OES_texture_float" )) {
       alert( "No OES_texture_float support for float textures!" );
       return;
}

if( gl.getParameter(gl.MAX_VERTEX_TEXTURE_IMAGE_UNITS) == 0) {
       alert( "No support for vertex shader textures!" );
       return;
}
```

Figure 10.2.3: Example of an render context test for a certain functionality

See *Figure 10.2.4*. Joshua Koo uses a `WebGLRenderTarget` available from Three.js to render to a texture instead of the screen. He then clones the texture for later re-use.

```
rtTexturePos = new THREE.WebGLRenderTarget(width, height, {
       wrapS:THREE.RepeatWrapping,
       wrapT:THREE.RepeatWrapping,
       minFilter: THREE.NearestFilter,
       magFilter: THREE.NearestFilter,
       format: THREE.RGBFormat,
       type:THREE.FloatType,
       stencilBuffer: false
});

rtTexturePos2 = rtTexturePos.clone();
```

Figure 10.2.4: Usage of the WebGLRenderTarget method from Three.js

See *Figure 10.2.5*. To construct the actual texture Joshua Koo uses `DataTexture` available from Three.js directly from raw data, width and height.

```
texture = new THREE.DataTexture(
data, width, height, THREE.RGBFormat, THREE.FloatType
);
```

Figure 10.2.5: Usage of the DataTexture method from Three.js

The Simulation shader uses a `DataTexture` as an input, updates the particles' positions and writes them back to a `RenderTarget`. This pass uses a bi-unit square (-1.0 to 1.0 by -1.0 to

1.0 in the Cartesian coordinate system), an orthographic camera and the ability to render to a texture. The Render shader uses the `RenderTarget` to distribute the particles in space and renders the particles to the screen.

FboUtils.js is an extension written for Three.js to add the following FBO utilities:

- An orthographic camera aimed at the texture
- A WebGLRenderTarget
- A vertex and a fragment shader to pass the initial texture from the CPU to the GPU using texture2D, available in GLSL
- An off-screen Three.js scene to which the camera is added and the texture is mapped onto a plane at which the camera is targeted.
- A method to render the floating point number data texture to a texture to add onto the plane.
- A method to add the floating point numbers in the Float32Array to the texture.
- A method to update the camera and texture at the main applications render time.

Brandon Jones (Toji) - WebGL2 Particles

https://toji.github.io/webgl2-particles/
https://github.com/toji/webgl2-particles

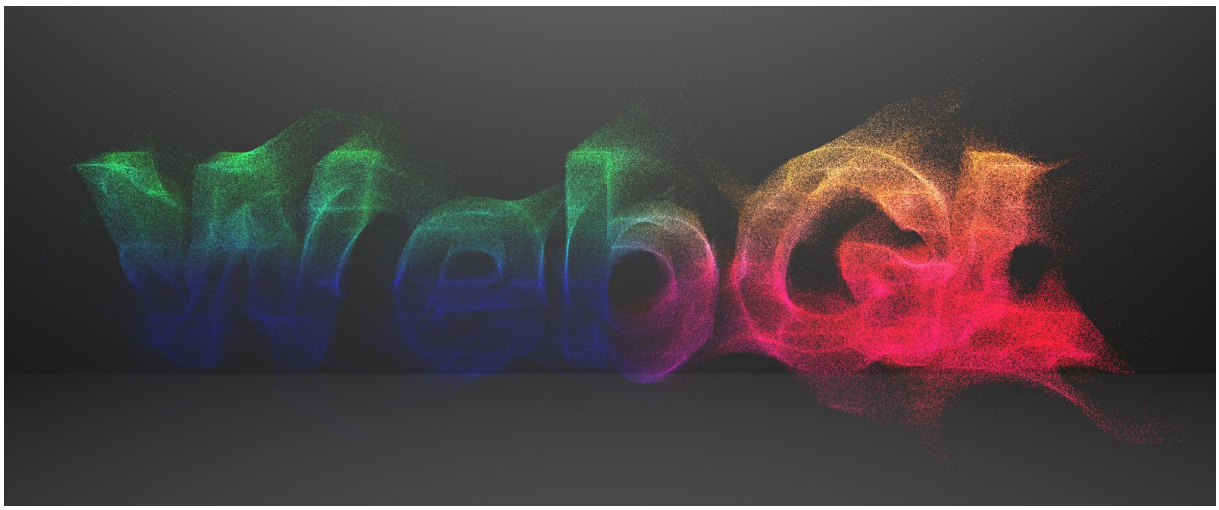


Figure 10.2.6: The source code of Brandon Jones - Macbook Pro 2015 - Chrome version 53 (64-bit) at ~ 1M particles

The demo of Brandon Jones, shown in *Figure 10.2.6*, implements a WebGL2 particle system that uses transform feedback, an alternate approach to FBO's only available in WebGL2. WebGL2 is experimentally supported in Chrome Canary and Firefox Nightly but it will take years before it becomes standardized. The demo also has a fallback for WebGL that uses the traditional FBO technique.

Possible issues

`OES_texture_float` and `gl.MAX_VERTEX_TEXTURE_IMAGE_UNITS` are possibly not fully supported on embedded devices and thus makes implementing FBO's using this technique risky when considering that Random Studio would like cross-device and cross-browser usability and performance.

To test this I will run a WebGL support test on every device Random Studio is targeting and logging all results for future reference. I'll be using the website http://webglreport.com/ as it tests for both `OES_texture_float`, `gl.MAX_VERTEX_TEXTURE_IMAGE_UNITS` and many other interesting information about WebGL support on the browser and the hardware support on the device. The results of the tests can be found here:

I'll test the previously mentioned demo's on the a large range of test devices and will log the possible errors and results.

Conclusion

After testing all the devices mentioned in the sheet I was negatively surprised at how poor the hardware support is on mobile devices. Both the Macbook and older Lenovo laptop were able to run all demo's on a smooth framerate and show a much larger support for WebGL FBO's than mobile devices. The Samsung Galaxy S7 Edge does not have `OES_texture_float` and therefore throws an error. The range of Apple devices (iPad Pro, iPad Mini 2, iPhone 5) all throw silent errors having to do with the FBO technique of moving textures around.

I'll have to seriously consider finding alternate techniques to be able to use a larger range of devices than desktop computers considering that Random Studio would like cross-device and cross-browser usability and performance. I knew the support would be flaky before I started testing the devices by the experience I had of general WebGL support on mobile devices that other developers have shared but had no idea it was this bad. As it stands no mobile device supports the use of FBO's though all devices tested have no issue running basic WebGL scenes.

Next research steps

I've decided to start looking for examples that might use the FBO technique or alternate techniques on mobile devices. It is strange that the all iOS devices report support for `OES_texture_float` even though they throw errors where desktop devices don't.

10.3 WebGL FBO support across devices and browsers - Continued tests

Juan Espinosa (Yomboprime) & Joshua Koo (ZZ85) - GPGPU Birds Flocking

As the first three particle systems failed to run on any of mobile devices I decided to start looking around for an example that I knew that might work and still uses the technique of framebuffer objects. I was positively surprised when I come across a bird flocking example, as shown in *Figure 10.3.1* that runs on the GPU and was lucky to find out that mobile support had been added recently.

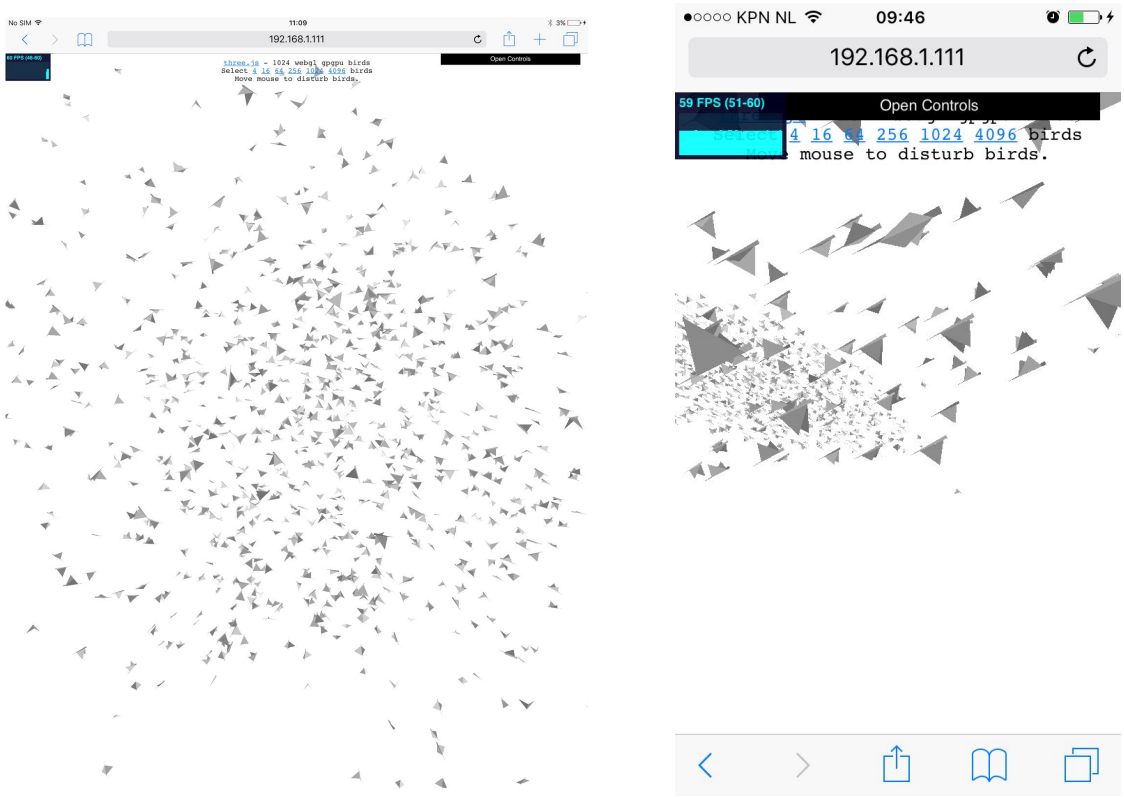


Figure 10.3.1: The source code of Juan Espinosa / Joshua Koo running on a iPad Pro 2016 - Safari and iPhone 5S - Safari

The demo of Juan Jose Luna Espinosa and Joshua Koo implements a flocking algorithm on the GPU that uses the traditional FBO technique. For abstraction purposes they have decided to build a universal wrapper around the FBO texture swapping.

When I dived into the source code and commits on Github relating to the example I quickly found out why this example is working and the other examples, that use the same underlying technique, don't. It turns out that the tested iPads and iPhone actually only support `OES_texture_half_float` even though they report to support `OES_texture_float` as shown in *Figure 10.3.2.*

```
OES_texture_float
OES_texture_float_linear
OES_texture_half_float
OES_texture_half_float_linear
OES_standard_derivatives
```

Figure 10.3.2: WebGLReport from the iPad Pro

In the Three.js repository of the example a commit was added to support iOS devices as shown in *Figure 10.3.3*.

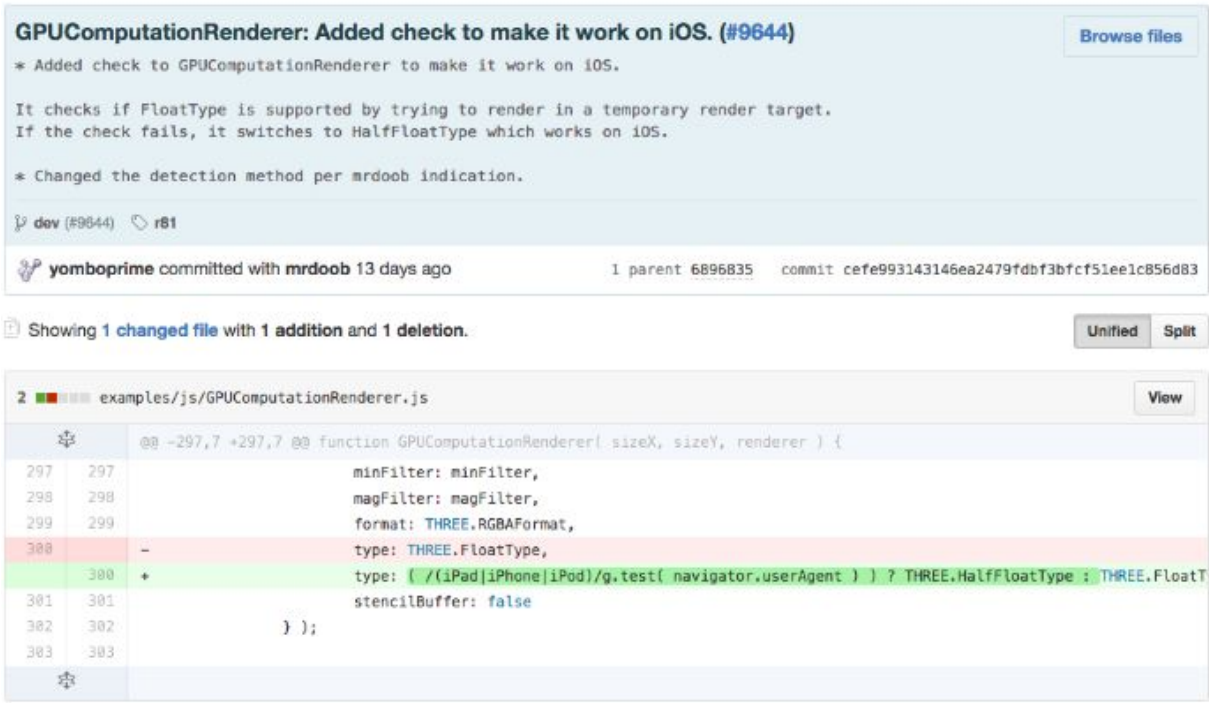


Figure 10.3.3: Commit to GPUComputationRenderer to add mobile iOS support

Juan Espinosa (Yomboprime) - GPGPU Water

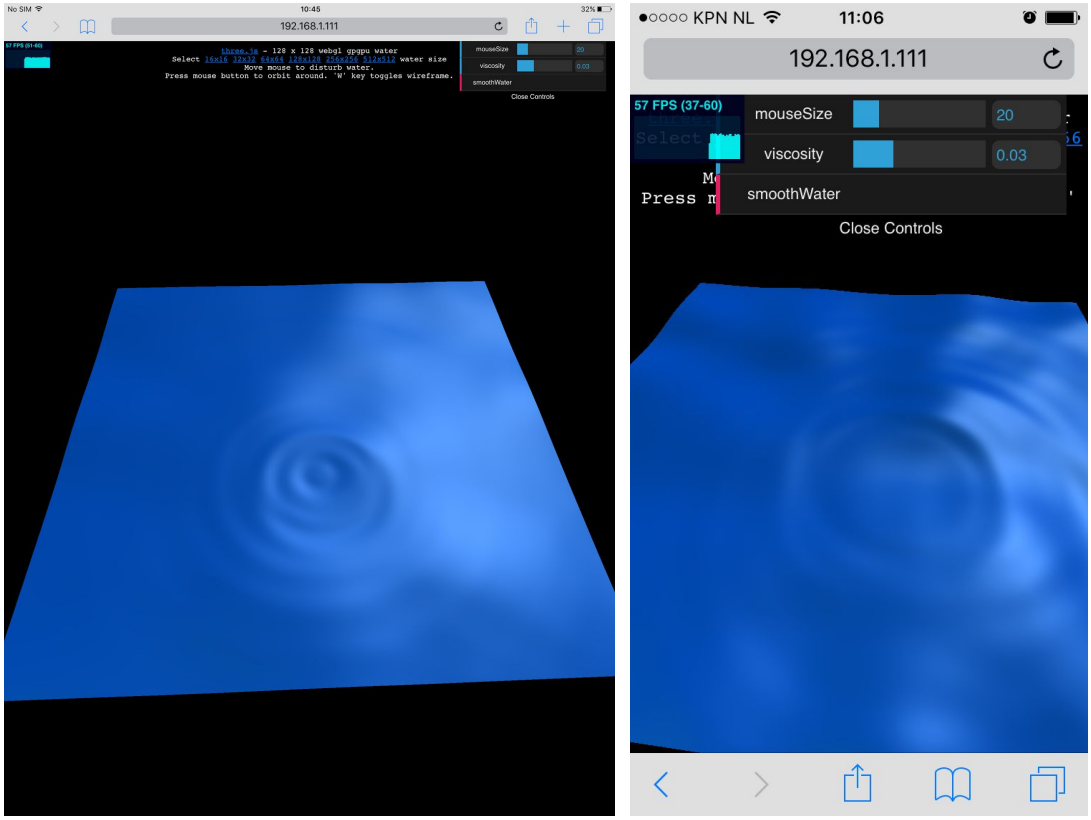


Figure 10.3.4: The source code of Juan Espinosa running on a iPad Pro 2016 - Safari and iPhone 5S - Safari

After I researched the previously mentioned bird flocking example by Juan Espinosa (Yomboprime) I decided to look further into related work he had made. One of the examples I came across was a simplified water simulation, as shown in *Figure 10.3.4*, that computes its simulation on the GPU, like the bird flocking example. Internally it uses the same GPUComputationRenderer.js to add FBO ping-pong functionality out of the box.

My specific interest in this example goes into the user interaction with the water layer (as the user influences the FBO texture layer in a direct way), the wireframe option and setting up the possibility to use a texture.

An important note is that the orbit controls are very clunky to use when using the simulation on touchscreen devices as you move the camera if you actually want to move the water and thus it only allows for tapping on the touchscreen. This is something to think about: should I limit the user interaction with the camera to improve the interaction with the water? The GUI (dat.gui.js) is also not functioning as intended though I don't consider this to be a real issue due to the fact that in final products there will never be a GUI to interact with.

The source code of GPUComputationRenderer is well commented and has instructions on how to implement it in the application so I won't expand on it here.

Research results

The GPGPU Birds Flocking example shows no errors or warnings except for known non-functionality of the Samsung Galaxy S7 Edge due to it not supporting `OES_texture_half_float` or `OES_texture_float`.

The GPGPU Water example shows a warning on the debug log, as shown in *Figure 10.3.5,* but is not application breaking and I will therefore not consider it that much of an issue at this point in time. What is breaking is the functionality of OrbitControls.js to move the camera around. It simply does not work well together with other events on touchscreen devices. I'll research alternatives in the future to approach this problem. At this point I'm thinking to use the OrbitControls on desktop and device orientation controls on mobile as that seems to work quite smoothly at first sight and requires no touch to orientate and thus I can use the touch abilities properly as input in the shader.

Figure 10.3.5: The source code of Juan Espinosa running on a iPad Pro 2016 - Safari gives a warning but functions well.

Conclusion

I'm very happy to find a working example that offers cross-device, except the Samsung Galaxy S7 Edge, and cross-browser support. The Samsung Galaxy S7 Edge simply does not support `OES_texture_half_float` nor `OES_texture_float` and is therefore unable to compute the textures off-screen necessary to compute the velocity of a particle. I will not be targeting Android devices and will primarily focus on maintaining cross-device support on iOS. Once Android device hardware, like the Samsung Galaxy S7 Edge, do support `OES_texture_half_float` nor `OES_texture_float` the flocking example should be able to run without any issue. Of course I've only tested a single Android device but if one of the newest premium smartphones does not support it I doubt lower end devices do.

I'll dig further into the source code of the GPGPU Bird flocking example and port whatever possible to my codebase to be able to start doing FBO testing on particle grids.

10.4 Cloth simulation compatibility tests

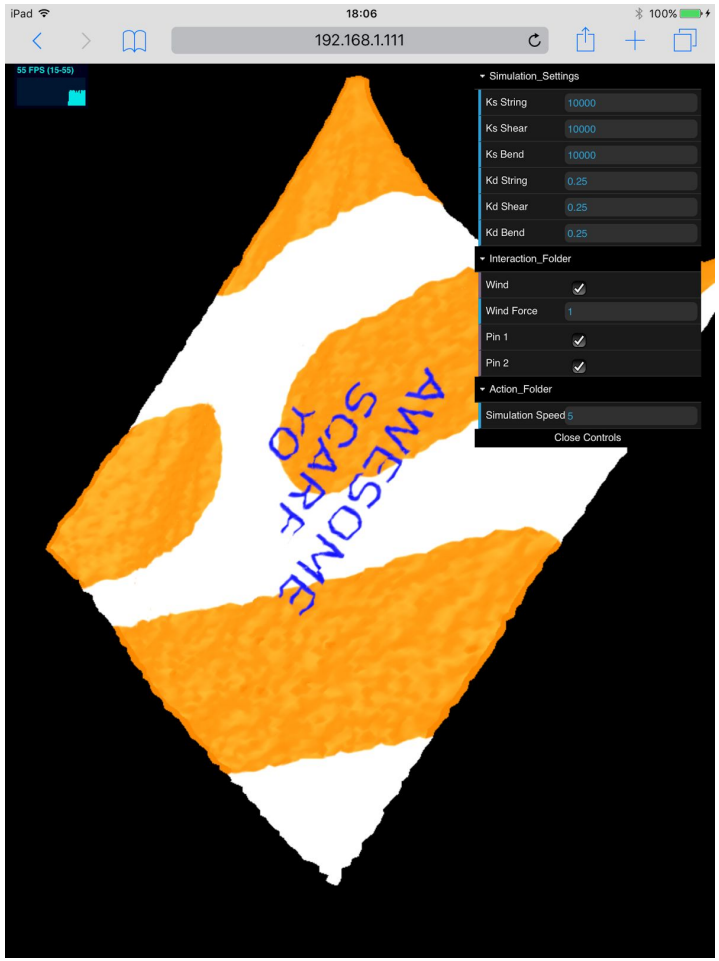


Figure 10.4.1: Early prototype showing microfluctuations caused by high particle counts and half floating point textures (iPad Mini 2, 2013)

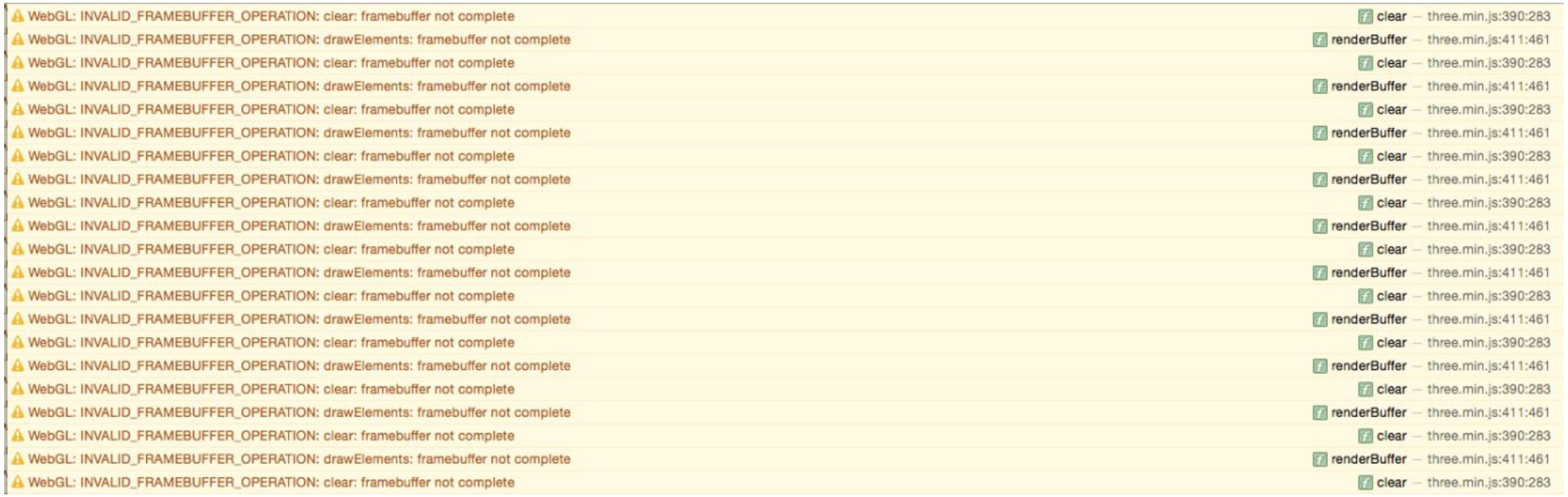


Figure 10.4.2: Debug messages on the iPad Pro indicating the lack of support for OES_texture_float

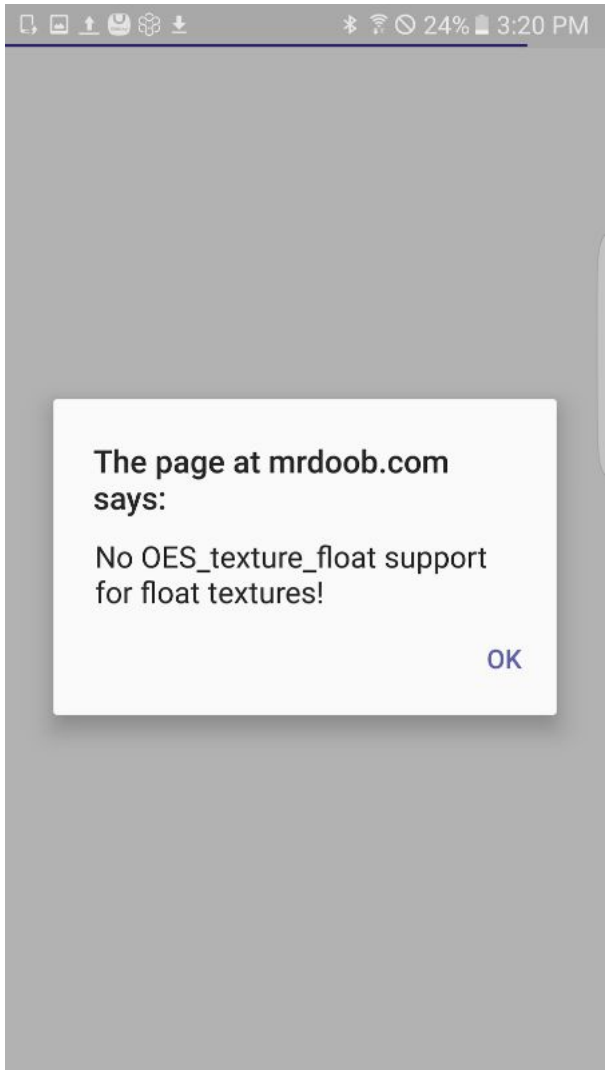


Figure 10.4.3: Error message on Samsung Galaxy S7 Edge indicating the lack of support for OES_texture_float

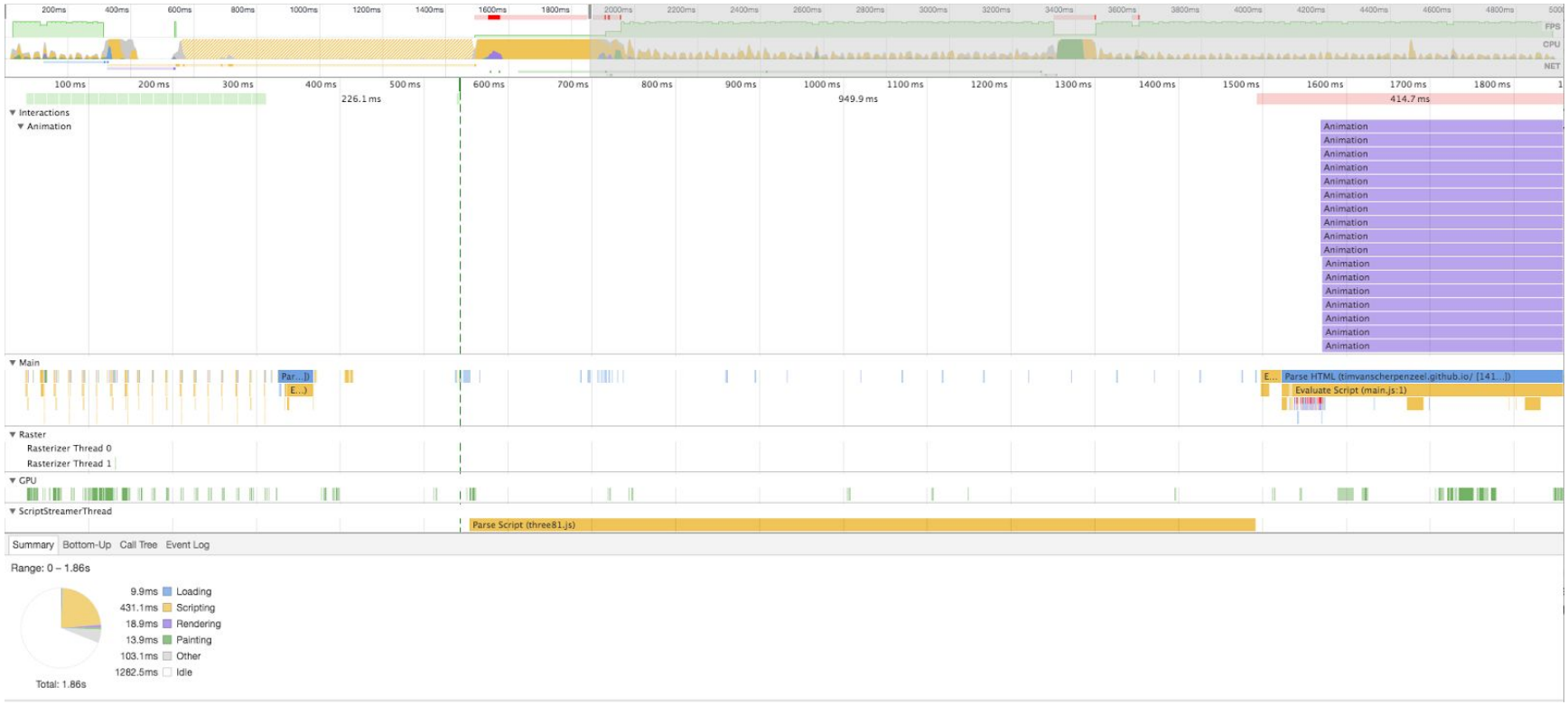


Figure 10.4.4: Initialization phase of the cloth simulation as seen in the timeline of Chrome developer tools

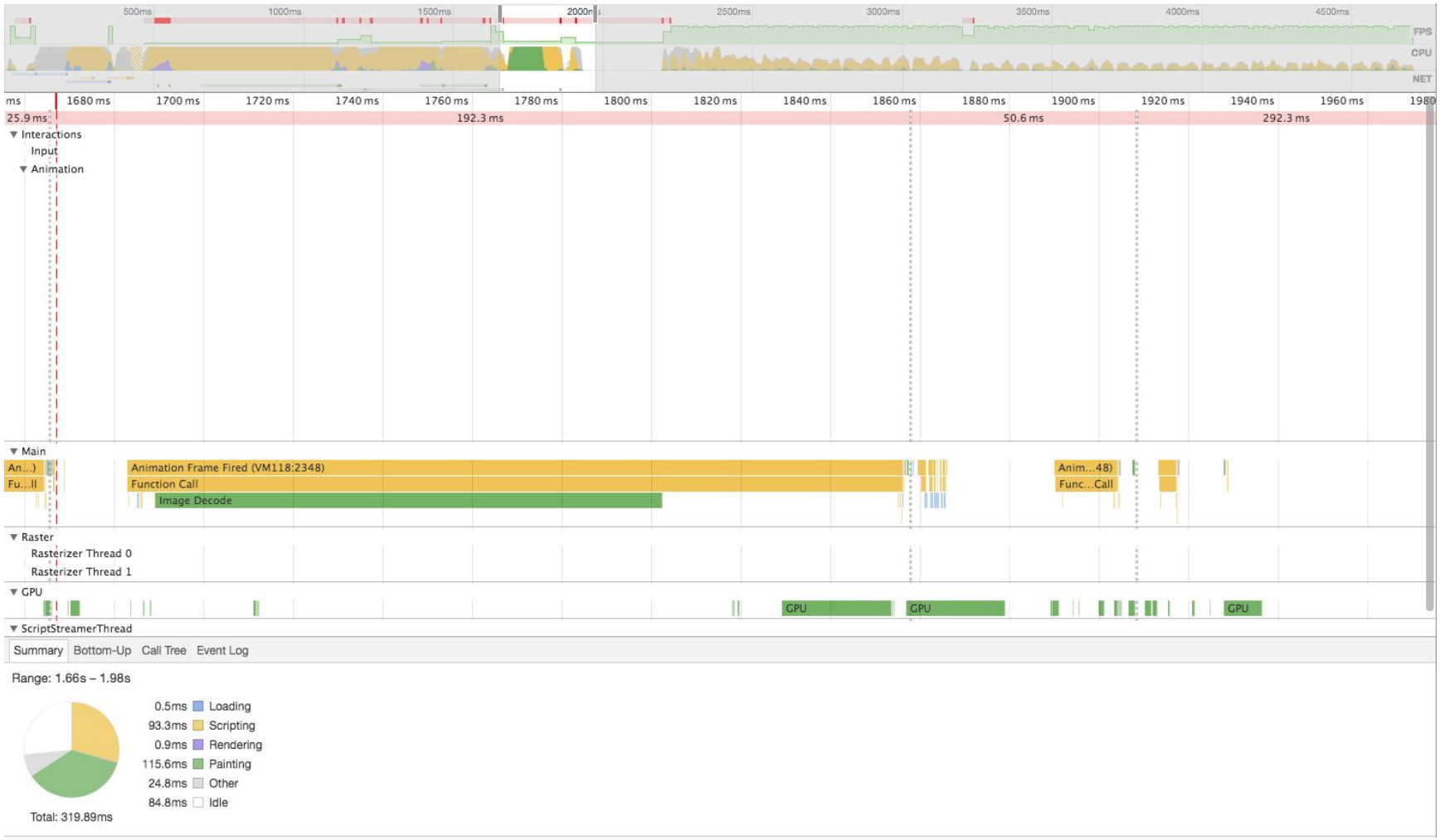


Figure 10.4.5: Image decode phase of the cloth simulation as seen in the timeline of Chrome developer tools

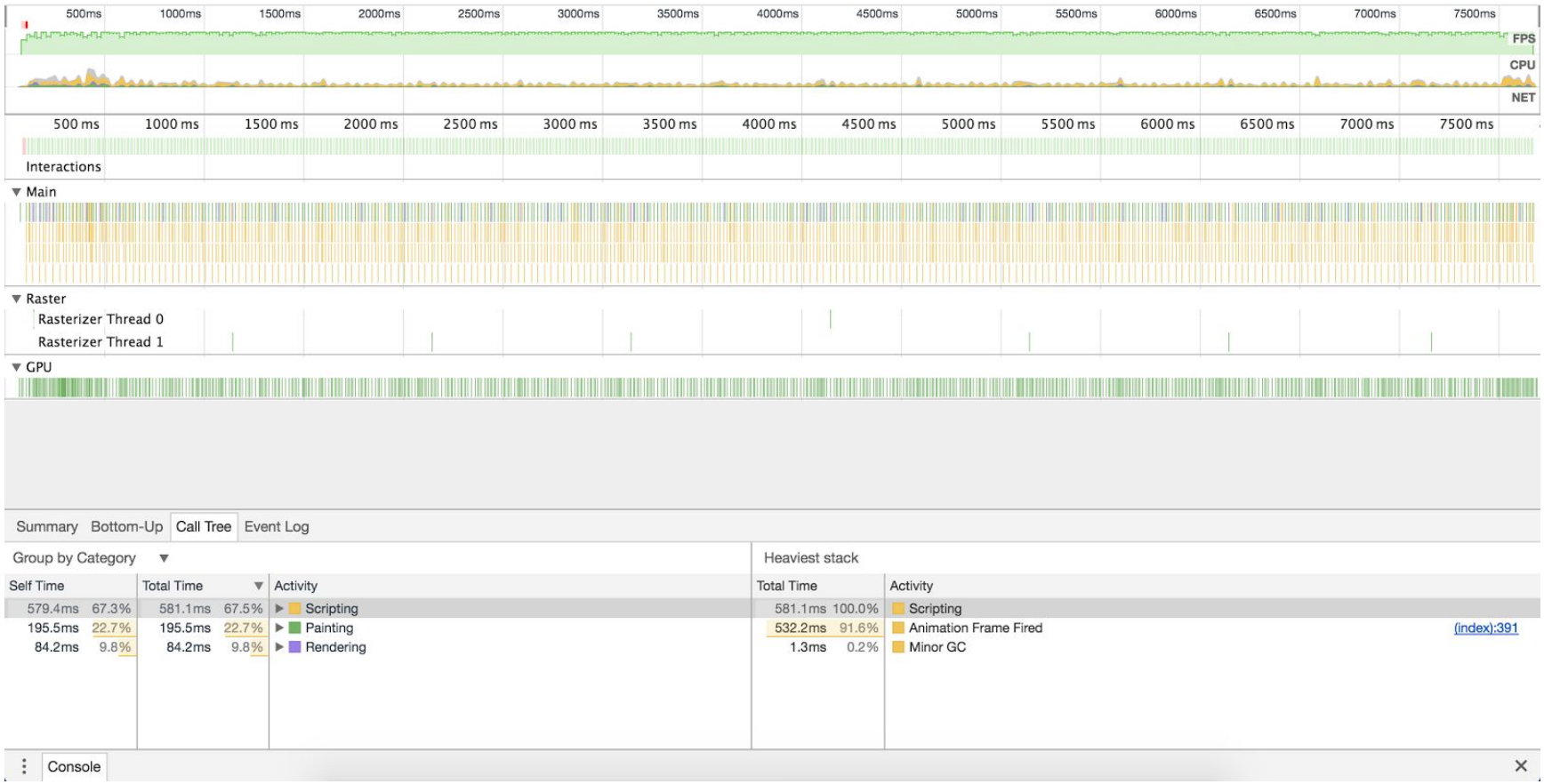


Figure 10.4.6: Stable GPU phase of the cloth simulation as seen in the timeline of Chrome developer tools (overview)

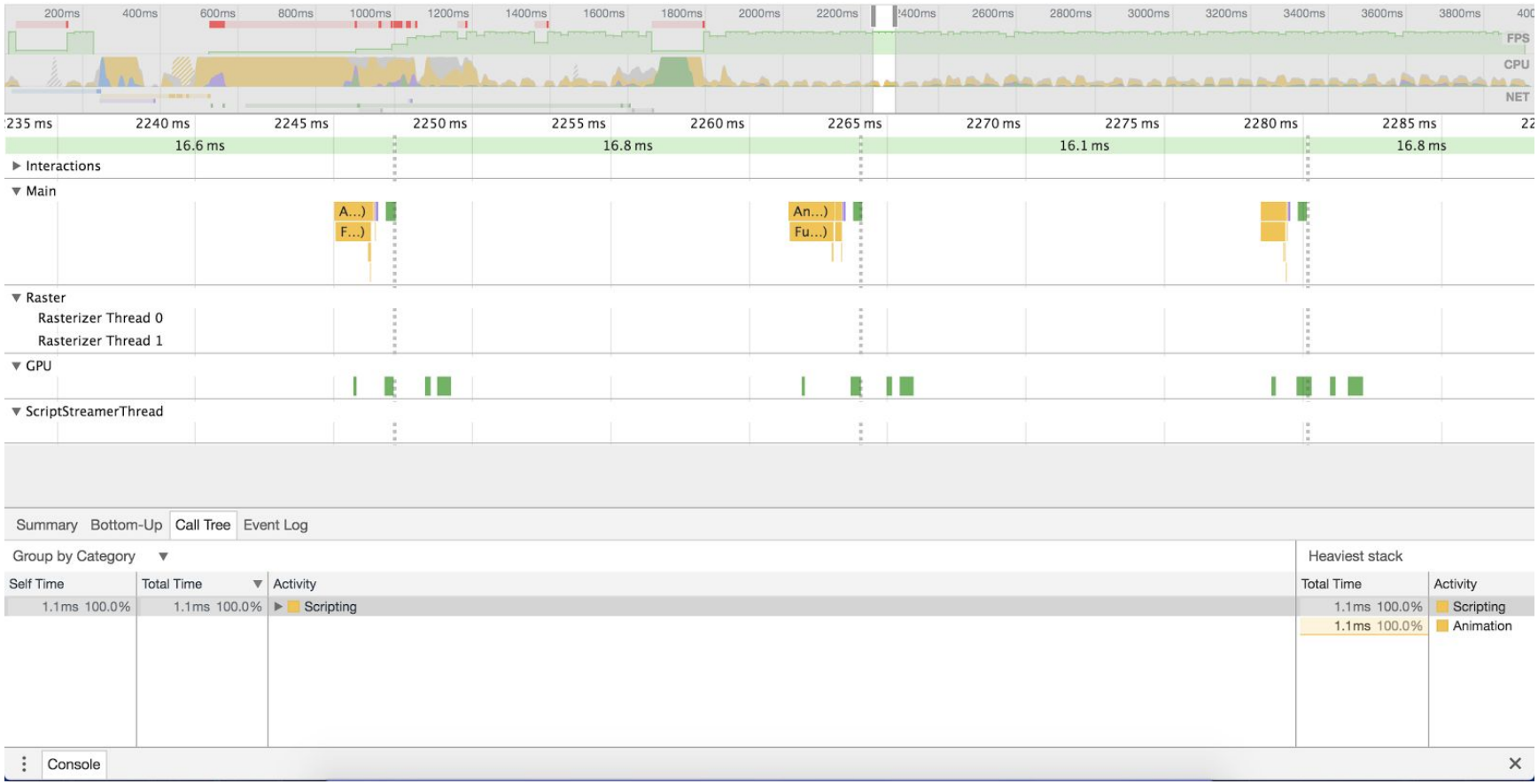


Figure 10.4.7: Stable requestAnimationFrame call of the cloth simulation as seen in the timeline of Chrome developer tools (detail)

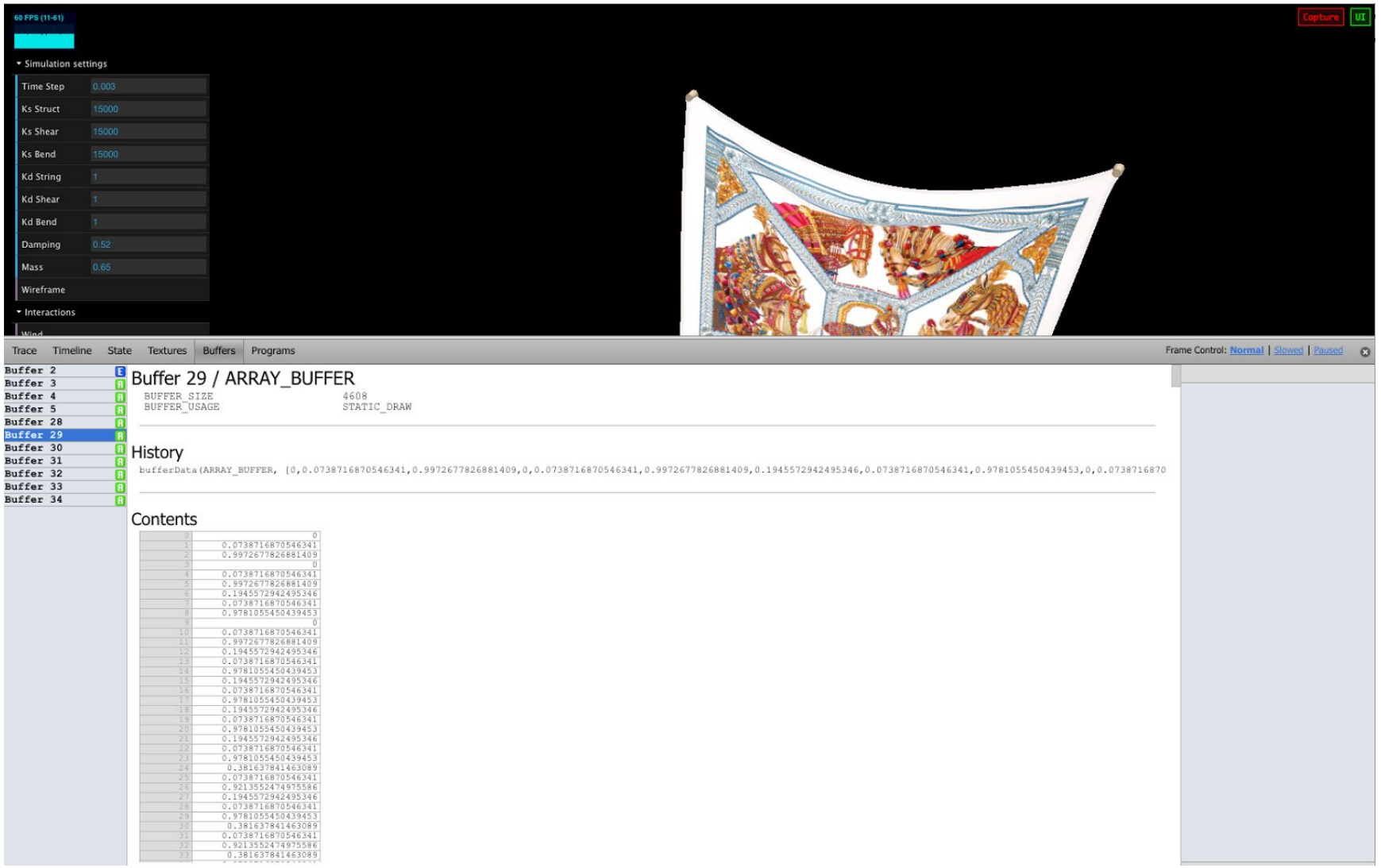


Figure 10.4.8: Off-screen data texture shown as array as seen in the texture tab of the WebGL inspector add-on