Implementation of a PIC simulation using WebGL

based on OpenPixi

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Leonard Weydemann
Registration Number 1028488

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Dr.techn. Andreas Ipp

Vienna, 22.04.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Leonard Weydemann
Reinprechtsdorfer Str. 52/7, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

This project's aim is to find a WebGL based alternative to the Java implementation of OpenPixi, a Java-based Particle-in-Cell (PIC) simulation software, and to add a third dimension. For this purpose, an existing JavaScript library, three.js, was chosen. A handful of approaches are explored and the resulting prototypes are then compared in terms of speed, as performance is a main concern. A shader-based implementation, the best performing of the prototypes, is then explained in more detail and recommendations for the future development of OpenPixi are given.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Particle-in-Cell Simulation . . . 1
- 1.3 OpenPixi . . . 2
- 1.4 WebGL . . . 3
- 1.5 Three.js . . . 3
- 1.6 General-Purpose Computing on Graphics Processing Units . . . 3
- 1.7 Related Work . . . 6

2 Implementation **7**
- 2.1 Methodology . . . 7
- 2.2 Usage . . . 9
- 2.3 Portability . . . 9
- 2.4 Resources . . . 9
- 2.5 Canvas Prototype . . . 10
- 2.6 Three.js Prototype in Two Dimensions . . . 10
- 2.7 Three.js Prototype in Three Dimensions . . . 10
- 2.8 GPGPU Prototype . . . 10

3 Results **17**
- 3.1 Testing Environment . . . 17
- 3.2 Comparison . . . 18

4 Conclusion and Future Work **21**

A Screenshots **23**

Bibliography **27**

CHAPTER 1

Introduction

In this chapter I will introduce some background knowledge about the problem at hand. In the following chapter I will explain the prototypes and the methodology behind them. Finally I will compare the prototypes' performance and give an interpretation of the results. The paper then concludes with recommendations for future work.

1.1 Motivation

The motivation behind this piece of work was to effectively use the Graphics Processing Unit (GPU) for OpenPixi, a Java-based Particle-in-Cell (PIC) simulation software. On the one hand it should be used for rendering the graphics, and on the other hand for computing the particle's movement. Currently OpenPixi's web page displays its Java application, although it is possible to download it and to run it locally. As Java applications tend to run slowly in browsers and web-accessibility was a main concern, WebGL was chosen to develop an alternative application in, because it is included in all modern web-browsers and makes use of the GPUs computing powers.

1.2 Particle-in-Cell Simulation

Particle-in-Cell (PIC) simulators simulate motion of charged particles in an electro-magnetic background, so-called plasmas. The motivation for examining plasma physics is the estimate that 99% of the matter in the universe is in the plasma state. PIC methods have been in use since as early as the 1960s [1]. Plasma simulations are used to describe magnetically- and inertially-contained fusion plasmas and to gain understanding of plasmas in space and man-made plasmas that occur in ion guns, microwave devices or nuclear explosions. They have also been applied to problems in solid and fluid mechanics [2], [3], [4]. Generally PIC codes use a grid, where electric and magnetic fields are defined on the grid points through which the particles move. For each time-step the next position is calculated according to the particle's position inside the grid

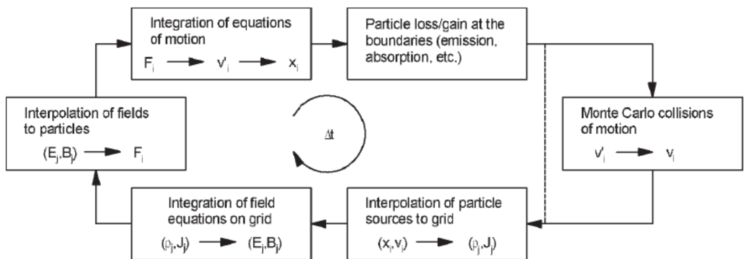

Figure 1.1: Flow Schematic for the PIC Scheme taken from Verboncoeur (2005) [1]. When using a collisional model, Monte Carlo collision schemes are applied.

and its set of properties. Equations associated with these kind of codes are the Lorentz force as equation of motion, solved in a so-called particle mover and Maxwell's equations calculated in the field solver to determine the electric and magnetic fields.

A PIC code usually consists of four components (see figure 1.1):

- Particle Mover:
  The particle mover solves the equations of motion for each particle. It also implements prepare and complete methods, which synchronize the velocity and the position in time. This is used to perform collisions among particles and between particles and boundaries
- Interpolator (particles to grid):
  Charged particles produce electrostatic fields when they are atf rest. Moving charged particles produce currents which in turn produce magnetic and electric fields. These charge and current source terms need to be interpolated to the field mesh.
- Field solver:
  Solves Maxwell's equations or more generally, partial differential equations.
- Interpolator (fields from grid to particle):
  Interpolates between particles at arbitrary positions and grid points.

1.3 OpenPixi

OpenPixi is an open-source PIC simulator developed in Java. Several constants of the simulation like the number of particles or strength of certain forces can be changed interactively. It is also possible to switch between different models for solving the equations. Future plans include a 3D version and interactive distributed versions.

As the program grew gradually in size and complexity, the code soon lacked a clear model. Initially it was planned to refactor the code into something more easily maintainable. Various 'code smells' can be detected, a fact that further promotes refactoring. Code smells are symptoms in the code that indicate a deeper problem but which do not keep the program from functioning properly. Consequently modularity, readability, and maintainability were kept in

mind when developing the prototypes. JavaScript, being a multi-paradigm language, supports many possibilities for this matter. Transporting existing Java code to JavaScript is thus relatively easy.

1.4 WebGL

The Web Graphics Library (WebGL) is a JavaScript 3D/2D graphics application programming interface (API). It is royalty-free and based on OpenGL ES 2.0 which is exposed through the HTML5 Canvas element as Document Object Model interfaces. Furthermore WebGL is also a shader-based API using the OpenGL shading language (GLSL), which enables code to be executed on a computer's GPU. WebGL as a web-standard is integrated into the major browsers like Safari, Google Chrome, Mozilla Firefox and Opera, and enables plugin-free 3D applications. It is designed and maintained by the non-profit Khronos Group. It was initially released on March 3, 2011 [5].

1.5 Three.js

Three.js is a lightweight JavaScript library that abstracts away the lower-level API calls. This way it makes developing WebGL applications easier and more productive. Three.js was created by Ricardo Cabello Miguel (known by his online handle - mr.doob) and the first version was released in 2010. Three.js is open-source with its source code located on github. It is included within a web page by linking to a local or remote copy. As the first versions rendered to Scalable Vector Graphics (SVG) and Canvas and only later adapted to WebGL, three.js offers a graceful fallback for simple scenes when WebGL is not supported by rendering to canvas or SVG. It was chosen mainly or its popularity and ease of use [5].

Figure 1.2 should serve as an introduction to three.js and aid in reading the prototype's source code. This code initializes a three.js scene complete with renderer and camera. It also creates a colored cube and places it in the scene and rotates it per render-call.

1.6 General-Purpose Computing on Graphics Processing Units

General-purpose computing on graphics processing units (GPGPU) is a term used when computations which are usually done on the central processing unit (CPU) are executed on the GPU, that is usually reserved solely for computer graphics. The GPU offers solutions for applications that need high arithmetic rates and data bandwidths. While the CPU is optimized for high performance on sequential code, the graphics computations on the GPU are of highly parallel nature. The speed, increased precision and expanding programmability of the hardware make it especially attractive for general-purpose computing. For this reason NVIDIA's compute unified devices architecture (CUDA) and the open computing language (OpenCL) have become quite popular. CUDA is a parallel computing platform and programming model that allows direct access to the parallel computational elements of supported GPUs. OpenCL, a framework for parallel programming of modern processors, also allows developers to use the GPU. An

```
<script src="js/three.min.js"></script>
    <script>
      var scene = new THREE.Scene();
      var camera = new THREE.PerspectiveCamera(
        75, window.innerWidth/window.innerHeight, 0.1, 1000);

      var renderer = new THREE.WebGLRenderer();
      renderer.setSize(window.innerWidth, window.innerHeight);
      document.body.appendChild(renderer.domElement);

      var geometry = new THREE.CubeGeometry(1,1,1);
      var material = new THREE.MeshBasicMaterial({color: 0x00ff00});
      var cube = new THREE.Mesh(geometry, material);
      scene.add(cube);

      camera.position.z = 5;

      var render = function () {
        requestAnimationFrame(render);

        cube.rotation.x += 0.1;
        cube.rotation.y += 0.1;

        renderer.render(scene, camera);
      };

      render();
    </script>
```

Figure 1.2: Creating a basic scene in three.js: the basic components scene, camera and renderer are the core of any three.js application. A render function that repeatedly calls *renderer.render(scene,camera)* is needed to make any object added to the scene appear. In this case it is a simple cube. The *requestAnimationFrame(render)* method calls the render function again but only if the tab is currently displayed. This aims to help save resources. The code is taken directly from the official three.js homepage [6]

OpenCL implementation for OpenPixi also exists, and since WebCL was released in March, 2014, it might be considered for future development. WebCL, which is a JavaScript binding to OpenCL, is not natively supported on current browsers at the time of this writing. Consequently it was not considered in this project. To use WebGL for GPGPU, one needs to recast computation into graphic terms. What this means is described in the following.

All of today's GPUs organize their computations in a similar matter called the graphics pipeline. This pipeline consists of several stages further explained abridgedly: The input of the pipeline is a list of vertices in object coordinates. The first stage transforms these vertices from object to screen space, performing lighting calculations and assembling them into triangles. Next, the rasterization stage determines the screen positions covered by said triangles and interpolating per vertex values resulting in a fragment for each pixel location. In the fragment

```
<script id="someShaderID" type="x-shader/x-fragment">

  //some shader code...

</script>
```

Figure 1.3: Code for a fragment shader. 'x-fragment' in line 1 can be replaced with 'x-vertex' to denote it as vertex-shader. The shader's ID can be any string. The shader code needs to be compilable and written in GLSL.

stage, color is computed for each fragment using the interpolated values from the previous step. This step can also take values from a texture. The vertex and fragment processing are usually programmable by the user in form of custom vertex and fragment shaders.

Three.js allows shader programming by making use of a shader material which contains a list of uniforms (read only variables that are consistent across vertex and fragment shader, and serve as input), a fragment shader and a vertex shader. The shader-programs are written as text inside the script and read out by three.js by pointing at them. The language used for shader-programming is GLSL. The GPGPU prototype uses a method called render to texture (RTT). This means that the particles' coordinates are read, updated and written to a texture for each render-loop-iteration. It is explained in detail in section 2.2.

GPGPU inside three.js

In chapter 2 I propose a three.js prototype that uses custom shaders to implement GPGPU. In this paragraph I want to give an overview of how shaders are used inside three.js. How GPGPU is applied exactly can be found in section 2.8. In three.js both vertex and fragment shader can be written by hand, the latter executing the bulk of the calculations. The vertex shader operates on every vertex, while the fragment shader operates on every fragment produced by rasterization. In the general case, this means changing the vertex' position and interpolating data between vertices that form a fragment. Shader code is included by adding a script in the manner found in figure 1.6. The code needs to be compilable and written in GLSL. GLSL itself is a C/C++ similar high level programming language for parts of the graphic card. The main types of GLSL are *float*, *int*, *bool* and *sampler*. Vector and matrix types are also supported. Samplers are used mainly for reading textures. Furthermore three types of inputs and outputs exist: attributes, uniforms and varyings. Attributes, which are input values that change every vertex can only be used for the vertex shader. Uniforms do not change during a render pass, but can be used by both shaders. Attributes as well as uniforms are read only. Varyings are read- and writable in the vertex shader but read-only in the fragment shader. Additionally three.js offers a few pre-defined uniforms to make programming shaders easier. These uniforms consist of matrices used for viewing and modeling transformations. These are used to transform coordinates from object space to world and camera space and camera position. To run the shader code it must be used in a shader material and the material in turn applied to a mesh (figure 1.4). The mesh also needs to be added to a scene and rendered to make sure the shader-code is actually run.

```
[...]
    var someShader = new THREE.ShaderMaterial( {

        uniforms: {
            someFloat: { type: "f", value: 0.0 }
        },

        vertexShader:
        document.getElementById('someVertexShaderID').textContent,

        fragmentShader:
        document.getElementById('someFragmentShaderID').textContent

    } );

    var mesh = new THREE.Mesh(new THREE.SomeGeometry( ... ), someShader);
    scene.add(mesh);

[...]
```

Figure 1.4: A ShaderMaterial is created and applied to a mesh of arbitrary geometry. Uniforms are an array of values of certain types. The types possible are accepted types of GLSL. The names of the values need to match the names inside the shader's code.

1.7 Related Work

As mentioned above, particle-in-cell simulation has been employed since the 1960s. So a vast amount of related literature can be found. With regards to computer simulation recent work seems to have concentrated on enhancing performance and advancing physical models which aim to make results more accurate [7]. Furthermore solutions for problems of numerical analysis in this field are developed further as more elaborate physical models need more complex integration techniques [8]. A broad overview of PIC simulations and plasma simulation in general can be found in Birdsall's and Langdons's book “Plasma Physics via Computer Simulation” [4]. Birdsall's and Langdon's book together with Verboncoeur's article [1] are also cited by OpenPixi's github page. Using the GPU highly parallel architecture for custom programming has only been possible for a couple of years. Utilizing parallel programming for PIC simulation has been considered for example in [9] and [10]. An approach similar to ours, using frame buffer objects and textures in the Open Graphics Library can be found in [11]. This project however takes on a novel approach: albeit using existing ideas and concepts, it proposes a browser-based application with usability in mind.

CHAPTER 2

Implementation

In this chapter the implementations themselves are discussed. Although it was clear from the beginning that a WebGL Implementation was the main goal of this project other web-based alternatives to a Java application were considered. The most basic approach is a prototype using the HTML5 Canvas element without the help of a library. The next prototypes explore the possibilities of three.js. The final one explores a more sophisticated approach to the problem. Screenshots of the prototypes can be found in the appendix.

2.1 Methodology

At first a very basic prototype was designed, which shows the capabilities of a plug-in free, HTML5-based approach. The other prototypes use three.js for creating the scene. The last one explores the possibilities of GPU enhanced calculations. To ensure the equal calculation of frames per second I used the library *stats.js* across the three.js versions. In the canvas prototype I calculate them by hand. I tested these calculations against the ones done by *stats.js* and found them to be the same. The FPS are influenced chiefly by the number of particles. All of the prototypes utilize a render and simulation loop. The latter, calling itself, calculates the FPS or updates *stats.js*. The time-steps executed per simulation call can be altered. Changing the attributes however requires a reset.

For graphical user interfaces (GUI), many JavaScript libraries can be found across the internet. For three.js *dat.gui* has been used widely. I also included it in the three dimensional prototypes. While its appearance is limited, it still offers many control elements. It supports folders, presets, saving values and listening for events. Simple buttons can be added by making a GUI-element call a function.

The prototypes all use a rather basic model for particle movement. The simulation consists of particles and a bounding box. OpenPixi uses a solver class which makes use of a certain force to calculate the motion of particles. Various solvers are available to choose from in the OpenPixi application. These include Euler, Semi-implicit Euler, Euler Richardson and Leapfrog. Each

implements its own model for motion. For the prototypes the Euler-solver was chosen for its simplicity. This solver calculates acceleration, velocity and sets the position of the particle for each time-step. This is done in the following way: The force $\mathbf{F}$ that is being exerted on the particles in our model results from multiplying the velocity of the particle with the negative drag constant $-d$ and the constant gravity vector $\mathbf{g}$. It is modeled as a function of the current velocity $\mathbf{v}_t$ and the particle's current position $\mathbf{x}_t$.

$$\mathbf{F}(\mathbf{v}_t, \mathbf{x}_t) = -d * \mathbf{v}_t + m * \mathbf{g} \tag{2.1}$$

The current acceleration-vector $\mathbf{a}_t$ is computed by dividing the current force $\mathbf{F}$ by the particles' mass m:

$$\mathbf{a}_t = \frac{\mathbf{F}(\mathbf{v}_t, \mathbf{x}_t)}{m} \tag{2.2}$$

In the next step the new position is set by adding the particles' current velocity times the time step dt to the previous position:

$$\mathbf{x}_{t+dt} = \mathbf{x}_t + \mathbf{v}_t * dt \tag{2.3}$$

Finally the new velocity is updated as well by multiplying the previous with the acceleration:

$$\mathbf{v}_{t+dt} = \mathbf{v}_t + \mathbf{a}_t * dt \tag{2.4}$$

This is done for each particle and render-loop iteration. Boundaries at the edge of the simulation area are detected and the direction of the particles is changed accordingly. This means that as soon as a particle's position is outside boundary limits along a certain axis, the sign of the corresponding part of the velocity vector is changed. It would be possible to include the particles diameter when detecting boundary limit collision, but this was not considered for the prototypes. The above suffices to benchmark the prototypes in terms of speed. Grid and collision detection have not been implemented.

Parallelization

The fact that we have a large amount of particles with equal properties, requiring the same calculations for every simulation step, makes this a suitable problem for parallelization. Since vertices and fragments can be calculated in a parallel manner on the GPU, the GPGPU prototype interprets every particle as a vertex and every particle's position is written to a texture by the fragment shader. The fragment shader does this by writing the position on this texture as values usually interpreted as color. The RGB values thus become XYZ values. The vertex shader then reads the texture's values and updates the vertices positions accordingly. This happens for every particle in a parallel way: The vertex shader reads the position texture and updates the particle's position on screen and the fragment shader updates the position texture by calculating the next position. The GPU's parallel computation methods can be applied to our problem in this manner. At least particle movement is accounted for this way and high numbers can be simulated without a large decrease in performance. Depending on the model used for collision

detection, this principle could also be applied. A grid could be implemented for example and collision detection would concentrate only on grid-cells which contain particles. Each grid-cell could then be simulated in a parallel way. In the proposed model for particle movement a texture is used where each pixel holds four values (for red, green, blue and opacity). While this model is particularly useful for storing the particle's position vector, it becomes difficult to use, when more complex data structures are required. Although more models for collision detection exist, they would still have to be translated into graphic terms when the development is limited to WebGL.

2.2 Usage

Each prototype can be started by opening the respective HTML-file in a Chromium-based browser, Firefox or Internet Explorer. WebGL was not successfully tested with Safari. It is necessary to have the scripts located in the correct directories. As mentioned above a few parameters can be set by the user. The fill-in form is replaced by *dat.gui* after the third prototype. Also it is possible in this implementation to follow random particles by pressing the 'S'-key and to terminate this mode with the 'A'-key. This was done to have a further look at the possibilities of three.js. Navigating in the three-dimensional prototypes is done by pressing the left mouse button and moving the mouse around. Holding down the scroll button and moving the mouse back and forth zooms in and out. The right mouse-button in addition enables moving the 3D view from left-right or up-down. In the case that WebGL or some of its features are not supported, an error-message appears. Error messages from inside WebGL or JavaScript are printed to the console, which should be checked if the application is behaving erroneously.

2.3 Portability

As of now WebGL is at least partially supported in all modern web-browsers. While mobile browsers do offer WebGL, there are a only few devices which fully support it. There also seems to be trouble between the Almost Native Graphics Layer Engine (ANGLE) which translates OpenGL calls to Window's DirectX calls when porting from Linux to Windows. Making use of floating point textures, as is done in the last prototype, is currently not supported on mobile hardware. Sometimes the use of textures inside the vertex-shader is not supported as well. It should also be noted that most GPUs put a limit on the size of textures.

2.4 Resources

The canvas prototype is modeled after an application simulating molecular dynamics. It can be found at [12]. For the three.js prototypes the official examples, found in the source code examples-directory have been a lot of help. Contained within the code, a few very useful scripts have also been found. Details on what scripts are used exactly can be found inside the prototype-code. The GPGPU-birds example [13] deserves most attention, as it serves as the basis for the GPGPU-prototype.

2.5 Canvas Prototype

This prototype uses the HTML5 Canvas element to render the particles on screen. All calculations are done inside JavaScript. To simulate the particle movement, a particle object encapsulates mass, position, acceleration and velocity. A solver taken from OpenPixi's source-code and pre-defined force is used to calculate the next position. The Euler-solver calculates the acceleration depending on mass, gravity and drag. The velocity is calculated dependent on acceleration and delta-time, finally the coordinates according to velocity are set. The delta-time is a constant set to control he the speed of motion. To throttle speed the user can change the steps per frame. The movement is calculated for every particle in the particle array of size N. The *paintCanvas()* method, then again iterates over the updated particles to draw them. To enhance performance, the particles are first drawn on an off-screen canvas. This means that only one particle is drawn onto the second canvas, and then redrawn N times to the visible one. This is less expensive than having to draw each circle N times.

2.6 Three.js Prototype in Two Dimensions

The second prototype uses the same components as the first. *PaintCanvas()* has been replaced with a *render()* method. For each frame, again, a set of step is executed. The particle object now contains its own mesh, which in turn contains its position. The position is then updated according to the same formulae as before. Meshes in three.js consist of a geometry and material. The former holds all data necessary to describe a model, the latter describing its appearance. For these meshes a basic two dimensional circle shape is being used, the material only denoting the color to fill in. While technically being a three-dimensional scene, the camera is fixed at a certain angle and z value, creating the impression of looking at a plane.

2.7 Three.js Prototype in Three Dimensions

At first a third coordinate was added to the particles properties and calculations. It still uses the same formula for motion. The circle mesh has been replaced by a sprite, which as opposed to the circle mesh always faces the camera. It is already implemented this way inside three.js. A sphere-mesh was tested first, but soon turned out to be computationally to expensive. The popular dat.gui has been added in this prototype. To make the bounding box visible, three grids have been added in red, green and blue.

2.8 GPGPU Prototype

Overview

The GPGPU prototype approaches the problem in a more elaborate manner: to speed up the simulation, it uses code executed on the graphics hardware. This way the browser's limitations to the CPU are overcome. It does this in a way described in 2.1. The prototype uses a simulation loop which calls a separate simulator to calculate the new position of the particles and a render

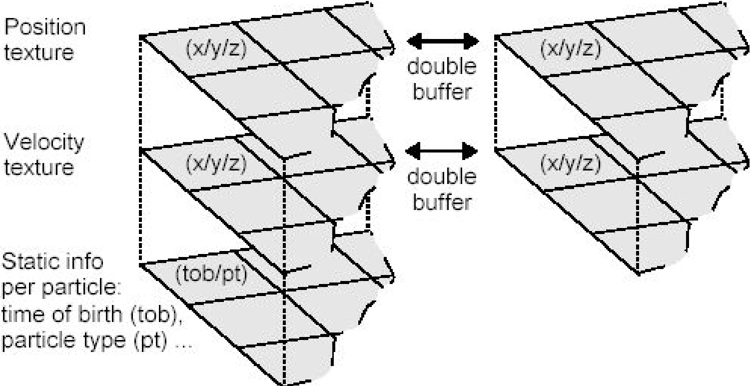

Figure 2.1: Data storage on textures from Latta (2004) [14]. The RGB values are reinterpreted as XYZ-coordinates, but can also be used for other data and passed between two buffers. This buffering is further described in the simulation part of section 2.8.

function for updating the position and rendering the particles to the screen. So for every iteration the simulator updates particle position, velocity and acceleration using the GPU and then passes these calculations to the rendering part, where the actual particles are updated accordingly.

Data Textures

The data is passed between these parts in the form of a texture. Each pixel of the texture usually holds the information for red, green, blue and opacity values.These values are interpreted as the x-, y- and z-coordinate of a particle's position. The opacity value is not used. This is shown in figure [latta]. The velocity in each direction is stored in the same way. This happens for acceleration analogously. Some devices do not support floating-point textures. In that case the values for each pixel would be clamped at a certain point after the comma. As less information can be stored this would result in a very limited space for particles to be in. This is hardly the case with desktop computer and laptop hardware. Our testing devices were also not limited by this and I use 32-bit-floating point values. Additionally reading from textures inside the vertex shader is also sometimes not supported, making it impossible to lookup and to update the particles' position. This process is explained further in [11]. The cited paper is based on OpenGL but the technical aspects are the same for WebGL. Because GPUs usually limit the size of textures possible the textures size is limited to 1024 squared pixels. This makes rendering of more than 1,000,000 particles possible and should be handled well on most modern GPUs.

Simulation Part

In an initialization-step a data-array is generated for each particle's acceleration, velocity and position. Each array is then stored in a texture in the way described in the previous section. Now for every simulation-call, a texture is taken from one frame-buffer object and fed into the fragment shader. Each shader takes one or two textures and the texture's resolution as input via

a list of uniforms and writes the calculated output as the color value to each texture fragment. To look up the correct texture pixel the window-relative coordinates of the current fragment are divided by the texture's resolution with the result used as lookup-vector. The shader then renders the texture using a quad-geometry covering the whole viewport to another frame-buffer object since it is not possible to write the shader program's result to the same texture it read from. In the next step the direction is reversed and the data is read from the second texture, then updated and written to the first. This is called ping-pong or more generally double buffering. The fragments result in a colored pixel on the texture after rendering. Three shaders used are each responsible for updating one texture according to the rules found in section 2.1. Since the fragments of the texture are updated in a parallel way, a performance gain can be observed. The texture containing the updated position is then passed to the render method in the main part of the prototype.

Render Part

The render part makes use of a vertex shader to read the new position from the texture created by the simulator. The particles are modeled by using three.js's *ParticleSystem* which basically consists of an arbitrary number of vertices without fragment shading in-between. Three.js offers a pre-defined vector containing the initial coordinates of the vertices of the particle system in object-space. This vector's x- and y-coordinates are used to look up the correct pixel in the texture for each vertex. The particle system's vertex-coordinates have been initialized with values that correspond to the center of each texture pixel when being used for lookup. After the simulator has been called and a texture containing the updated positions has been received, this texture is then fed into a vertex shader. There each vertex is set to a new position in accordance with the data stored in the texture. This too happens in a parallel way, and aids in enhancing the performance of the simulation as a whole.

Implementation

In the implementation the *simulate()* function seen in figure 2.2 takes care of the ping-pong rendering. The render method is called in every render-loop iteration. After each call a boolean is set to reverse the direction of the rendering. The render functions called each time, take input textures pass them to a shader and then write the output to the second buffer. What happens inside each of these functions can be seen in figure 2.3. The mesh in our case is a plane whose *ShaderMaterial* is being overwritten for using a different shader. The texture is then rendered onto the mesh and read by the fragment shader. Figure 2.4 shows how the data from one texture is read and then being used to update the position texture. The render method which called the simulator, then uses the output to update the vertices. How the vertex shader does this can be seen in figure 2.5

```
[...]
  this.simulate = function(){

    if (pingpong) {

      renderAcceleration(rtVel1, rtAcc1, rtAcc2);

      renderVelocity(rtPos1, rtAcc1, rtVel1, rtVel2);

      renderPosition(rtPos1, rtVel2, rtPos2);

    } else {

      renderAcceleration(rtVel2, rtAcc2, rtAcc1);

      renderVelocity(rtPos2, rtAcc2, rtVel2, rtVel1);

      renderPosition(rtPos2, rtVel1, rtPos1);

        }

    pingpong = !pingpong;

  }
[...]
```

Figure 2.2: The simulation-loop found inside *SimulatorRenderer.js*. At first the acceleration texture is updated (according to formula 2.2) then the new acceleration is combined with the old velocity into the updated velocity texture (formula 2.4). Finally the position is updated and stored in the second position texture (formula 2.4). In the next iteration the direction is reversed. The texture-names have been shortened in this example.

```
function renderPosition(position, velocity, output) {

        mesh.material = positionShader;

        positionShader.uniforms.texturePosition.value = position;

        positionShader.uniforms.textureVelocity.value = velocity;

        renderer.render(scene, camera, output);

        this.currentPosition = output;

    }
```

Figure 2.3: The mesh's (in our case a plane) material is overwritten to load the required shader-program. The uniforms, two textures, are updated and the current position set. The other render methods work analogously.

```
<script id="fragmentShaderPosition" type="x-shader/x-fragment">

      uniform vec2 resolution;
      uniform sampler2D textureVelocity;
      uniform sampler2D texturePosition;

      void main() {


        vec2 uv = gl_FragCoord.xy / resolution.xy;

        vec3 position = texture2D(texturePosition, uv).xyz;
        vec3 velocity = texture2D(textureVelocity, uv).xyz;

        gl_FragColor=vec4(position + velocity, 1.0);

      }

</script>
```

Figure 2.4: Position- and velocity-textures are read, then added together. This is done according to 2.3. The resolution is necessary to calculate a correct lookup vector. The other two shaders are responsible for the remaining calculations. Each pixel now holds the values for the particles' coordinates. The opacity value is set to 1 throughout all shaders.

```
<script id="particleVertexShader" type="x-shader/x-fragment">


    uniform sampler2D lookup;

        void main() {

            vec2 lookupuv = position.xy ;
            vec3 pos = texture2D(lookup, lookupuv).rgb; //
            vec4 mvPosition = modelViewMatrix * vec4(pos, 1.0);//
            gl_PointSize  = 1.0;
            gl_Position = projectionMatrix * mvPosition;

      }

</script>
```

Figure 2.5: Lookup-texture is read out and each vertex' position updated. The necessary view-transformation is also applied. The point size determines the particle's appearance.

CHAPTER 3

Results

In this chapter I will present the results of the prototype evaluation. I measured the FPS for every prototype on two different computers. Furthermore I tested the prototypes across mobile devices and browsers and compared the results. Floating point textures and texture support in vertex shaders are not supported on certain devices. The computers and mobile devices are further described below.

3.1 Testing Environment

The prototypes were tested on the following devices described in tables 3.1 and 3.2.

	Computer A	Computer B
OS	Windows 7 64 Bit	Windows 7 64 Bit
Processor	Intel Core i3 M370 @ 2,4Ghz	Intel Core i5 4200H @ 2,8Ghz
RAM	4,0 GB	8,0 GB
GPU	NVIDIA GeForce 310	NVIDIA GeForce GTX 760M

Table 3.1: Specifications of computers used for testing the prototypes.

While computer B is generally a more powerful machine the computers' most notable difference regarding our use is their graphics cards. The benchmarking tool *3DMark06* [15] gives us the median values of 3274 for computer A and 19221 for computer B. The higher score makes it obvious that computer B is supported by the better graphics hardware.

Mobile Device A	Galaxy Nexus with Android 4.3
Mobile Device B	iPhone with iOS7

Table 3.2: Mobile devices used for testing the prototypes

While both mobile devices (see figure 3.2) did not support WebGL they still could be used for testing fall-back methods. Three.js's *CanvasRenderer* which renders by using the slower Canvas 2D Context API was used to test the three.js prototypes. The Canvas in *CanvasRenderer* refers to the fact that it uses Canvas instead of WebGL to render scenes. Both *WebGLRenderer* and *CanvasRenderer* are embedded in the web page through the canvas element.

3.2 Comparison

FPS were measured by monitoring them over the span of 10 seconds and noting minimum and maximum with constant steps per frame. The difference between the browsers was negligible on computer B and barely noticeable on computer A. The three.js implementations were not successfully tested in Safari 5.1.1 on a MacBook Pro A1278, because WebGL failed to load properly. The tables 3.3 and 3.4 contain the measured FPS for each prototype and a given number of particles N. FPS values above 60 were cut off, as animations are then percepted as smooth and visually appealing. Also higher numbers make no difference in animation on 60Hz monitors. The prototypes can still be compared however by looking at their FPS values for high particle numbers. A hyphen was put where the simulation took above 10 seconds to load or did not move, once loaded. Since the three.js 3D prototype uses textures which can slow the animation down I also tested it by not supplying a texture. Three.js then falls back onto WebGL's *gl_point* to render the particles. This makes it possible to compare it to the GPGPU prototype, which uses the same method. The difference between rendering with and without a texture was only at around 3 to 4 FPS. For this reason the values are not included in the tables. The three.js 3D protoype is able to handle 10000 particles with 7 FPS on computer B however when *gl_point* is used. This means using textures is not feasible for higher particle numbers.

We can clearly see that computer B performs better than computer A. The difference becomes already apparent in the canvas prototype which does not use the GPU for acceleration. The differences become even more evident when comparing the prototypes which are based on WebGL. The advantage of using the GPU can be further emphasized by comparing the GPGPU prototype to the 3D-prototype in three.js. This means that the effort put into programming using the proposed technique really pays off. The mobile devices do not handle three dimensional scenes as complicated as our implementations. The FPS results were found to be approximately the same on both phones. The HTML5 Canvas prototype supported 100 particles at around 30 fps. Around 5 FPS could be reached for the three.js prototypes with canvas fall-back implemented, excluding the GPGPU implementation as a fall-back was not possible. Higher particle numbers resulted in very little movement or difficulties loading the application.

N:	100	1000	10000	100000	1000000
Canvas	35	3	-	-	-
three.js 2D	46-47	3	-	-	-
three.js 3D	>60	27-28	-	-	-
GPGPU	30-38	30-38	30-37	29-31	9-11

Table 3.3: FPS measurement results for computer A for each prototype and a certain number of particles N.

N:	100	1000	10000	100000	1000000
Canvas	>60	7	1	-	-
three.js 2D	>60	7	1	-	-
three.js 3D	>60	>60	-	-	-
GPGPU	>60	>60	>60	>60	24-26

Table 3.4: FPS measurement results for computer B for each prototype and a certain number of particles N.

CHAPTER 4

Conclusion and Future Work

A WebGL based alternative for OpenPixi's Java application was found by exploring the possibilities of a pure HTML5 prototype and two- and three-dimensional prototypes that use the versatile and popular graphics library three.js. Background information on particle-in-cell simulations was also given. A method was implemented which allows to effectively use the GPU to accelerate computations for particle movement by storing the particles' coordinates in a texture. This results in the simulation being able to handle particle numbers of more than one million relatively well. Using the graphics card this way for general computations is called General-purpose computing on graphics processing units or GPGPU. Although programming on the graphics hardware makes it necessary to translate computing problems into graphical terms the performance gain shows that this approach pays off. The prototypes' performance has then been compared in terms of FPS, where the results show that the GPGPU prototype is performing best on desktop computers, surpassing the others immensely. This makes it the most attractive for future development. Since mobility was also a concern, this solution however does not seem to be as attractive for mobile development because mobile devices usually do not support the techniques used in the GPGPU prototype. Instead other solutions should be considered if future work concentrates on mobility. Using three.js's own fallback *CanvasRenderer* which makes no use of WebGL for this results in unsatisfactory performance.

As shown in this work GPGPU inside the browser is not only possible but also very feasible. Since collision detection (between particles and mesh, and particles themselves) is a computationally expensive procedure using the GPU for this as well seems only natural. As computing on the GPU is limited by the need to remodel program flows to graphic processing, one may prefer a simpler three dimensional solution, doing all calculations inside JavaScript. Although JavaScript engines are constantly being developed further it will still be behind a GPU solution performance-wise. Since errors inside the shaders are only printed as general WebGL errors it is hard to debug shader code. Although the JavaScript console of web-browsers usually also print the line number where the WebGL error was thrown, it takes a lot understanding of three.js inner workings to understand what went wrong. I tested two Chrome/Firefox extensions which offer WebGL content tracing.

WebGL inspector [16] offers more features specially made for WebGL, but has not been updated for a while. Notably it lets you keep track of what shaders are running and makes it possible to disable and enable them at run-time. The other, Google's Web Tracing Framework [17], makes it possible to further make connections between JS code and WebGL content. Currently it is still under construction.

Mobile web-browsers do not yet fully support WebGL. Additionally not every mobile device has the hardware necessary to render WebGL scenes. This makes a canvas solution more attractive. Three.js can also render 3D content to canvas without the use of WebGL. This however results in only few FPS, since the CPU now needs to do the graphics calculations. Other libraries provide a better solution to this problem however. One of them, pixi.js [18], is modeled for two dimensional applications, using WebGL where possible and falling back to canvas where not. If a faster web-based OpenPixi application in two dimension is desired, pixi.js would probably be the best choice. OpenPixi also has an OpenCL implementation. It would be possible to use WebCL to transport this functionality to a web-based version. This would probably make it easier to use the GPU, especially when more complex data structures are needed or translating a programming model into graphic terms is not possible.

It should also be mentioned that user interfaces are not limited to existing libraries in three.js. It would be possible to design one's own, by placing static elements at a fixed position in front of the camera. Ray tracing, tracing the path of light through pixels, can be used for determining which elements are clicked. Examples for this can be found inside the examples directory [19].

WebGL remains a highly versatile environment for 3D-graphics applications and as shown in this work is also suitable for performance enhancement through GPGPU. Three.js serves as an adaptable and accessible framework to develop WebGL applications in. Its popularity and the fact that it is constantly updated adds to its appeal.

APPENDIX A

Screenshots

Figures A.1-4 show screenshots of the prototypes discussed in chapter 2. The images result from testing the prototypes on computer B whose specifications can be found in chapter 3. The prototypes were all loaded with 100 particles and no gravitational forces. Figure A.3 and A.4 also include the libraries *stats.js* for FPS-measurement and *dat.gui* which is used for graphical user interfaces.

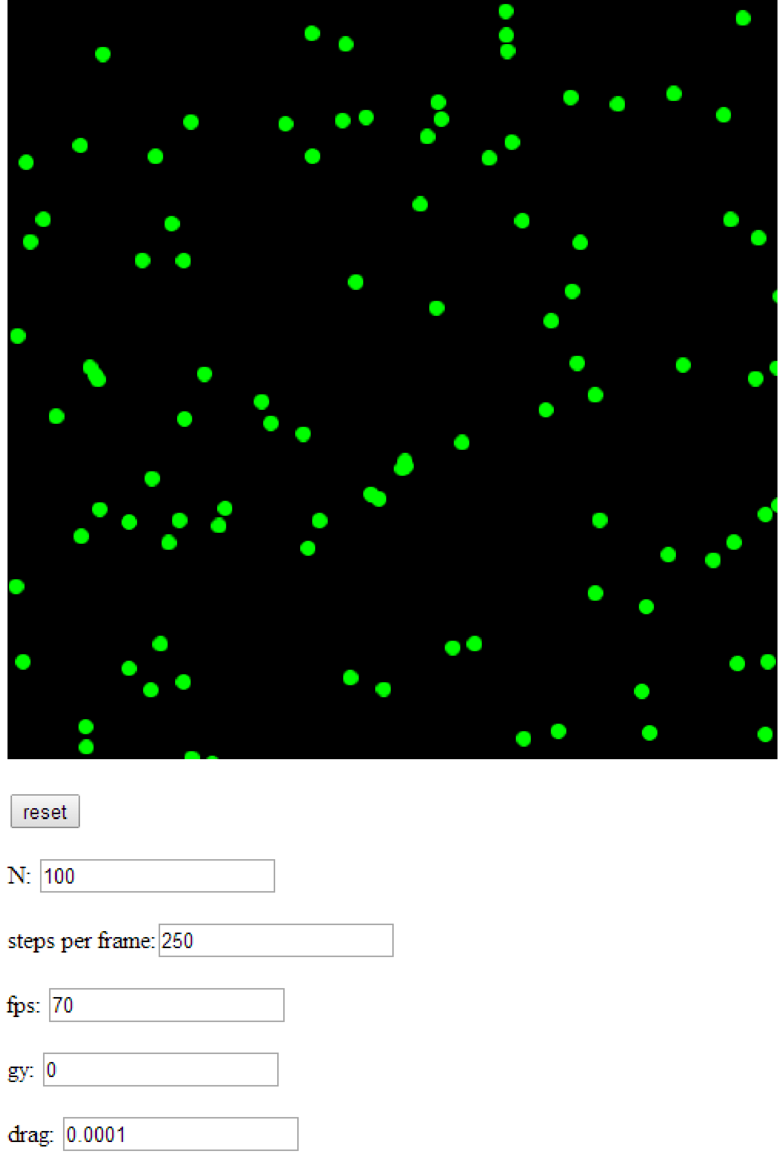

Figure A.1: The HTML5-Canvas Prototype. The user can change number of particles N, steps per frame, gravitational force in y-direction and the drag constant. The FPS are updated each render-loop iteration. To have a change of values take effect the reset button must be clicked.

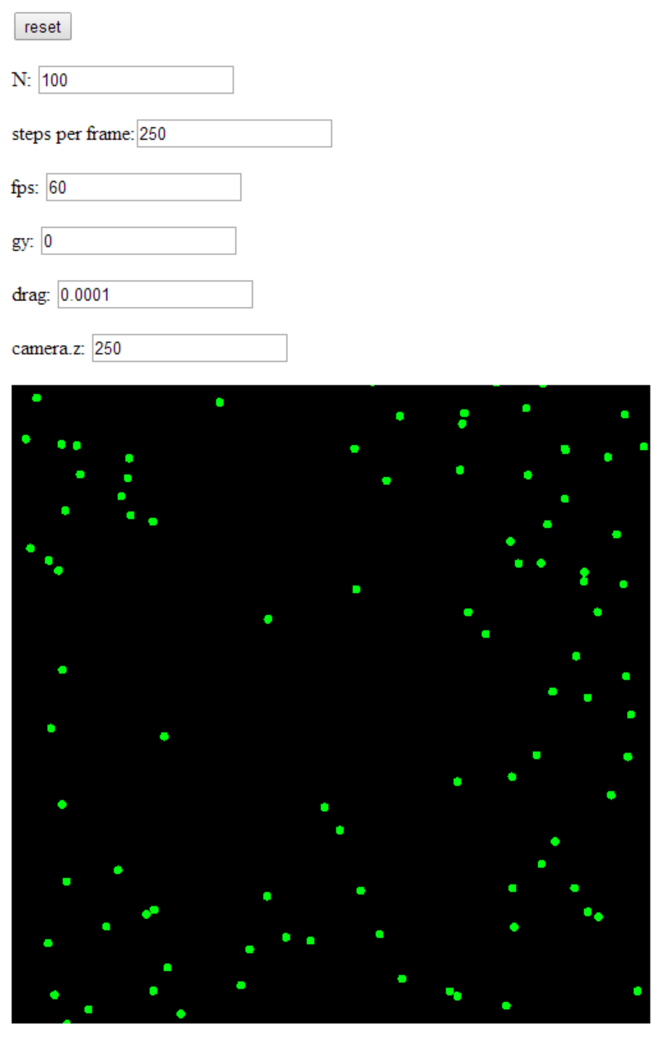

Figure A.2: The two dimensional three.js prototype. It has the same control elements as described in figure A.1 with the addition of the camera's position on the z axis which enables the user to zoom in and out.

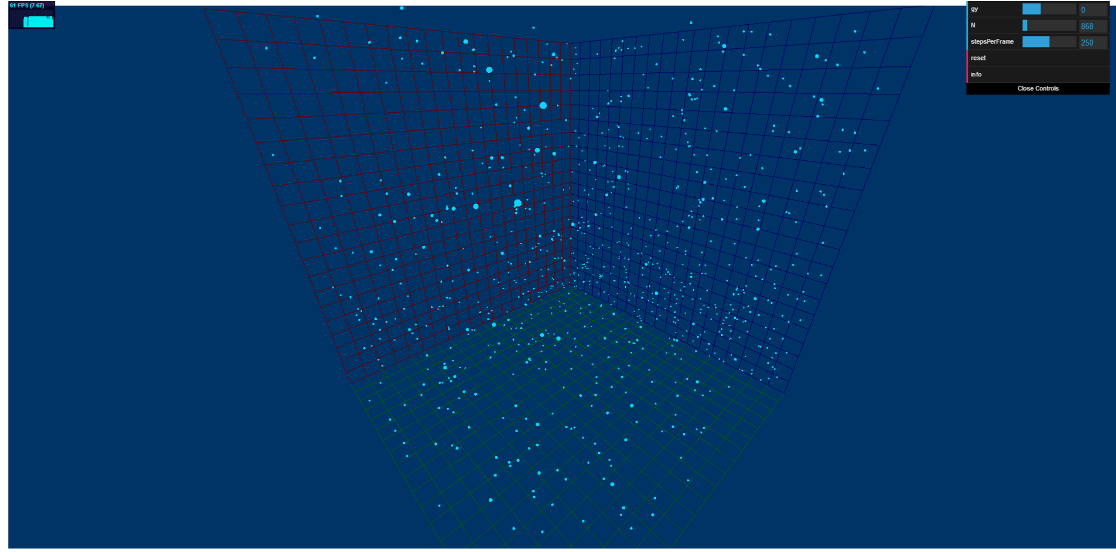

Figure A.3: The three dimensional three.js prototype. The screenshot also includes the *stats.js* seen in the upper left corner. Visible in the upper right is the graphical user interface made with *dat.gui*. The simulation uses a turquoise circle as sprite for the particles appearance.

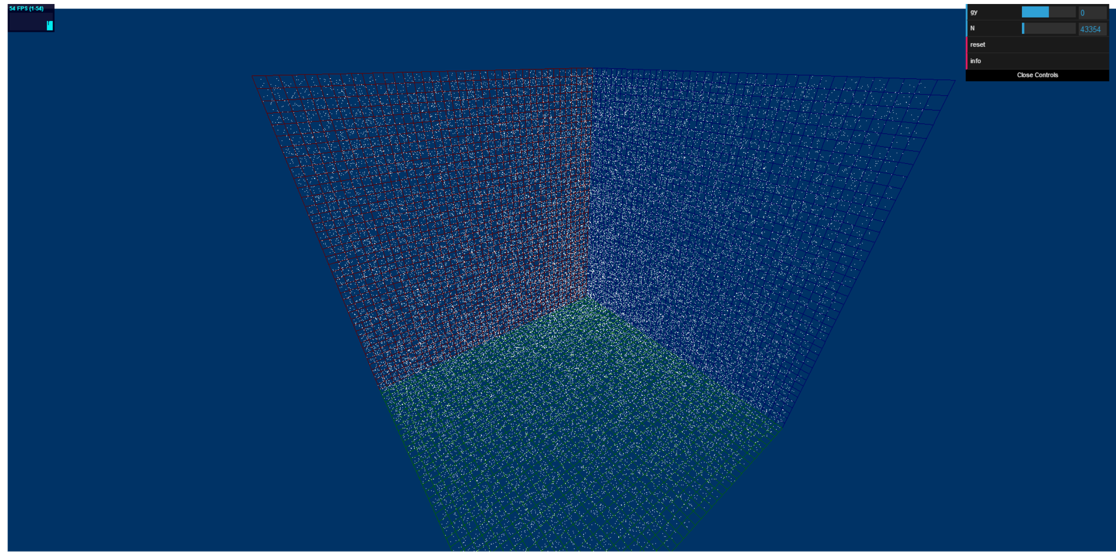

Figure A.4: The GPGPU prototype. In contrast to figure A.3 no sprite is in use and the particles are represented by WebGl's *gl_point*.

Bibliography

[1] J P Verboncoeur. Particle simulation of plasmas: review and advances. *Plasma Physics and Controlled Fusion*, 47(5A), 2005.

[2] G.R. Liu and M.B. Liu. *Smoothed Particle Hydrodynamics*. World Scientific Publishing Co. Pte. Ltd, 2003.

[3] Byrne, Ellison, and Reid. A survey of solar flare phenomena. In *Space Science Reviews, Volume 3, Issue 3, pp.319-341*, 1964.

[4] C K Birdsall and A B Langdon. *Plasma Physics via Computer Simulation*. Taylor & Francis Group, 2005.

[5] Tony Parisi. *Programming 3D Applications with HTML5 and WebGL*. O'Reilly, 2014.

[6] Official three.js website: http://threejs.org/docs/, April 22, 2014.

[7] H Burau, R Widera, W Hoenig, G Juckeand, A Debus, T Kluge, U Schramm, T E Cowan, R Sauerbrey, and M Bussman. Picongpu: A fully relativistic particle-in-cell code for a gpu cluster. *Plasma Science, IEEE Transactions on*, 38(10), 2010.

[8] T Tuckmantel, A Pukhov, J Liljo, and M Hochbruck. Three-dimensional relativistic particle-in-cell hybrid code based on an exponential integrator. *Plasma Science, IEEE Transactions on*, 38(9), 2010.

[9] Xianglong Kong, Michael C. Huang, Chuang Ren, and Viktor K. Decyk. Particle-in-cell simulations with charge-conserving current deposition on graphic processing units. *Journal of Computational Physics*, 2011.

[10] George Stantchev, William Dorland, and Nail Gumerov. Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU. *Journal of Parallel and Distributed Computing*, 2008.

[11] Peter Kipfer, Mark Segal, and Ruediger Westermann. Uberflow: a GPU-based particle engine. In *HWWS '04 Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, 2004.

[12] Daniel V. Schroeder. http://physics.weber.edu/schroeder/software/demos/moleculardynamics.html, April 22, 2014.

[13] Authors github page: https://github.com/zz85, example can be found at: threejs.org/examples/#webgl_gpgpu_birds, April 22, 2014.

[14] Lutz Latta. Building a million particle system. *Game Developers Conference*, 2004.

[15] Official website: http://www.3dmark.com/, April 22, 2014.

[16] Webgl inspector github page: http://benvanik.github.io/webgl-inspector/, April 22, 2014.

[17] Google webgl tracing framework github page http://google.github.io/tracing-framework/, April 22, 2014.

[18] Official website: http://www.pixijs.com/, April 22, 2014.

[19] http://threejs.org/examples, April 22, 2014.