3D Animation of Fluids with Web-Technology

Lars Behl, Tobias Clemens Plankl

RheinMain University of Applied Sciences, Wiesbaden, Germany

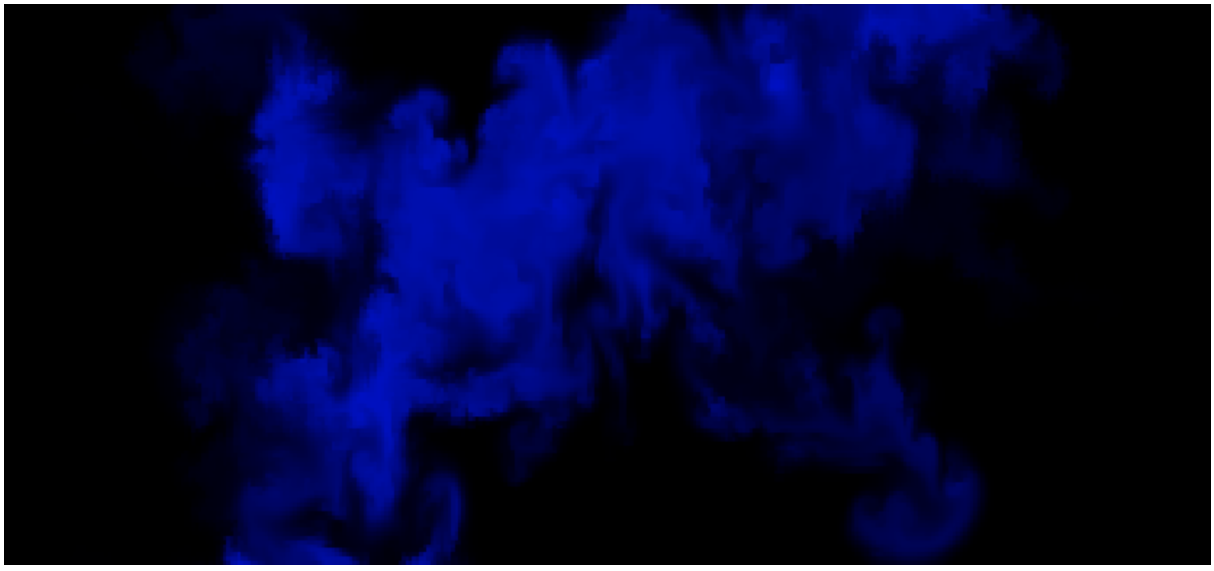

Figure 1: *Image from 2D fluids animations*

Abstract

The main goal of this paper, is to develop an application that animates fluids. A 2D animation was built using web technologies. The main focus of the animation was that it could be executed in real-time and that the user could interact with the fluid. To better understand the development of the software, the basic physics required for the simulation of the fluids is first discussed. Then, already designed systems are used and brought up to date with the help of new web technologies such as Angular, Typescript, WebGl and Three.js. Subsequently, experiments are conducted in which the simulation is checked for performance and hardware requirements. The result of the paper is a 2D animation of fluids that the user can interact with in real time. No installation is necessary for the simulation, because the application can simply be accessed via a website. The paper is dedicated to schoolars, which want to expand their knowledge in the field of animation and understand how fluids are mapped in computers, so that they ensure a realistic simulation.

1. Introduction

Fluids are needed in many 3D animations, they play an important role in movies, games and realistic simulations. Smoke from smoke grenades, water from rivers and other gases or fluids, are all calculated using similar equations. Due to the fact that they can be used in many different areas, they represent an interesting research area and it is worthwhile to have a closer look at the calculations. There are two different uses of simulations. On the one hand there are simulations which model physically accurate fluids. In this case the calculation time plays a subordinate role, as they are typically performed offline.
On the other hand there are the simulations which run in real time. Here it is more important that they look as realistic as possible but still remain a short calculation time. Smaller mathematical errors are accepted.

In this paper the authors focus on simulating fluids in real time.

The calculation should be done as fast as possible but still give nice and realistic looking results. The theoretical basics, which are needed for the calculation will be briefly explained. The most important formulas are the Navier-Stokes Equations which will be discussed later. However, the main focus of this paper is on the implementation of a simulation using newfangled web technologies like Angular, Typescript in combination with Three.js and WebGL.
The simulation is only shown in 2D for now. This is due to the fact that the understanding of fluids should be in the focus. In further projects, this simulation can then be built upon and extended by one dimension to animate fluids in 3D. It simulates the force acting on a fluid and its resulting motion. The user can interact with the fluid in real time and thus change the force and the fluid.

2. Related work

The development of the simulation was based on already existing projects. For the theoretical background, two scientific papers were used. The article [Sta03] deals with a simple and fast implementation of real-time fluids for game engines. At the beginning, the essential physical and mathematical basics, which are needed for the implementation, are briefly discussed. As already mentioned in the introduction, the Navier-Stokes equations are also discussed and how a real time algorithm can be derived from them. Subsequently, the implementation of the mathematical equation in the code and the data structure used are described. Along the way, the article presents step by step implementation in C code.
Another source for this paper is the book [BMF07] which futher explains the theoretical background. In contrast to the first article, the book describes the physical and mathematical principles again in much more detail. More details are considered, such as surface tension. However, the actual implementation via code is not explained.
In addition the book [F*04] and the master thesis [Kon13] were used as additional source for the theoretical background as well as details about the implementation.
The theoretical basics are also explained in this paper in section 3. It is therefore not assumed that the above two scientific sources are known. However, if there is interest in understanding the theory behind the calculations in more detail, it is recommended to have a closer look at these two scientific sources, in the above order.

In addition to the scientific sources that serve the theoretical part, existing projects were also used that helped in the practical implementation of the simulation. It was decided to implement the project using Angular, Typescript and Three.js. So that the authors did not have to worry about implementing the basic structure of the project, the Github project [NgT] was used as a basis. More details why they decided to use the technologies listed above can be found in the section 4.
Another project that served as a template was the Github project [Flu]. This repository contains a simulation of fluids in 2D. The project also refers to the article [Sta03]. Thus, the code was implemented following the same approach. However, since the project was from 2015, it was ported to newer technologies.

3. The physics of fluids

To animate fluids correctly, it must be possible to describe the motion of fluids mathematically. However, the mathematical description of fluids is not very simple and requires a lot of mathematical and physical knowledge.
The basic formula that provides a good model for the representation of fluids in nature is the Navier Stock Equation.

$$\begin{aligned} \frac{\partial u}{\partial t} &= -(u * \nabla)u + \nu\nabla^2 u + f \\ \frac{\partial \rho}{\partial t} &= -(u * \nabla)\rho + \kappa\nabla^2 \rho + S \end{aligned} \tag{1}$$

Equation 1: *The physic of fluids: Navier-Stock-Equations*

However, there is the problem that the Navier Stock Equation requires a lot of computational effort and time. In this paper, the goal is to develop a simulation that can respond to user inputs in real time and animate the motion of a fluid.
Therefore, other implementations and alogrithms are used. These are based on the Navier-Stokes Equation but are faster and more performant. These algorithms have already been implemented in the paper [Sta03] with the help of C and will later be discussed step by step.

It is fundamental to know that the fluid is divided into velocity and density. The velocity is mapped with the help of vectors and indicates with which strength and in which direction the parts of the fluid move. Each point in the space is assigned a vector.
The first formula of the Navier-Stock Equation would now be able to calculate the change of the velocity over an infinitesimal time step from the current state.
The second formula is needed for density and basically does similar things as the first one. During a time step it calculates how the density spreads with relation to the velocity. In contrast to the velocity, the density is visible to us. Density maps the actual particles, be it smoke particles or other objects, which are set in motion by velocity. However, instead of mapping each individual particle and simulating them, Density represents the actual number of particles in a certain space by a percentage. If the density is 1, the space is full of particles and if it is 0, the space is empty.
Thereby the space can be modeled in two different ways

1. Particle based simulation
2. Grid based simulation

Both models are briefly discussed, but the implementation focuses on the latter one.

3.1. Particle based simulation

There are several different methods using particles to simulate fluids. A commonly used one is the *Smoothed Particle Hydrodynamics* method presented in [MCG03] by Muller at al.
It is capable of simulating fluids in which the particles interact with themselves efficiently. To reduce the complexity only the neighboring particles within a certain distance get evaluated for potential interaction. Even thought particle based simulations are typically

faster than gird based simulations they yield a lower result of visual quality [BS10].

3.2. Grid based simulation

For the grid based simulation multiple so called grids are used to store the information, like velocity and density, of the fluid. Thereby each cell in the grid represents a tile of the simulation domain. As this paper discusses the simulation of fluids in 2D, the grids are also twodimensional.
The size per dimension could be interpreted as the resolution of the fluid, because the calculations are run for each cell in the grid. Better resolutions contribute to more convincing results but also take longer to compute and thus require higher powered hardware to retain a framerate eligible for interactive use.

3.3. Evolving density

As already mentioned, the implemented simulation should be able to run in realtime for the user to interact. One of the possible interactions is to add new fluid, to be more precise to increase the density of the fluid, in the affected area. The other possible interaction with the fluid is to add forces which affect the velocity of the fluid. The interaction is somewhat comparable to moving an object through dense smoke. The smoke gets affected by the movement of the object and therefor also starts to move. The area influenced by the user interaction is calculated by the difference of gaussians.
The density of the fluid is not static. On one hand, the fluid diffuses into its neighboring cells, on the other hand the fluid dissolves over time. The implementation currently does not model the diffusion of the fluid.
The two core components of the simulation are the velocity and the density grid. The density of the fluid gets affected by the present velocity field. One challenge now is to calculate the density of the grid cells for each simulation step. A possible solution is to simulate one particle per grid cell. The initial position of the particle is the center of the cell. To calculate the density of the cell, the particle is traced back in time by calculating the movement it has performed when affected by the velocity field. As mentioned before, the density grid describes the density at the center of the cell. In order to now be able to calculate the density at an arbitrary position the value is interpolated between the neighboring cells. This calculation results in the density of the currently inspected grid cell for the current time step.
This step of the simulation is generally known as advection.

3.4. Evolving velocities

Until now, we have looked how the modified algorithms for density work and explained them. Similar to how density spreads, velocity spreads as well. Nevertheless, there are a few peculiarities to consider.
During a time step, the velocity is influenced by the addition of new force and by the self-advection. As with density, velocity also diffuses among neighboring cells, but this was neglected here as well. The addition of new force and the self-advection is identical to the density and can be used here again.
As already announced, there is a special feature to be considered with velocity. If a fluid moves, then a vortex often takes place in reality. This vortex does not represent the velocity so far and therefore density can not be swirled. In order to animate this vortex, the velocity field must be mass conserving. The velocity field consists of a gradient field and a mass conserving field. In order for the velocity field to be massconserving it is needed to substract a gradient field from it. A gradient field is the field which represents the pressure of a fluid. The pressure of a fluid can be imagined like an altitude map. At the places where the pressure is high, there is a mountain and at the places where it is low, there is a valley. The force now acts downward from the mountain in the direction of the valley. This gradient field is calculated with the help of complex mathematics. The first step is to calculate the velocity divergence. The velocity divergence indicates for each time step the change of the forces between the neighbors of the grid cells. With the help of this divergence and the Poisson equation, the pressure can now be calculated. How this is done in the program is explained in more detail in 5.7. For the theoretical background it is recommended to look at [BMF07].

3.5. Vorticity

When calculating the velocities, the conservation of mass of the fluid has already resulted in the formation of vortices. However, these vortices decay very quickly in the simulation. In reality, however, there are many vortices, especially with gases and winds, and these also remain relatively long.
So that the vorticity remains longer a further calculation on the velocity can be accomplished. This calculation is called Vorticity Confinement. The calculation is executed every timestep. Simply explained, the algorithm only search for vortices in the velocity. If it finds turbulences it supports them and tries to amplify them. [sof]

4. Used technologies

As of the COVID pandamic the project should be easily available to different plattforms and machines. Web technologies are a good way of achieving this. One well known framework for working with WebGL is *ThreeJS*. It is a library which supplies easy to use APIs for working in either two or three dimensions [Thr].
After reading the in section 2 mentioned papers and searching for demo projects on the web, the authors decided to implement the simulation in *TypeScript*. TypeScript is an open-source superset of the *JavaScript* language and is developed and maintained by Microsoft Corp. [Typ]. It adds support for statically typing the code, which allows the TypeScript compiler to validate the code on transpilation and expose errors like typing errors before running the code. In addition, more advanced text editors and integrated development environments supply syntax highlighting and error checking in the editor.
To be able to concentrate on the development of the simulation itself, the authors decided to use an Angular 11 project. Thereby the angular compiler handles the compilation and packaging of the application [Ang].
The base of the project was cloned from [NgT] which already integrates *ThreeJS* into *Angular*. The animation itself is run outside of the Angular context to prevent unnecessary change detection and re-rendering of the canvas element.

4.1. Real time simulation

For the user to interact with the fluids in realtime, the simulation of the fluids itself has to be in realtime. The algorithms used for the simulation could be run in parallel. As browsers do not provide an API for multithreading, running the simulation on the CPU would be limited to a single thread. However it is possible to run multiple threads at once when using *WebWorkers* to offload work from the main thread. Compared to multiple threads in traditional applications, WebWorkers do not share memory and therefor the shared data would have to be copied to every worker [Weba].
Fortunatly there is another possibility to run the simulation. The GPU is highly parallelized hardware. In the graphics pipeline the *Fragment Shader* needs to be implemented by the developers and is calculated for each pixel candidate. A pixel candidate is a fragment, where one or more parts of geometry is located [Fraa].
This parallelization of the *Fragment Shader* could be used to run the algorithms needed for the simulation in parallel for each cell of the grid. This potentially increases the performance of the simulation and enables the possibility of realtime interaction of the user.
Another way to leverage the power of the GPU is to implement *Compute Shaders*. These are shader programs that are specialized for cumputational work. As of January 2021, Compute Shaders are part of *WebGL 2.0 Compute* which is still in specification and has only experimental support in some browsers [Webb].

5. Implementation

As previously mentioned the application is implemented using *Angular 11* as the basis. For the simulation and animation *ThreeJS* is used as an easy to use graphics library.
The following chapters describe the general structure of the project as well as the structure of the code.

5.1. Animations in the browser

In general a browser exposes an API to render an animation to a canvas element. This method is called **requestAnimationFrame()** and is part of the window object. It expects a callback function that gets called before the next repaint of the canvas element. Each call to requestAnimationFrame() does request a single repaint. In order to run an animation for multiple frames, the callback of requestAnimationFrame has to call requestAnimationFrame again [req].
Angular 11 is a sophisticated framework for developing single page applications. It internally uses mechanisms to detect wether the content of currently active components has changed or not. When a change is detected, the component and all its child components get re-rendered. Components are the basic building blocks of UI elements in angular. A component consists of an *HTML* file containing the possibly visible elements, an *TypeScript* file handling the logic and a stylesheet. The angular runtime handles the injection of visible components into the dom [Ang].
In order to prevent change detection for the canvas that is used to render the simulation, the animation run outside of the angular context.

5.2. Porting code from JavaScript to TypeScript

The in section 2 mentioned demo application is the basis of the application the authors implemented. As the new implementation is using *TypeScript* and the other one uses *JavaScript*, the code needed to be ported. Because of the similarities of these two languages, the porting process itself was not as complicated as expected.
The orignially used version of *ThreeJS* exposed a slightly different API than the current one. Tracking down the changes in the API and the associated changes in the code was a time consuming process. In addition, debugging shader programs in *WebGL* is very difficult as it is not possible to print information to a command line. An extension of WebGL is available to debug shaders, however this extension could only be used to retrieve the translated shader program [Webc], which is already present in clear text.
While porting the code, the source application was stripped from all unused code and UI-Elements. The functionality provided by the UI, like changing the dissipation factor of the fluid, was substitued by static values for those factors. In later revisions it is possible to add a user interface offering comparable functionality. Stripping the application to a minumum viable product reduced the number for sources of error and therefor potentially minimized the need of debugging.
While porting the application brings insight on how to implement a fluid simulation, it is still needed to understand what the code actually does. A general understanding of how the graphics pipeline works and the implemented algorithms for simulating fluids is key to understand how the application achieves realtime and interactive animation of fluids. The in section 2 mentioned papers were the entrypoints for researching realtime fluid simulation.

5.3. Splitting up the code

To run one of the calculations like *advect* or *splat*, the application needs to:

1. Store and load information about the current state (e.g. velocity)
2. Setup the shader and pass all values needed for the calculation
3. Run the calculation which are performed on the graphicscard

For each of those steps, the application contains one source file handling that specific part.
To store the information about the current state of the simulation, the so called *slab* class was implemented. Instead of using twodimensional arrays to store the grid, the application uses **WebGLRenderTarget**s. This class is provided by *ThreeJS* and can be used as a target of the rendering pipeline. What this means is, that instead of rendering to a canvas, the result of the pipeline is stored in the render target. To instantiate a **WebGLRenderTarget** one needs to specify the width and height of the target, as well as some additional options.
One slab contains two of these render targets, one containing the information of the previously performed calculation on this specific slab and one the results for the current calculation.
The next step is to set up the shader program running on the graphicscard and to pass the information needed to perform the calculation. Therefor the application contains a class called **SlabopBase**. It is the base class all specialized operations, like advect, extend. Internally it can store five objects provided by *ThreeJS*:

1. A **PlaneBufferGeometry**
2. A **ShaderMaterial**
3. A **Mesh**
4. A **OrthographicCamera**
5. A **Scene**

All of these objects get instantiate when the constructor is called. The constructor expects a string containing the fragment shader program, an object containing the type and value of the uniform variables and a Grid, which is an object containing the size and the scale of the grid used for simulation. The **PlaneBufferGeomerty** and **ShaderMaterial** are used to create the actual **Mesh** present in the scene. This is important, as the rasterization step only produces fragments for pixels covered by a primitive [frab]. This Mesh is used to cover the whole scene with primitives.
All classes extending the **SlabopBase** can store additional information needed for the calculation. For example, the **Advect** slabop additionaly stores information about the grid size, the timestep and the dissipation rate.
The extending classes need to initialize the uniforms present in their specific fragment shader, which get passed to the **SlabopBase** in the super call of the constructor. Additionaly the classes all implement a method called "compute" which sets the correct scene and fragment shader program, passes all needed values to the graphicscard and sets the correct render target before "rendering" respectively calculating the result.

The third end last part of the code related to performing the simulation is the fragment shader used for the calculation. A fragment shader is a program implemented in the **OpenGL Shading Language**. As previously mentioned it is part of the rendering pipeline and gets called for each fragment generated in the rasterization step. Several values are passed to the fragment shader from the rendering pipeline. In addition, it is possible to pass values from the CPU side of the application to the shader. In this case the information provided by the slabops.
The implemented fragment shaders are used to run the needed calculations in parallel for each grid cell, thus enabling the simulation to run in realtime.

5.4. Rendering the simulated fluid

The last step of the animation is to render the result of the simulation step to the canvas element in order to visualize the fluid. Therefor the fragment shader **displayscalar** is used. The density grid, the color of the fluid and a bias value get passed to the shader.

```
uniform sampler2D read;

uniform vec3 bias;
uniform vec3 scale;

varying vec2 texCoord;

void main()
{
    // get the FragColor of the the texture
    gl_FragColor = vec4(
        bias + scale *
        texture2D(read, texCoord).xxx,
        1.0
    );
}
```

Listing 1: *Fragment shader rendering the simulated fluid*

As shown in listing 1 the program itself extracts the dentsity of the fluid for a fragment, multiplies it with the color to adapt the opacity according to the density and adds up the bias. As the bias is set to zero for the current implementation, it does not have any effects on the result.

5.5. Implementing the user interaction

As already mentioned, the user can interact and control the simulation. He has the possibility to add density. This density is visible for the user. However, without a force acting on this density, the user does not see any effect yet. Therefore he also has the possibility to add velocity. As mentioned earlier, this velocity influences the density and thus the desired effect is created. The user can perform both interactions with the mouse. If he keeps the right mouse button pressed it is possible to add density to the density slab. If the left mouse button is held down, velocity is added to the velocity slab. The base for the implementation is to intercept the data of the mouse. For this purpose a separate class "Mouse" was implemented. The mouse class contains functions that are responsible for mouse interaction. This includes the movement of the mouse over the canvas or a click on the canvas. A click actually only changes an indicator variable, which is used to determine whether density or velocity should be added. When tracking the movement, more is captured, both the current point and the direction of movement of the mouse is stored for a short time. Since all the data of the mouse is now available, this data must now be used to add Density or Velocity to the respective slab. For this purpose, another class and a shader called "Splat" was implemented. The class Splat is called in the function addForces(). Depending what kind of mouse button is pressed, Splat calculates destiny or velocity. Thereby the calculations differ only slightly. The calculation of the velocity field requires the movement direction of the mouse, whereas the calculation of the destiny field requires a value of how much destiny should be added to the clicked cells.

```
// velocity or density
uniform sampler2D read;
uniform vec2 gridSize;
// force or source
uniform vec3 color;
// grid cell the mouse points to
uniform vec2 point;
// goal slab
uniform float radius;


float gauss(vec2 p, float r)
{
    return exp(-dot(p, p) / r);
}

void main()
{
    vec2 uv = gl_FragCoord.xy / gridSize.xy;
    vec3 base = texture2D(read, uv).xyz;
```

```
    vec2 coord = point.xy - gl_FragCoord.xy;
    vec3 splat =
        color *
        gauss(coord, gridSize.x * radius);
    gl_FragColor = vec4(base + splat, 1.0);
}
```

Listing 2: *Fragment shader used to interact with the fluid*

The actual calculation as seen in listing 2 is then performed again on a shader on the graphics card. The current force of the fragment is queried, the vector between the mouse position and the fragment to be observed is evaluated and the new effects of the click are calculated using Gauss. Finally, these are added to the current state.

5.6. Implementing advect

As already mentioned in the theory part, for the movement of the velocities and the density a function is needed, which calculates the current slabs from the slabs of the previous time step. This function is called "Advect" and is implemented as follows. Advect is splitted in a class and a fragmentshader. The actual calculation takes place again outsourced in the shader. As already explained in section 3.3 each fragment is traced back to the previous position. The actual calculation is then performed on this point. The force of the slab is dissipate and an interpolation is performed between the neighbors of the calculated fragment. With the help of this interpolation the influence of the neighbor fragments on the actual fragment is calculated. All diagonally lying fragments are considered as neighbors in this implication. Listing 3 shows the implementation of the adevection as a fragment shader.

```
gl_FragColor = vec4(
        dissipation *
        bilerp(advected, p), 0.0, 1.0
        );

vec2 bilerp(sampler2D d, vec2 p){

    vec4 ij;

    ij.xy = floor(p - 0.5) + 0.5;
    ij.zw = ij.xy + 1.0;

    vec4 uv = ij / gridSize.xyxy;

    vec2 d11 = texture2D(d, uv.xy).xy;
    vec2 d21 = texture2D(d, uv.zy).xy;
    vec2 d12 = texture2D(d, uv.xw).xy;
    vec2 d22 = texture2D(d, uv.zw).xy;

    vec2 a = p - ij.xy;

    return mix(
        mix(d11, d21, a.x),
        mix(d12, d22, a.x),
        a.y
    );
}
```

Listing 3: *Fragment shader used for advection*

5.7. Implementing projection

In the section 3.4 it was already discussed that during the calculation of the velocities further calculations have to be taken care of, so that the fluid remains mass-conserving. These calculations are performed in the project function. In order for a fluid to be mass conserving, the gradient field must be subtracted from the actual field, as mentioned before. Therefore, the calculation of the gradient field must be performed first. To do this, the process looks like this:

1. Calculate the velocity divergence
2. Calculate the gradient field (pressure field) using the Poisson equation and the previously calculated velocity divergence.
3. Subtract the gradient field from the actual velocity field

To calculate the gradient field, the velocity divergence is needed. The divergence of a vector field indicates whether more velocity is pointing in or out of an infinitesimal piece. For the calculation again a shader and a class named 'Divergence" was created. The shader gets the current velocity field and then calculates the divergence fragment by fragment and stores it in a new slab. For the calculation the 4 neighbors of the actual fragment are looked at. From the left and right neighbors we are only interested in the horizontal force, since these only have an influence on the current fragment. Exactly the same applies to the lower and the upper neighbor, but here we are interested in the vertical force. With the help of these forces we can calculate the divergence of the fragment.
After calculating the divergence, the gradient field can now be calculated with the Poisson equation. The calculation is also performed in a shader with a class named "Jacobi". For each fragment, the previous pressure of the neighbors is queried. For the actual fragment under consideration, the pressure is not considered, but the previously calculated divergence. With the help of these values, the Poisson equation is then performed. Interesting is that the complete calculation is performed 50 times per fragment. For the theoretical background referred again to [BMF07] and [Sta03].
The last step is the easiest one. Only the previously calculated gradient field must now be subtracted from the actual velocity field. Again, this is done fragment by fragment using a shader and a class called "Gradient". The four neighbors of the fragment are considered here as well and the pressure of the actual fragment is calculated with the help of these. This calculated value is then subtracted from the actual velocity field and the new one is calculated.

5.8. Implementing boundary conditions

Boundary is another slabop and it is used to keep the volume of the fluid consistent by not letting it escape a bounding box. To achieve this, four lines, which are geometric objects, are created. These lines represent the top, bottom, left and right boundary. These lines occupy the most outer columns, respectively rows of the animation grid. The advantage of this approach is, that now it is fairly easy to determine which grid cells have to be effected by the boundary check. In fact, the graphics pipeline itself already solves this problem, as the fragment shader is only executed for each pixel candidate that is covered by a primitive. These primitives are the previously created lines.

```
uniform sampler2D read;
uniform vec2 gridSize;
uniform vec2 gridOffset;
uniform float scale;

void main()
{
    vec2 uv =
        (gl_FragCoord.xy + gridOffset.xy)
        / gridSize.xy;
    gl_FragColor = vec4(
        scale * texture2D(read, uv).xyz,
        1.0
    );
}
```

Listing 4: *Fragment shader for calculating boundary conditions*

As listing 4 shows, the fragment shader is fairly simple. It only needs to access the position of the current fragment, normalize it and read the value out of the velocity grid. The final step is to invert that velocity so that the fluid gets reflected by the bounding box.
One potential point of failure is, that if the fluid is travelling in parallel to a border, both components of the velocity get inverted. This means, that even if the fluid would never actually collide with the border, it is treated like it would.

5.9. Implementing the solver

The solver is the main class needed for simulating a fluid. All the previously mentioned mechanisms like *advect* or *project* are methods of the given class. The solver initializes all needed slabs and slabops. The most important method of the class is **step()**. When called, the solver performs one complete cycle of simulating the fluid.
In the beginning of a step, the advection of the velocity is calculated and the boundary conditions are checked. Afterwards the density is advected. This is followed by the addition of forces the user wants the fluid to get influenced by. Next the vorticity is calculated and added to the velocity. The last step before calling the **project()** method is to, once again, check the boundary conditions and change the velocity of the fluid if necessary.
By repeatedly calling the solvers **step()** method for each requested animation frame, the fluid is simulated.

5.10. Implementing vorticity

The actual implementation of the simulation would now be complete. However, since the simulation looks better when vortices are taken into account, the "Vorticity" and "Vorticity Confinement" classes are used to try to support and maintain vorticity rotation. The calculation explained in the following take place in the solver step between the advect function and the project function.

In the class and the shader "Vorticity" the vortices must be recognized for the first time. The velocity slab is passed to the shader. On this shader it is checked fragment by fragment whether a vortex near the point and to what extent this fragment is affected by it. To check this mathematically the neighbors of the fragment are also taken into consideration. On the basis of these it is recognizable whether the forces around the fragment are vortex-like. Subsequently, the effect of the forces forming a vortex are calculated on the current point are calculated and stored there in a new slab of the vorticity force.

After calculating the vorticity for each grid cell, the influence of neighboring vortices on the current grid cell is computed by **vorticityConfinement**. For this operation the project contains, as for all other slabops, a fragment shader and a complementing class.
For each cell the force it is experiencing by the neighboring cells is computed. This is accomplished by extracting the previously calculated vorticity for all for neighboring cells and the current cell itself. The next step is to subtract the absolute value of the vorticity of the bottom from the top cell and the left from the right cell. This results in a twodimensional vector containing the force in x and y direction. The vector is then multiplied with the inverse square root of the length of itself, a static curl factor and the voritcity in the current cell.
The last step is to multiply the force by the current timestep and adding this value to the current velocity of the cell. This newly updated value is then stored in the velocity slab.

6. Experiments

After the actual simulation is implemented, you can experiment with it. Due to the fact that there are many different constants and variables that can be adjusted, they can be changed and the impact on the simulation can be illustrated.
The system the experiments were performed on contains the following hardware:

- Intel Core i5-7300U
- 8GB DDR3-1866MHz
- Intel HD Graphics 620

The experiments were repeated on a machine with significantly more computational power:

- Intel Core i7-10700K
- 16GB DDR4-3200MHz
- GeForce RTX 3070

It is important to note that the framerate is limited to the monitors framerate. The refreshrate for the more powerfull machine is 144 Hz and 60 Hz for the less powerfull.

6.1. Change resolution of grid size

One of the parameters that was explored during the experiments is the grid size. It describes the resolution of the simulation domain and has a significant impact on how convincing the animation is. By increasing the resolution the computational effort does also increase. While the first machine achieved an interactive framerate of about 60 FPS with a grid size of 512 by 256. Increasing the resolution to 800x600 the framerate dropped below 20. Table 1 shows all results of the performed experiments on both machines.

6.2. Change iteration counter at the Poisson-Equation

In the function "project()" the pressure of the fluid is calculated using the Poisson Equation. The equation is executed several times to

Grid size	Framerate i5	Framerate i7
512x256	~60 FPS	~144 FPS
800x600	~16 FPS	~144 FPS
1920x1080	~4 FPS	~70 FPS

Table 1: *Results for the experiments regarding the grid size*

achieve better results. If the equation would not be executed several times, the fluid would not be correctly mass conserving. As a standard value 50 iterations were chosen, thereby the interplay of performance and real simulation was exhausted in the best possible way. However, if the number of iterations were increased, the simulation would simulate more realistically, but the performance of the simulation would suffer enormously and the FPS would drop. The reason for this is that now each frame needs longer to be calculated. Table 2 shows the results of the performed experiments.

Poisson equation iterations	Framerate i5	Framerate i7
50	~60 FPS	~144 FPS
500	~8 FPS	~144 FPS
1000	~6 FPS	~71 FPS

Table 2: *Results for the experiments regarding the number of iterations the poisson equation is performed*

7. Conclusions

As the application shows it is possible to implement realtime fluid simulations with web technologies. The achieved results are convincing. However there are several limitations present. First of all the resolution of the fluid is a limiting factor as with higher resolution the results may look better but the needed computational power also increases. In addition the implemented algorithms result in visually pleasing simulations, but the physical properties are only approximated. Also it is important to note that normally a scene in a realtime graphics application like video games consists of more than just a fluid simulation. Other objects like characters and the scenery also have to be simulated and rendered. Tweeking the parameters like the grid size and iteration count for the poisson equation is mandatory to achieve a good tradeoff between computational expense and visually pleasing simulations.
In later iterations a user interface for editing parameters like the "Curl Factor" or the number of iterations used to calculate the pressure of the fluid may be added. Currently fluids are only simulated in a twodimensional domain. This application can be used as a starting point for further projects implementing realtime fluid simulation in three dimensions.

References

[Ang] Angular. Accessed: 27.01.2021, 11:10 AM. URL: https://angular.io/.

[BMF07] BRIDSON R., MÜLLER-FISCHER M.: Fluid simulation: Siggraph 2007 course notes video files associated with this course are available from the citation page. In *ACM SIGGRAPH 2007 courses*. 2007, pp. 1–81.

[BS10] BRALEY C., SANDU A.: Fluid simulation for computer graphics: A tutorial in grid based and particle based methods. *Virginia Tech, Blacksburg* (2010).

[F*04] FERNANDO R., ET AL.: *GPU gems: programming techniques, tips, and tricks for real-time graphics*, vol. 590. Addison-Wesley Reading, 2004.

[Flu] Fluids-2d. Accessed: 27.01.2020, 3:34 PM. URL: https://github.com/mharrys/fluids-2d.

[Fraa] Fragment. Accessed: 27.01.2020, 12:00 PM. URL: https://www.khronos.org/opengl/wiki/Fragment.

[frab] Fragment shader. Accessed: 06.02.2021, 12:20 PM. URL: https://www.khronos.org/opengl/wiki/Fragment_Shader#:~:text=A%20Fragment%20Shader%20is%20the,a%20%22fragment%22%20is%20generated.

[Kon13] KONTAXIS C.: *Fluid simulation for computer graphics*. Master's thesis, 2013.

[MCG03] M"ULLER M., CHARYPAR D., GROSS M. H.: Particle-based fluid simulation for interactive applications. In *Symposium on Computer animation* (2003), pp. 154–159.

[NgT] ng-three-template. Accessed: 27.01.2021, 11:20 AM. URL: https://github.com/JohnnyDevNull/ng-three-template.

[req] Window.requestanimationframe. Accessed: 03.02.2021, 12:50 PM. URL: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame.

[sof] Vorticity confinement for eulerian fluid simulations. Accessed: 03.02.2021, 11:30 AM. URL: https://softologyblog.wordpress.com/2019/03/13/vorticity-confinement-for-eulerian-fluid-simulations/.

[Sta03] STAM J.: Real-time fluid dynamics for games. In *Proceedings of the game developer conference* (2003), vol. 18, p. 25.

[Thr] Threejs. https://github.com/mrdoob/three.js/. Accessed: 27.01.2021, 10:45 AM. URL: https://github.com/mrdoob/three.js/.

[Typ] Typescript. Accessed: 27.01.2021, 11:00 AM. URL: https://www.typescriptlang.org/.

[Weba] Web workers api. Accessed: 27.01.2021, 11:20 AM. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.

[Webb] Webgl 2.0 compute. Accessed: 27.01.2020, 12:20 PM. URL: https://www.khronos.org/registry/webgl/specs/latest/2.0-compute/.

[Webc] Webgl_debug_shaders. Accessed: 27.01.2021, 1:30 PM. URL: https://developer.mozilla.org/en-US/docs/Web/API/WEBGL_debug_shaders.