# Two-Dimensional Fluid/Smoke Simulation in WebGL using the OpenGL Shading Language

David Vajcenfeld
University of Toronto
Toronto, ON, Canada
david.vajcenfeld@mail.utoronto.ca

## Abstract

This paper describes an implementation of fluid and smoke simulation in WebGL, a web-based graphics library based on OpenGL. The Navier-Stokes equations are used to govern fluid, and calculated using separate shader programs on the GPU. The Poisson pressure equation is estimated using the iterative Jacobi method.

***CCS Concepts:*** • **Computing methodologies → Physical simulation**.

***Keywords:*** Fluid simulation, smoke simulation, WebGL, GLSL, shaders.

## 1 Introduction

This paper describes an implementation of fluid and smoke simulation in WebGL. The Web Graphics Library, commonly referred to as WebGL, is a JavaScript API for "high-performance interactive 2D and 3D graphics" [1] that can easily be viewed in modern web browsers. It is based on OpenGL ES 2.0 and makes use of the OpenGL ES Shading Language (GLSL ES).

In this project, a fluid and smoke simulation in two dimensions was implemented, by solving the Navier-Stokes equations.

## 2 Related Work

The *Fluid Simulation* course notes from SIGGRAPH go into detail on the Navier-Stokes equations, how they are split into components for computation, and how the pressure equation is solved in both two and three dimensions. [3]

The *GPU Gems* book, and more specifically chapter 38 *Fast Fluid Dynamics Simulation on the GPU*, goes into the details on fluid simulation in two dimensions using shaders. This project primarily attempts to implement this in WebGL. [5]

The paper *Smoke Simulation using WebGL Technology* covers generally the same content as the chapter in the *GPU Gems*, but implemented in WebGL and in three dimensions. Most importantly, it provides details on how the approach is extended to three dimensions. [2]

Finally, the *WebGL Fluid Simulation* code repository by Pavel Dobryakov implements a similar smoke-like fluid simulation in WebGL. This code repository was used initially as a reference as a basis on how WebGL was set up and interacted with the shaders, as the author if this project was greatly unfamiliar with the works of WebGL. Similarly, the idea and implementation of "splattering" fluid particles by mouse click-and-drag was taken from here. [4]

## 3 Method

The Navier-Stokes equations consist of a set of partial differential equations (PDEs) that govern the motion of simulated viscous fluids and smoke. [3]

In WebGL, we implement this as a set of WebGL programs written in GLSL and executed on the GPU. Each program consists shaders which operate on the level of a single pixel or texel, respectively. A texel is a texture pixel, the smallest component of a texture map, which we use to represent fluid particles in our simulation. This paper will refer to "texel" and "pixel" interchangeably.

We solve the Navier-Stokes equations analytically by splitting them into a series of computational steps by the Helmholtz-Hodge Decomposition. These are then implemented as individual shader programs, and their values computed onto a set of framebuffers which represent various quantities. Framebuffers are matrices that contain a four-vector for each texel in our simulation.

### 3.1 Advection Program

This program handles the advecting (i.e. transporting) of the fluid particles by applying their velocities.

Traditionally this would be done by computing the position of a particle after the time step as in Equation 1, and moving it appropriately.

$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \Delta t\boldsymbol{v}(t) \tag{1}$$

Instead, in the fragment shader of the advection program we trace the position of each texel to its position in the previous time step, and then copy the values over from there, as in

*CSC417 Physics-Based Animation, 2020, Toronto, ON, Canada*
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Equation 2. In Equation 2, $c(\boldsymbol{x})$ represents either the velocity or the density at position $\boldsymbol{x}$, as this advection program is executed on both the velocity and density framebuffers.

$$c(\boldsymbol{x}) = c(\boldsymbol{x} - \Delta t \boldsymbol{v}(t)) \tag{2}$$

### 3.2 Divergence Program

This program calculates the divergence of the vector field $\nabla \cdot \boldsymbol{v}$ as the first step in solving the pressure equation, by using the discrete equation in Equation 3.

$$\nabla \cdot \boldsymbol{v} = \frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \tag{3}$$

In Equation 3, $\Delta x$ and $\Delta y$ represent the "space step" in the two dimensions. In our case this is the same in both dimensions, so we have $\Delta x = \Delta y$.

### 3.3 External Forces Program

This program just applies the effects of external forces that act on the liquid, such as gravity or user input, onto the velocities of the particles, using , where $\boldsymbol{g}$ is the forces vector.

$$\boldsymbol{v}(t + \Delta t) = \boldsymbol{v}(t) + \Delta t \boldsymbol{g} \tag{4}$$

### 3.4 Jacobi Method Program

This program is a simple iterative solver for the pressure equations. This method, the Jacobi method, converges slowly, but is fast to execute on the GPU. Consequently, we use around forty iterations to arrive at a sufficiently-accurate estimate.

The update-step equation used is Equation 5, where $x$ is the pressure $p$, $b$ is the divergence $\nabla \cdot \boldsymbol{v}$, and the parameters $\alpha = -(\Delta x)^2$ and $\beta = 4$. Also, $k$ represents the iteration.

$$x_{i,j}^{(k+1)} = \frac{1}{\beta}\left(x_{i-1,j}^{(k)} + x_{i+1,j}^{(k)} + x_{i,j-1}^{(k)} + x_{i,j+1}^{(k)} + \alpha b_{i,j}\right) \tag{5}$$

### 3.5 Gradient Subtraction Program

This program corrects the divergence of the velocity by subtracting the gradient of the pressure, creating the divergence-free vector field $\boldsymbol{u}$, as seen in Equation 6.

$$\boldsymbol{u} = \boldsymbol{v} - \nabla \boldsymbol{p} \tag{6}$$

### 3.6 Boundary Condition Program

Finally, an important part of fluid simulation is to enforce conditions on the boundaries. Here we update the velocities and pressures to satisfy the Neumann boundary conditions, enforcing that the rate-of-change normal to the boundary is zero.

Equation 7 and Equation 8 show the velocity and pressure updates, respectively, for particles on the left boundary. The updates apply similarly to other the other boundaries.

$$\frac{\boldsymbol{u}_{0,j} + \boldsymbol{u}_{1,j}}{2} = 0 \implies \boldsymbol{u}_{0,j} = -\boldsymbol{u}_{1,j} \tag{7}$$

$$\frac{\boldsymbol{p}_{1,j} - \boldsymbol{p}_{0,j}}{\Delta x} = 0 \implies \boldsymbol{p}_{0,j} = \boldsymbol{p}_{1,j} \tag{8}$$

Below, in Equation 9, we see the update logic performed by this program.

$$x_{i,j} \leftarrow \begin{cases} s \cdot x_{1,j} & \text{if } i = 0 \text{ left boundary} \\ s \cdot x_{W-2,j} & \text{if } i = W-1 \text{ right boundary} \\ s \cdot x_{i,1} & \text{if } j = 0 \text{ top boundary} \\ s \cdot x_{i,H-2} & \text{if } j = H-1 \text{ bottom boundary} \\ x_{i,j} & \text{otherwise} \end{cases} \tag{9}$$

## 4 Results

As a result, we see a colorful simulation of a continuous fluid swirling around in the WebGL canvas, as seen in Figure 1.

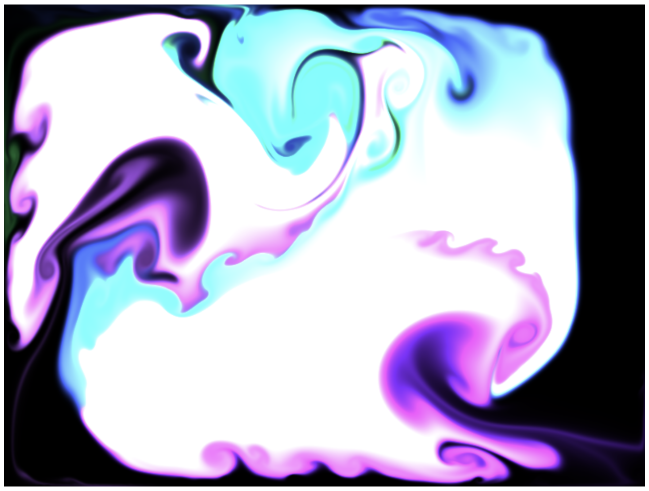

**Figure 1.** A screenshot of the resulting simulation.

There are some issues, that could be fixed with a bit more work. Firstly, there is an issues where the edges actually seem porous and fluid seems to "seep" through them. This is most likely a result of the back-tracing way that advection is done, and should be resolved. Due to the same issue, fluids that touch a boundary will seem to "flow in" from that direction forever when there is a force acting away from that boundary.

Secondly, this method could be extended to three-dimensions with relative ease, and was originally planned to be. It was not, however, due to time constraints.

## 5 Summary

In summary, this project created a (mostly) successful implementation of a two-dimensional continuos fluid in WebGL. The Jacobi method for estimating the pressure equation proved fast and accurate enough for this 2D visual simulation.

The author was impressed by how easily textures could be used as the fluid "particles", and the usual assembly step skipped. Furthermore, the author finds that WebGL is a great framework for physics-based simulation development due to its relative ease in set-up, deployment and OS-independence.

## References

[1] 2019. WebGL API. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[2] Wang BIAO. 2017. Smoke Simulation using WebGL Technology. ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ 12 (Mar 2017), 1–6.

[3] Robert Bridson and Matthias Muller-Fischer. 2007. Fluid Simulation. *SIGGRAPH* (Aug 2007). https://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf

[4] Pavel Dobryakov. 2017. WebGL Fluid Simulation. https://github.com/PavelDoGreat/WebGL-Fluid-Simulation

[5] Mark J. Harris. 2004. *GPU Gems* (1 ed.). http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch38.html