Basic Liquid Simulation in Vanilla WebGL

Yahiya Hussain
Yahiya.Hussain001@umb.edu
University of Massachusetts Boston

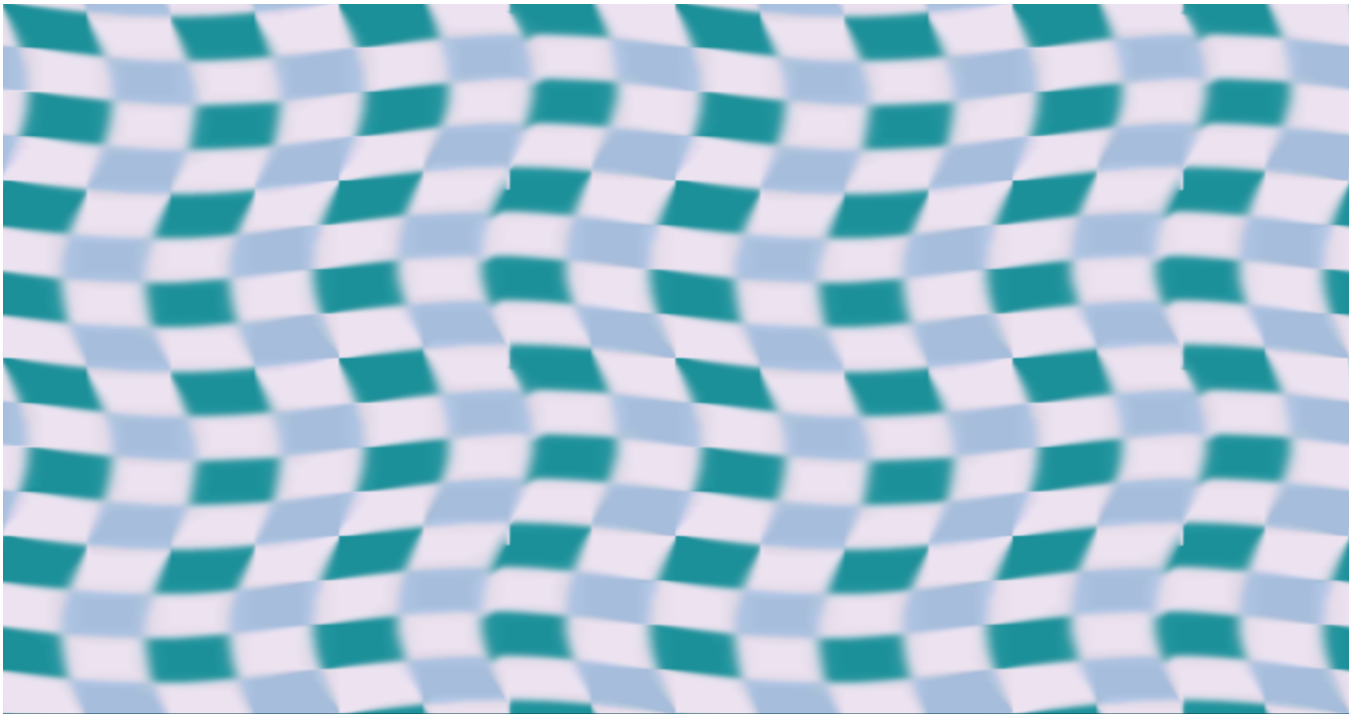

Figure 1: The beginnings of virtual liquid!

ABSTRACT

This semester I embarked on a treacherous journey to simultaneously learn the ins and outs of WebGL and learn the basics of liquid simulation on a computer. I managed to create the first starts at good 2D simulation of water and also created a lot of cool little mathematical art projects along the way. I also learned and implemented in-depth knowledge of how shaders work, especially with textures, which were absolutely vital to this project. I learned and implemented basic equations of computational liquid simulation such as the Navier-Stokes Equations for liquid with no compression using iterative methods to calculate quantities such as pressure fields such as the Jacobi iteration method and ping-ponging. I still have a lot to learn, but I have put real work in since September to make this idea into a reality.

CS460, Fall 2019, Boston, MA

ACM ISBN 1337.
https://CS460.org

KEYWORDS

WebGL, Visualization, Vanilla, Navier-Stokes, Vectorfield, ping-ponging, Jacobi Iteration, boilerplate

ACM Reference Format:
Yahiya Hussain. 2019. Basic Liquid Simulation in Vanilla WebGL. In *CS460: Computer Graphics at UMass Boston, Fall 2019*. Boston, MA, USA, 3 pages. https://CS460.org

1 INTRODUCTION

Liquid: so mysterious and so beautiful and yet scientists have made so much progress towards simulating liquid in an accurate and useful way. Whether modelling the corrosion of pipes or the air flying past airplane wings, the Navier-Stokes equations are indespensible for the purpose of simulating the real world so accurately. I wanted access to this magical mathematical world and so I embarked on this project. I even learned Vanilla WebGL since I wanted to really see the gears in motion in my machine and because 2D visualization isn't very 3D heavy.

2 RELATED WORK

During the course of the project, I have also created loads of spin-off projects to dazzle my eyes and those of others. It really is an art

to make bugs in the code that just splatter the screen with works of magnificent mathematical art. These can be found in loads of folders in the github repo, but it seems they don't all work properly in Github, so it may be better done in a python http server.

3 METHOD

My project is not a simple one because with great power in WebGL comes great boilerplate code, but I can focus on key highlights. At the very top, I set up my shaders using a large number of XMLHTTP Requests. I borrowed some utils from the book "Programming in Webgl," which have been heavenly in removing bugs and bloat from my code, with a bunch of tweaks to make them more to my liking of course. I required a lot of different shaders, all designed to accomplish different purposes. I always used the same vertex shader, because all I used it for was to pass texture coordinate data to the fragment shader and since I was always rendering to a 2D quad, this code never changed. I needed an advect shader, divergence shader, pressure shader, and subtract shader and I will discuss these later on in this paragraph.

I also set up quad geometry and texture coordinates, but I haven't changed that code in months because it was all I needed. I first started out by initializing a variety of textures. I set up two textures full of colors, one of which I initialized to a cool icy grid pattern that you see in the screenshot. In addition to many textures initialized, I set up 2 pairs of velocity textures. I needed to have 2 velocity textures in order to store x and y floats for the velocities in different directions and I needed 2 of these kinds of pairs for a programming technique called ping-ponging. Since GPUs have all pixels calculated at basically the same time, you cannot be assured that a pixel will have been updated before you read it. Thus in order to modify a texture at every time-step efficiently I need to ping-pong: I used a texture as an input and the other as an output in a framebuffer and then switched them every frame. Apparently the canvas is a framebuffer, who knew!

I had a SetTimeInterval function which called the Render function at set time intervals in order to allow my code to run easily and decently smoothly with little thinking and pain. In this render loop I ran the advect shader in order to cause the velocity textures to move their own velocity values one time-step and then I used the new velocity texture, according to the Navier-Stokes equations, to calculate the divergence of each pixel and store that in a divergence texture. I then used Jacobi iteration to iteratively solve the system of pressure equations in a for loop, using ping-ponging again for pressure textures. Then at the end I fed the pressure texture into the subtract shader to subtract the pressure gradient field from the velocity field as Navier-Stokes says. Then I at long last moved the colors using the new velocity texture and then renderered it to the screen. I can't believe how powerful the average GPU is to handle so much math in milleseconds!

3.1 Implementation

What was really fun was using SetTimeInterval to easily set up a rendering loop. Javascript is really cool!

```
setInterval(() => {
    render();
  }, 40);
```

I created and abstracted functions in order to get the rendering done in an easy to understand and quick to bugtest way. I am pretty proud of this because it was quite a mess before.

```
render = function(){

//PING
// advect velocity in x direction through itself
advect_component(deltatime, [1./side, 1./side],
FB_VELOCITY1_X, FB_VELOCITY1_X, FB_VELOCITY1_Y,
fb_velocity2_x);
// advect velocity in y direction through itself
advect_component(deltatime, [1./side, 1./side],
FB_VELOCITY1_Y, FB_VELOCITY1_X, FB_VELOCITY1_Y,
fb_velocity2_y);
// subtract calculate and subtract pressure
subtract_pressure(deltatime, density, [1./side,
1./side], FB_VELOCITY2_X, FB_VELOCITY2_Y,
fb_velocity1_x, fb_velocity1_y);
// advect colors texture 1 through the velocity and
store in colors texture 2
advect_component(deltatime,  [1./side, 1./side],
FB_COLORS1, FB_VELOCITY1_X, FB_VELOCITY1_Y, fb_colors2);

// draw it
renderToTexture(gl, null, canvas.width, canvas.height);
gl.drawArrays(gl.TRIANGLE_STRIP, 0, n);

//PONG
// do same thing again
advect_component(deltatime, [1./side, 1./side],
FB_VELOCITY1_X, FB_VELOCITY1_X, FB_VELOCITY1_Y,
fb_velocity2_x);
advect_component(deltatime, [1./side, 1./side],
FB_VELOCITY1_Y, FB_VELOCITY1_X, FB_VELOCITY1_Y,
fb_velocity2_y);
subtract_pressure(deltatime, density, [1./side,
1./side], FB_VELOCITY2_X, FB_VELOCITY2_Y,
fb_velocity1_x, fb_velocity1_y);

// except advect colors texture 2 through velocity and store in colors
advect_component(deltatime,  [1./side, 1./side],
FB_COLORS2, FB_VELOCITY1_X, FB_VELOCITY1_Y, fb_colors1);

// draw it
renderToTexture(gl, null, canvas.width, canvas.height);
gl.drawArrays(gl.TRIANGLE_STRIP, 0, n);
}
```

3.2 Milestones

3.2.1 Set up Quad geometry. It took a really long time to get geometry/vertex data for a quad and texture coordinates going great, but "WebGL Programming Guide," was of great utility.

3.2.2 Rendered using multiple shaders. Compiling shaders and linking uniform locations gets really complicated and luckily I

abstracted and made it easy, but then I had to learn how to switch shader contexts and upload new data to shader uniform locations quickly and easily. I always tended to choose ease over performance.

3.2.3 Learned to switch textures to allow ping-ponging. Now this was a confusing doozie. I had to understand that textures are stored in an internal array of bindings in WebGL, not to mention that creating shaders is complicated and rendering to a shader by putting it inside a framebuffer is also weird.

3.2.4 Implemented Navier-Stokes. I greatly leveraged my new-found texture prowess in order to create a plethora of textures to store everything from pressure to divergence and velocity components. I finally got a basic liquid simulation to work. It was a long journey and although there is still a lot of reworking to be done to make it more liquid-like, I think I did a good job.

3.3 Challenges

Describe the challenges you faced.

- Challenge 1: Vanilla WebGL is hard and complicated
- Challenge 2: Navier-Stokes equations are weird and foreign
- Challenge 3: Tutorials are written by people who are too experienced
- Challenge 4: There is anxiety in forcing yourself outside your comfort zone, again and again

4 RESULTS

In addition to the beginnings of liquid simulation, I have made loads of spin-off projects which are all cool and viewable in a suitable python http server, or on github if possible. Here are some of my favorites:

Figure 2: Rainbow Delight

5 CONCLUSIONS

This semester was a tough one. I learned so much graphics programming, I can't even describe it. After geometry was squared away, I was free to experiment like crazy with textures and go mad, and that I did. I made tons of cool little demos and came closer to simulating the complex nature of liquid than I ever could have imagined doing 2 years ago when I learned to code in college. This was a really fun and deeply rewarding experience and I can't wait to find out what I push myself out of my comfort zone to do next.

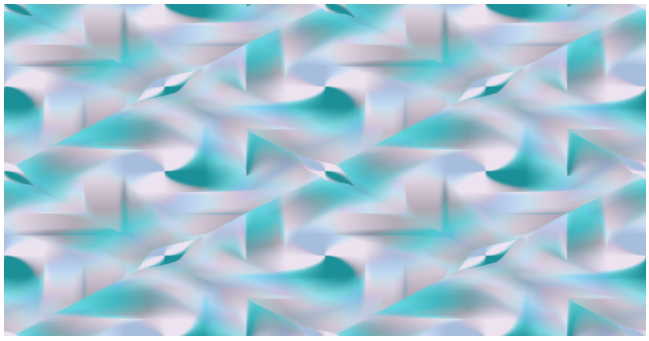

Figure 3: Diamonds!

Figure 4: Clouds of Red

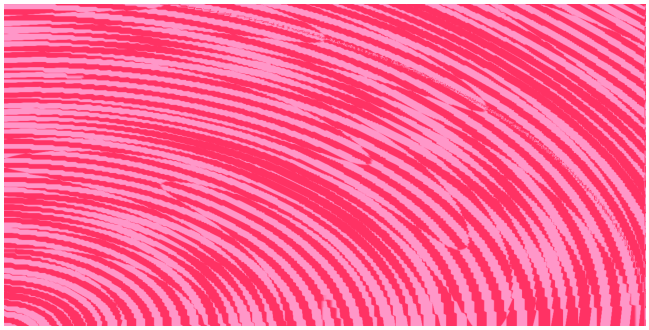

Figure 5: Swirls

REFERENCES

[1] Ricardo Cabello et al. 2010. Three.js. *URL: https://github. com/mrdoob/three.js* (2010).
[2] Daniel Haehn, Nicolas Rannou, Banu Ahtam, P. Ellen Grant, and Rudolph Pienaar. 2012. Neuroimaging in the Browser using the X Toolkit. *Frontiers in Neuroinformatics* (2012).
[3] Kouichi Matsuda and Rodger Lea. 2013. *WebGL programming guide: interactive 3D graphics programming with WebGL.* Addison-Wesley, Boston, Massachusetts.