

Jesse Allas

Shuang Zhao

CS 114

6/11/2020

## CS 114 Fluid Dynamics Simulation

**Project Overview:** The goal of this project was to simulate fluid dynamics in a 2D space. I did this in Javascript and HTML with a 200 x 200 vector field. With this vector field, I simulate the motion of fluids through the field using the Navier-Stokes Equations. These equations are commonly used in physics to simulate fluid dynamics, so using them here made sense. My goal is to calculate the vector and density of multiple fluids in the vector field. I am displaying the presence of fluids by using a different color for each fluid (red, green, blue) and the heavier the color appears in a location in the vector field, the denser the liquid is at that point. To handle the motion of the fluids, the vector field calculates the vector and density. Other variables include viscosity, diffusion, and timestep (rate of time).

**Derivation:** The two things that my code has to do is change the velocity in the vector field and change the density in the vector field. To do this, I am using the Navier-Stokes Equations, which are commonly used in physics to describe the motions of physics. With this in mind, I am using the incompressible version of the Navier-Stoke Equation. My reasoning for using the incompressible fluids is that I am not handling changes in volume, and incompressible fluids are simpler. Regardless, here is the definition of the incompressible Navier- Stokes Equation.

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \nu \nabla^2 u = -\nabla w + g$$

$u$  is the velocity

$\nu$  is viscosity

$\frac{\partial u}{\partial t}$  is the variation in velocity

$\nabla$  is the divergence for the point in the vector field.

$-\nabla w$  is internal sources

$g$  is external sources.

For the purpose of my simulation, there are no internal sources or external sources, so the function simplifies to this:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \nu \nabla^2 u = 0$$

Of course implementing sources is as straightforward as adding a variable to the system, but this didn't lead to interesting results, so I cut it out. Something like gravity could be considered one of these sources.

With the function simplified, what we are looking for is the variation in velocity, so we rewrite the function as this:

$$\frac{\partial u}{\partial t} = - (u \cdot \nabla)u + \nu \nabla^2 u$$

The variation in velocity is the updated velocity after a step in time so my code will update each time step to calculate the new velocity by using the above method.

With the function simplified, it is at this point that I would like to point out that this function belongs to the class of convection diffusion equations, and they make up this form:

$$\frac{\partial c}{\partial t} = D \nabla^2 c - v \cdot \nabla c$$

Where c is the variable of interest (in this case velocity), D is the diffusivity (in this case viscosity) and v is the velocity. Using a convection diffusion equation, we can handle the second part of our system, which is the transfer of density in our velocity field.

$$\begin{aligned} \frac{\partial \rho}{\partial t} &= k \nabla^2 \rho - u \cdot \nabla \rho \\ &= - u \cdot \nabla \rho + k \nabla^2 \rho \end{aligned}$$

In this case,  $\rho$  is density, and is our variable of interest. k is diffusion and u is velocity. Since v and u are both velocity, I don't need both of these variables, so from this point on I will only use u to reference velocity.

These two functions are what I will be simulating in my code, first I will calculate for velocity, and then I will calculate density with these new velocities.

### Velocity (Equation 1)

$$\frac{\partial u}{\partial t} = - (u \cdot \nabla) u + \nu \nabla^2 u$$

### Density (Equation 2)

$$\frac{\partial \rho}{\partial t} = - u \cdot \nabla \rho + k \nabla^2 \rho$$

Each timestep I will update the velocity of the points in the vector field (equation 1) and after that I will use these velocities to calculate the motion of the densities (equation 2).

## Code

### Variables:

As mentioned before, my code first handles updating velocity, and then updating density. It is important to note that there will be three different fluids in the system. To get started, I need a set of variables and I need to initialize a vector field.

```
//vector field data
var colors;
var vertices;

var N = 256; //grid dimension
var iter = 4; //iterations

//previous densities
var s = [];
var s2 = [];
var s3 = [];
//current densities
var density = [];
var density2 = [];
var density3 = [];
```

```
//velocity of vector field
var vx = [];
var vy = [];
var vx0 = [];
var vy0 = [];

//diffusion for three fluids
var diff = 1;
var diff2 = 1;
var diff3 = 1;

//viscosity for three fluids
var visc = 1;
var visc2 = 1;
var visc3 = 1;
var dt = 0.1; //time step
```

**Colors** and **Vertices** are lists used for the initialization of the vector grid, they're used to denote the vertices of each square in the grid, along with the colors of each square. **N** is used to denote the size of the grid, the vector field is an **N x N** grid. **Iter** is used during diffusion, more on this latter. **S** and **Density** are lists used to note the density at each space in the vector field. **S** is the previous density at that point, and **Density** is the current density at that point. Note that there are three lists for both **S** and **Density**, that is because there are three fluids. **Vx**, **Vy**, **Vx0**, and **Vy0** are horizontal and vertical velocity, and previous horizontal and vertical velocity. These four are lists storing all the points in the grid. Unlike the density variables, there aren't three sets of these, since all the fluids are sharing the vector field. **Diff**, **Diff2**, and **Diff3** are the diffusion rates for each fluid. **Visc**, **Visc2**, and **Visc3** are the viscosities of each fluid. **Dt** is the time step. For our code we will update the vector field to simulate fluids after ever 0.1 second.

**Initialization:** With our variables defined, we can initialize our vector field. On the left hand side you can see my initialization of the vector grid. A double for loop with the dimensions -1 to 1 will initialize the vector grid in the center of my camera. I use **step** to note how many squares I want in my grid, and **N** is then set to be the dimensions. Looping through the grid dimensions, I create a square out of six vertices and six colors, one for each of those vertices (the colors are all zeroes because they're empty) Then for each of these squares, I add an empty variable to the density and velocity lists.

```
step = 0.1; //this determines the number of gridsquares
var vertcount = 0; //counting vertices
N = 2/step;
vertices = []; //array of verts
colors = []; //color array

for (i = -1 - step; i < 1 + step; i += step)
{
  for (y = -1 - step; y < 1 + step; y += step)
  {
    sq += 1;

    vertices.push(i, y, 0.0);
    vertices.push(i, y + step, 0.0);
    vertices.push(i + step, y, 0.0);

    vertices.push(i + step, y + step, 0.0);
    vertices.push(i, y + step, 0.0);
    vertices.push(i + step, y, 0.0);

    colors.push(0.0,0.0,0.0);
    colors.push(0.0,0.0,0.0);
    colors.push(0.0,0.0,0.0);
    colors.push(0.0,0.0,0.0);
    colors.push(0.0,0.0,0.0);
    colors.push(0.0,0.0,0.0);
    vertcount += 6;

    s.push(0.0);
    s2.push(0.0);
    s3.push(0.0);
    density.push(0.0);
    density2.push(0.0);
    density3.push(0.0);
    vx.push(0.0);
    vy.push(0.0);
    vx0.push(0.0);
    vy0.push(0.0);
  }
}
```

**Simulation:** This is the meat of the code, this is how I handle the fluid simulation. Let's look back at the equations I derived.

### Velocity (Equation 1)

$$\frac{\partial u}{\partial t} = - (u \cdot \nabla)u + \nu \nabla^2 u$$

### Density (Equation 2)

$$\frac{\partial \rho}{\partial t} = - u \cdot \nabla \rho + k \nabla^2 \rho$$

In both cases,  $\nu \nabla^2 u$  and  $k \nabla^2 \rho$  are the diffusion, the spreading of the liquid at a point.

$-(u \cdot \nabla)u$  and  $-u \cdot \nabla \rho$  are the convection, or how the fluid follows the velocity in the vector field. So if I was to boil it down to the basics, my code would run in four steps:

Diffuse velocity, convect velocity, diffuse densities, convect densities. I run my code through an update function that follows those four steps with a few added parts.

```
//get viscosity
var v1 = visc0n ? visc : 0.0;
var v2 = visc20n ? visc2 : 0.0;
var v3 = visc30n ? visc3 : 0.0;

//diffuse velocity
diffuse([1,2], [vx0,vy0], [vx,vy], v1,v2,v3, dt);

project(vx0, vy0, vx, vy);

//convect velocity
convect([1,2], [vx,vy], [vx0,vy0], vx0, vy0, dt);

project(vx, vy, vx0, vy0);

//get diffusion
var d1 = diff0n ? diff : 0.0;
var d2 = diff20n ? diff2 : 0.0;
var d3 = diff30n ? diff3 : 0.0;

//diffuse density
diffuseDye([0,0,0], [s,s2,s3], [density,density2,density3], [d1, d2, d3], dt);

//convect density
convect([0,0,0], [density,density2,density3], [s,s2,s3], vx, vy, dt);
```

First I check if viscosity is enabled. I created some user controlled systems to allow customization of how the simulation works, and customizing viscosity was one of those options. Next I diffuse the velocity. After this, I run a function called **Project**. This function maintains that the system is mass conserving, an idea not original to me, but to an article I found. After this, I run convect on the velocity, project one more time, and then diffuse and convect the density as well. Notice that I check if diffusion is enabled too, this is another customization option I gave the users.

### Diffusion for Velocity:

Diffusion is denoted as  $\nabla^2 u$ . Simply put, this is saying that the change in value is equal to the current velocity plus a combination of its neighbors times viscosity all relative to the timestep. It could be displayed as this:  $v_{new} = v_{old} + k\Delta t(v_{right} + v_{left} + v_{top} + v_{bottom} - 4v_{old})$

The reason we are adding everything to  $v_{old}$  is because we are taking the change in velocity and adding it onto our current velocity. At first glance, it should be as straightforward as looping through each point and calculating this value. The issue however is that values are being used as they are being calculated, which causes messy and inaccurate results. The method to use then would be Gauss-Seidel relaxation. This method iterates through the calculation multiple times, using the values as they are created to give an accurate result. I won't go into a full explanation of Gauss-Seidel relaxation, but this requires rewriting the function. Initially taking a function of form  $a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$ ; it rewrites the function as  $\frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} = x_1$  so the above function won't work because rewriting it would solve for old velocity. Instead we create a function that calculates the diffusion backwards in time.

$$v_{old} = v_{new} - k\Delta t(v_{right} + v_{left} + v_{top} + v_{bottom} - 4v_{new}).$$

$$v_{old} = v_{new} - k\Delta t v_{right} - k\Delta t v_{left} - k\Delta t v_{top} - k\Delta t v_{bottom} + 4k\Delta t v_{new}.$$

Now we can rewrite it in the correct form.

$$\frac{v_{old} + k\Delta t v_{right} + k\Delta t v_{left} + k\Delta t v_{top} + k\Delta t v_{bottom}}{1 + 4k\Delta t} = v_{new}$$

As seen in the next code snippet, this final form is what is implemented to find the new velocity.

```

var i,j,k;
for (k = 0; k < 4; k++) {
  for (i=1; i<=N; i++) {
    for (j=1; j<=N; j++) {
      for (var l = 0; l < b.length;l++) {
        //Gambill method viscosity implimentation
        var visct;
        var m1 = 0.0; var m2 = 0.0; var m3 = 0.0; var d1 = 0.0; var d2 = 0.0;
        var V = density[IX(i,j)] + density2[IX(i,j)] + density3[IX(i,j)];
        var thresh = 0.0;
        if (density[IX(i,j)] > thresh) {
          m1 = density[IX(i,j)] * V * Math.pow(diff,1/3); //mv1
        }
        if (density2[IX(i,j)] > thresh) {
          m2 = density2[IX(i,j)] * V * Math.pow(diff2,1/3); //mv2
        }
        if (density3[IX(i,j)] > thresh) {
          m3 = density3[IX(i,j)] * V * Math.pow(diff3,1/3); //mv3
        }
        /////
        visct = Math.pow(m1+m2+m3,3);

        var kt = dt * visct * (N - 2) * (N - 2);
        x[1][IX(i, j)] = (x0[1][IX(i, j)]
          + kt*(x[1][IX(i+1, j)] + x[1][IX(i-1, j)]
            + x[1][IX(i, j+1)] + x[1][IX(i, j-1)]))/(1 + 4*kt);
      }
    }
  }
  set_bnd(b, x);
}

```

As seen above, **kt** is the calculated diffusion rate given the viscosity and the dimensions of the grid (it's N-2 because the edges are walls to keep the fluid inside). This runs into a problem however that I haven't touched on; what happens when multiple fluids are in a gridspace at one time? Then the viscosity becomes a mixture of the two. With that, I present the Gambill Equation:

$$\nu^{1/3} = x_a \nu_a^{1/3} + x_b \nu_b^{1/3}$$

The Gambill Equation is a derivation of viscosity given multiple mixtures, where x is the mass of the fluid, and v is the viscosity for that fluid. Given there are three fluids in this system, I can write the function as:

$$\nu = \left( x_a \nu_a^{1/3} + x_b \nu_b^{1/3} + x_c \nu_c^{1/3} \right)^3$$

Now the question becomes, what is the mass? Mass is equal to density times volume. However this system has no way of knowing volume, as it doesn't store it. For the purposes of this

simulation, I am setting mass equal to the three densities combined in the given point. While not accurate to real life, it does provide a nice distribution for the purposes of calculating viscosity.

### Convection/Advection for Velocity:

The next part in our equations are  $-u \cdot \nabla \rho$  and  $-(u \cdot \nabla)u$ , which Wikipedia labels as convection. Convection is the transfer of heat in fluids, which isn't entirely accurate to what this code is doing. A better definition would be **Advection**; the transfer of matter by a flow of liquid. Advection is how the fluid is carried around via velocity. Looping through each point, we can use the velocity and time step to calculate the distance traveled. Taking this distance traveled and calculating where it lands is difficult since if it doesn't land perfectly in a square, then we have to evenly distribute it amongst its neighbors. What is easier is to look at each point and calculate the previous position, and place that value into the current point. Of course the previous point is not guaranteed to be perfectly in a square either, so we take the average of surrounding squares and proportionally scale them with respect to how close they are to the previous point.

```
for(j = 1, jfloat = 1; j < N - 1; j++, jfloat++) {
  for(i = 1, ifloat = 1; i < N - 1; i++, ifloat++) {
    for (var l = 0; l < d.length; l++)
    {
      //find the previous points using the distance
      tmp1 = dtx * velocX[IX(i, j)];
      tmp2 = dty * velocY[IX(i, j)];

      x = ifloat - tmp1;
      y = jfloat - tmp2;

      //get the neighboring points and scales
      if(x < 0.5) x = 0.5;
      if(x > Nfloat + 0.5) x = Nfloat + 0.5;
      i0 = Math.floor(x);
      i1 = i0 + 1.0;
      if(y < 0.5) y = 0.5;
      if(y > Nfloat + 0.5) y = Nfloat + 0.5;
      j0 = Math.floor(y);
      j1 = j0 + 1.0;

      s1 = x - i0;
      s0 = 1.0 - s1;
      t1 = y - j0;
      t0 = 1.0 - t1;

      var i0i = Math.floor(i0);
      var i1i = Math.floor(i1);
      var j0i = Math.floor(j0);
      var j1i = Math.floor(j1);

      //set the current position to the combined value of the previous point calculation
      d[1][IX(i, j)] = s0 * ( t0 * d0[1][IX(i0i, j0i)] + t1 * d0[1][IX(i0i, j1i)] )
        + s1 * ( t0 * d0[1][IX(i1i, j0i)] + t1 * d0[1][IX(i1i, j1i)] );
    }
  }
}
```



Notice the last for loop. **d** is a list of lists, each one being a setting that needs to be updated, here we are looping through vertical and horizontal velocity.

### Advection and Diffusion for densities:

Everything I covered so far handles the transferring of velocity through the vector field, but this won't mean much without moving around the fluid densities. Fortunately this is more or less the same method, only simpler. Diffusion is the same as before, only instead of moving velocity, we are moving the densities. This means we don't need to calculate viscosity. Instead we use our diffusion rate.

```
for (k = 0; k < 4; k++)
{
  for (i=1; i<=N; i++)
  {
    for (j=1; j<=N; j++)
    {
      for (var l = 0; l < 3; l++)
      {
        var a = dt * diff[l]
          * (N - 2) * (N - 2);

        x[l][IX(i, j)] =
          (x0[l][IX(i, j)]
          + a*(      x[l][IX(i+1, j)]
          +x[l][IX(i-1, j)]
          +x[l][IX(i  , j+1)]
          +x[l][IX(i  , j-1)])))/(1 + 4*a);
      }
    }
  }
}
```

Notice the last for loop, given there are three fluids with three densities, I loop through each one and update it. Other than that, the code remains unchanged.

Advection remains unchanged, in fact we can use the same function for both. The only difference is that instead of changing vertical and horizontal velocity, we are looping through the three different densities.

### **Set\_bnd and Project:**

The last two functions are used to keep the system running smoothly. These are also heavily taken from a set of articles I read about fluid dynamics, with very minimal changes needed on my part.

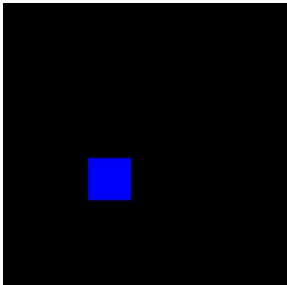
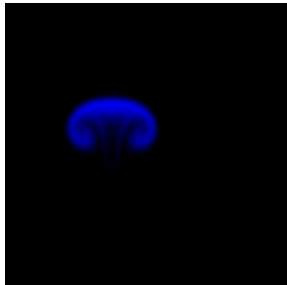
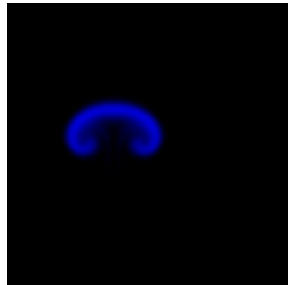
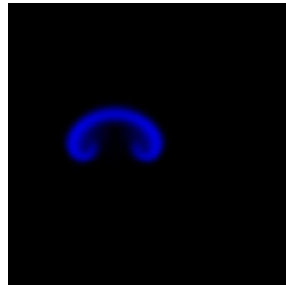
The first function is called **Set\_bnd**, and it is used to set the walls of the vector grid. The whole system is taking place in a 2D box, and we don't want anything "leaking" out of the system, so this function is used to redirect any flow away from going off the edge of the grid. Think of it as creating walls for the system. The function is very simple, any time a velocity vector component (x or y) would point out of the box, it flips that vector component to go in the opposite direction. I made very minimal changes to this based on it's original source, mainly because I don't have to. The two changes I made were to the dimensions of the box to better fit my vector grid, and to the efficiency of the system. The system originally took in the list of variables to edit one by one and edited each one separately. I changed the code to take in a list of lists, and edit each of those lists simultaneously. A minimal change, but it makes the code run just a bit faster.

The second function is called **Project**, and it is used to keep the system incompressible and mass conserving. It does this by using a Poisson Equation (a linear system), which is a commonly used equation for mass distribution, and Gauss Seidel relaxation. Again I made very little changes to this function, mainly because there was nothing that needed to be changed.

### **Experimentation:**

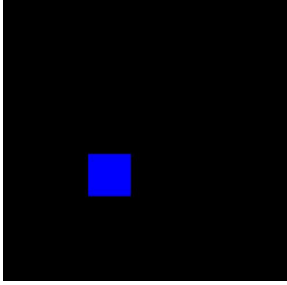
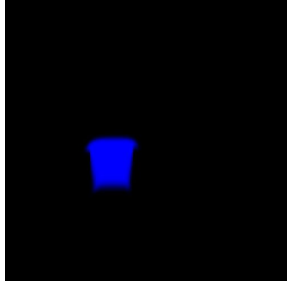
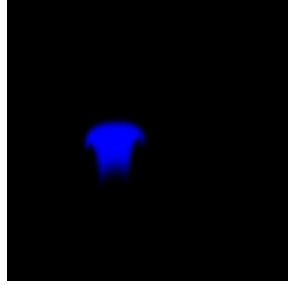
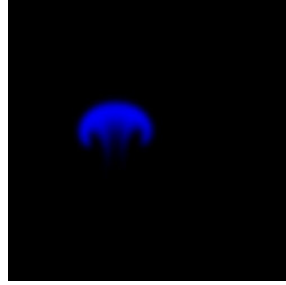
Of course I have to check to see how accurate my method is, so I conducted a few experiments. It is practically impossible to calculate a full vector field by hand, so calculating the velocities by hand is out of the question. Instead I will have to find other ways to test my code. With that in mind, I intend to base my testing on the timestep of the simulation. Normally, I have the timestep set to 0.1, this means that the code will simulate the motion of fluids in 0.1 second intervals. Theoretically, given different timesteps, they should all generally follow the same path. When stopped at the same time, all the timesteps should have somewhat similar patterns. I say "somewhat" and "generally" because given a smaller timestep, the simulation should be more

accurate. So I started my simulation with a perfectly square fluid block in the center of the vector field (close enough). I also set each point in the square to have a vertical velocity of 0.3. With this starting point, I set my time step to 0.1 and let it run for 2 seconds in the simulation. That would mean 200 iterations of my code. I then repeat it for 0.2 seconds at 50 iterations, and 0.05 seconds at 400 iterations. Below are my results.

Starting position. Time = 0.0	Time step = 0.2 Time = 2.0	Time step = 0.1 Time = 2.0	Time step = 0.05 Time = 2.0
			

As you can see, the fluid has the same flow with each timestep, only with each smaller timestep getting more and more complete. Given the results, the fluid does as I expect; accurately moving through the system with the correct velocity, with smaller timesteps leading to more precise results.

With my velocity working correctly, the next part of the code that I want to test is the viscosity. I will use the same method, testing the motion of the fluid just as before, only this time I will include viscosity. This time I am setting velocity to 2.0 and viscosity to 0.0001.

Starting position. Time = 0.0	Time step = 0.2 Time = 2.0	Time step = 0.1 Time = 2.0	Time step = 0.05 Time = 2.0
			

Again, the fluid follows the same flow with each time step, only the differences now are more noticeable. I believe this is because of how viscosity is implemented in the Navier-Stokes equations. The reduction from viscosity relies on the current velocity based on the laplacian term. Velocity in the timestep is smaller given smaller timesteps, so the change in viscosity is spread out over a larger period of time.

**Incompressible Navier-Stokes equations** (*convective form*)

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} - \nu \nabla^2 \mathbf{u} = -\nabla w + \mathbf{g}.$$

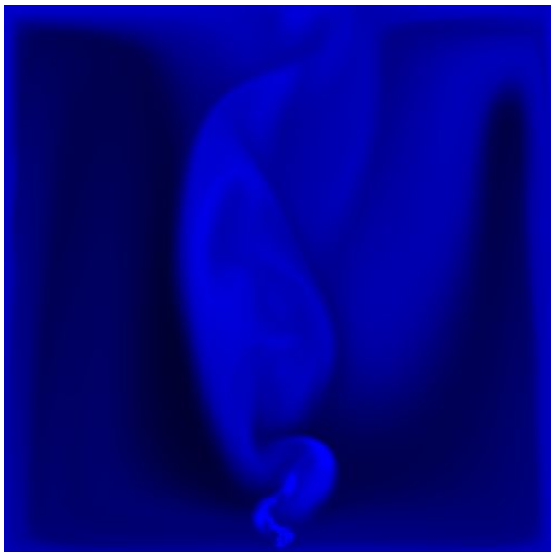
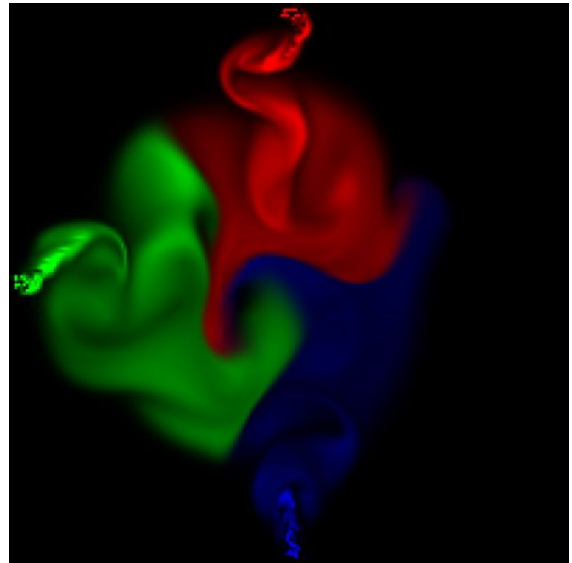
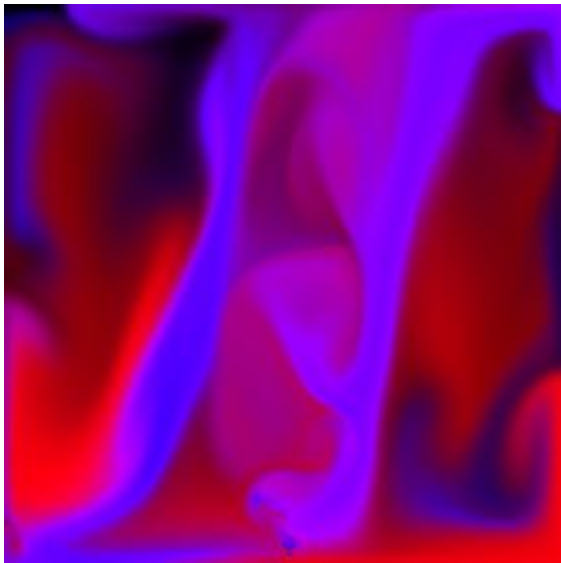
### User Controls:

When this project runs, the user is greeted to an empty, black vector grid. On the side there are multiple tools the user can click on to run the system. First there are three sliders, each one controls three different fluids. Turning one on will shoot out that fluid like a hose from the side of the grid. Increasing the slider increases the rate at which the fluid is outputted. Below these settings are three more sliders, turning these on to activate diffusion for each fluid, and the soldiers change the rate of diffusion for the respective fluid. Below these are the viscosity switches. These work the same as the diffusion switches, only this time they change the viscosity for each fluid.

**Closing Thoughts:**

This was a very difficult task. I took a lot of information out of different articles detailing the implementation of fluid dynamics, but I still had to implement a lot of methods myself, since most didn't deal with viscosity, and none of them dealt with multiple fluids in a system.

However overall I am satisfied with my end result. I believe that velocity and the flow of density in the vector grid was implemented correctly. Viscosity I have my doubts on, but it still works in a realistic manner. I was surprised at how few sources there were talking about mixtures of viscosity and how it would affect velocity. Regardless, I am happy with my end product.



**Sources:**

Wikipedia: Navier-Stokes Equations

[https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes\\_equations](https://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations)

Wikipedia: Convection-Diffusion Equation

[https://en.wikipedia.org/wiki/Convection%E2%80%93diffusion\\_equation](https://en.wikipedia.org/wiki/Convection%E2%80%93diffusion_equation)

Wikiversity: Partial differential equations/Poisson Equation

[https://en.wikiversity.org/wiki/Partial\\_differential\\_equations/Poisson\\_Equation](https://en.wikiversity.org/wiki/Partial_differential_equations/Poisson_Equation)

Real Time Fluid Dynamics for Games by Joe Stam

<https://pdfs.semanticscholar.org/847f/819a4ea14bd789aca8bc88e85e906cfc657c.pdf>

Department of Computer Science & Information Engineering National Taiwan Normal

University: Iterative Methods by Berlin Chen

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.721.5373&rep=rep1&type=pdf>

Neutrium: Estimating Viscosity of Mixtures

[https://neutrium.net/fluid\\_flow/estimating-the-viscosity-of-mixtures/](https://neutrium.net/fluid_flow/estimating-the-viscosity-of-mixtures/)

Iterative Methods Gauss-Seidel Method Jacobi Method

[http://web.iitd.ac.in/~hegde/acm/lecture/L11\\_system\\_of\\_eqns\\_iterative\\_methods.pdf](http://web.iitd.ac.in/~hegde/acm/lecture/L11_system_of_eqns_iterative_methods.pdf)

Gamasutra: Practical Fluid Dynamics by Mike West

[https://www.gamasutra.com/view/feature/1549/practical\\_fluid\\_dynamics\\_part\\_1.php?print=1](https://www.gamasutra.com/view/feature/1549/practical_fluid_dynamics_part_1.php?print=1)

Lectures in Elementary Fluid Dynamics by J. M. McDonough

<http://web.engr.uky.edu/~acfd/me330-lctrs.pdf>