Interactive 2D Screen Space Fluids with Shaders

Jaron Kong

Abstract

A method for the real-time simulation and rendering of fluids in 2D space. The technique itself is targeted at video game applications, with a focus on portability and ease of integration. Most of the fluid simulation and rendering is handled by fragment shaders running on the GPU, making integration fairly straightforward in a number of deployment settings. Running on the GPU also enables faster performance and frees up the CPU to perform other operations.

Introduction

Fluids are a common desired effect in video games. They can take the form of liquids like water, atmospheric effects like smoke, and special effects like explosions. Often times these effects, particularly special effects, are approximated with particle systems. While excellent for performance, particle systems are typically disconnected from the actual game world, that is, they are not interactive and do not react to the player.

In order to create compelling effects and environmental elements, things need react realistically to user interactions. Water should splash when stepped in. Smoke should distort when walked through. The list goes on.

In this project, I propose a simplified implementation of a liquid simulation and rendering technique for 2D applications. 2D space is particularly attractive for things like fluids because rendering is significantly simpler in 2D space than 3D. For this project, the simulation and rendering are essentially the same thing and take place on the same texture memory on the GPU.

The goal was about keeping things as self-contained as possible, as such, all simulation calculations are executed as textures passed through fragment shaders. The heavy use of fragment shaders is great in a 2D context due to the fact that I can directly render all results to a single quad and render that quad directly to the screen. This keeps things simple and fast.

I chose to implement my project in the GameMaker game engine. The engine supports a wide array of platforms and is an excellent tool for 2D game development.

The Algorithm

The algorithm itself is based on taking textures, passing them through a variety of fragment shaders, then rendering the results to another texture and passing that new texture through another fragment shader. It is essentially an assembly line of textures being moved through different fragment shaders.

Temperature [1 channel]: How hot the fluid at a pixel is.

Density [1 channel]: How dense the fluid at a pixel is.

Pressure [1 channel]: How compressed the fluid at a pixel is.

Velocity [2 channels]: How fast the fluid at a pixel is moving.

Divergence[2 channels]: How much a pixel at a current pixel needs to move to balance out the pressure.

In order to continuously render new frames, I needed a sort of double buffer for the textures to switch back and forth between so I have access to the old frame while writing to a new frame. To accomplish this I utilize two textures for every component of the fluid simulation, and have functions that will "swap" the texture pointers so I can alternate between them.

For the simulation itself, I make use of five different properties, each tracked in their own texture: temperature, density, pressure, velocity, divergence. There is also an extra texture for obstacles that will deflect the fluid. Temperature, density, pressure, and velocity have two textures each since they need to animate with the swapping technique discussed earlier. The divergence does not need to and is a single texture.

Density represents the location of the fluid. Temperature is how hot it is and affects things like buoyancy in fluids like smoke and fire. Pressure is the pressure of each pixel of the fluid. Velocity is how fast a pixel of fluid is moving. Divergence is similar to a derivative of velocity and reflects how much a single pixel is changing in a given frame.

Each of the fluid simulation properties is stored on a texture, where different color channels correspond to different variables. The temperature, density, and pressure properties only use the red color channel on their textures. Velocity and divergence use two, red for x value, and green for y value.

The following outlines the steps of the algorithm:

1. Apply the velocity to itself (advection).
2. Apply the velocity to the density (advection).
3. Apply the velocity to the temperature (advection).
4. Apply buoyancy.
5. Apply any fluid sources.
6. Compute divergence.
7. Compute pressure iteratively.
8. Apply pressure to velocity.

```
vec2 u = texture2D(gm_BaseTexture, v_vTexcoord).xy - vec2( 0.5 );
    vec2 timeScaledStep = u * TimeStep;
    vec2 coord = vec2( v_vTexcoord.x - timeScaledStep.x, v_vTexcoord.y +
timeScaledStep.y );
    gl_FragColor = vec4( vec3( Dissipation * (texture2D(SourceTexture,
coord).rgb - vec3(0.5)) + vec3( 0.5 )), 1.0  );
```

Advection Code

TimeStep is how fine each simulation step is. Smaller values take longer but yield better results.

Dissipation is how much energy is lost ambiently during advection.

The numerous addition and subtraction of 0.5 is to allow negative values to be stored in the texture as discussed in a later section of this report.

The concept of advection is at the core of the algorithm. Advection in this context means getting things to "move" on the textures. By looking up the velocity at a pixel, I can do a sort of backwards lookup to find the pixel where the current pixel "came from" and use the new pixel color. Over successive iterations, this results in visual movement. The reason for this backwards lookup involves the nature of how GPU fragment shaders work. One cannot change the fragment color of a different pixel, that is one cannot say "make the pixel ten units away red." Instead, one must say "the pixel ten units away is red, make this pixel red." Using this concept of advection, I can move pixels around on the textures. Above is the main piece of code for advection. There are several constants including a time step size and a dissipation constant. As seen by the first three steps of the algorithm, the advection will advance the state of the simulation one frame.

Buoyancy simulates convection currents caused by a difference in temperature within a fluid. The equation takes in a constant as well as the difference between the temperature at a pixel and the global ambient temperature. A vector indicating the upwards direction is also necessary. Buoyancy is the reason I need the temperature texture and the results of the buoyancy calculation are applied to the velocity texture.

$$f_{buoy} = \sigma\left(T - T_0\right)\hat{\mathbf{j}}.$$

Buoyancy equation.

Fluid sources are handled in a straightforward matter. In this application, fluids always start with zero velocity and with a predefined uniform density and temperature. To create a fluid source, one need only draw the shape of the source onto the density and temperature textures directly. For instance, to create a circular source of fluid on the screen, draw a circle onto the density and temperature textures. Because this is 2D and the simulation and rendering are essentially the same thing, I can directly place fluid sources onto the texture and have it automatically integrated into the simulation.

HalfInverseCellSize * (vE.x - vW.x + vS.y - vN.y)

Divergence Equation

vN, vS, vE, vW are the velocities of north, south, east, west neighboring pixels. HalfInverseCellSize is a constant based on the size of fluid cells in the simulation.

Divergence is a simple operation, and essentially takes a given pixel and looks at the neighboring velocities and takes the difference in X and Y velocities. The velocities are combined and multiplied against the inverse of the fluid cell size to reflect the total amount of fluid that is leaving the cell. The divergence is used for the pressure calculation later.

$$(pN + pS + pE + pW + Divergence) * InverseCellSize$$

Pressure Equation

pN, pS, pE, pW are the pressures of north, south, east, west neighboring pixels. Divergence is the divergence at a given pixel.

Computing pressure is the most intensive part of the algorithm, and requires several iterations to refine the results. The actual pressure computation is surprisingly simple, the pressure calculation shader basically looks at every pixel and its neighbors, and tries to equalize them, using the current pressure and divergence value. The equation is given by the sum of all neighboring pressures plus the divergence multiplied by the inverse of fluid cell size. Fluid cell size is basically the pixel size of the simulation. Fluids in real life tend to move from areas of high pressure to low pressure. By having each pixel try to equalize itself the simulation can slowly converge towards an equilibrium, or more accurately, converge towards the pressure needed to achieve equilibrium. What is written to the actual pressure texture during the pressure computation step is the pressure a pixel "wants" to exert to achieve equilibrium. Spending more iterations on this step leads to more accurate results but takes extra time to compute. For reference, around 40 iterations will lead to decent results reasonably quickly.

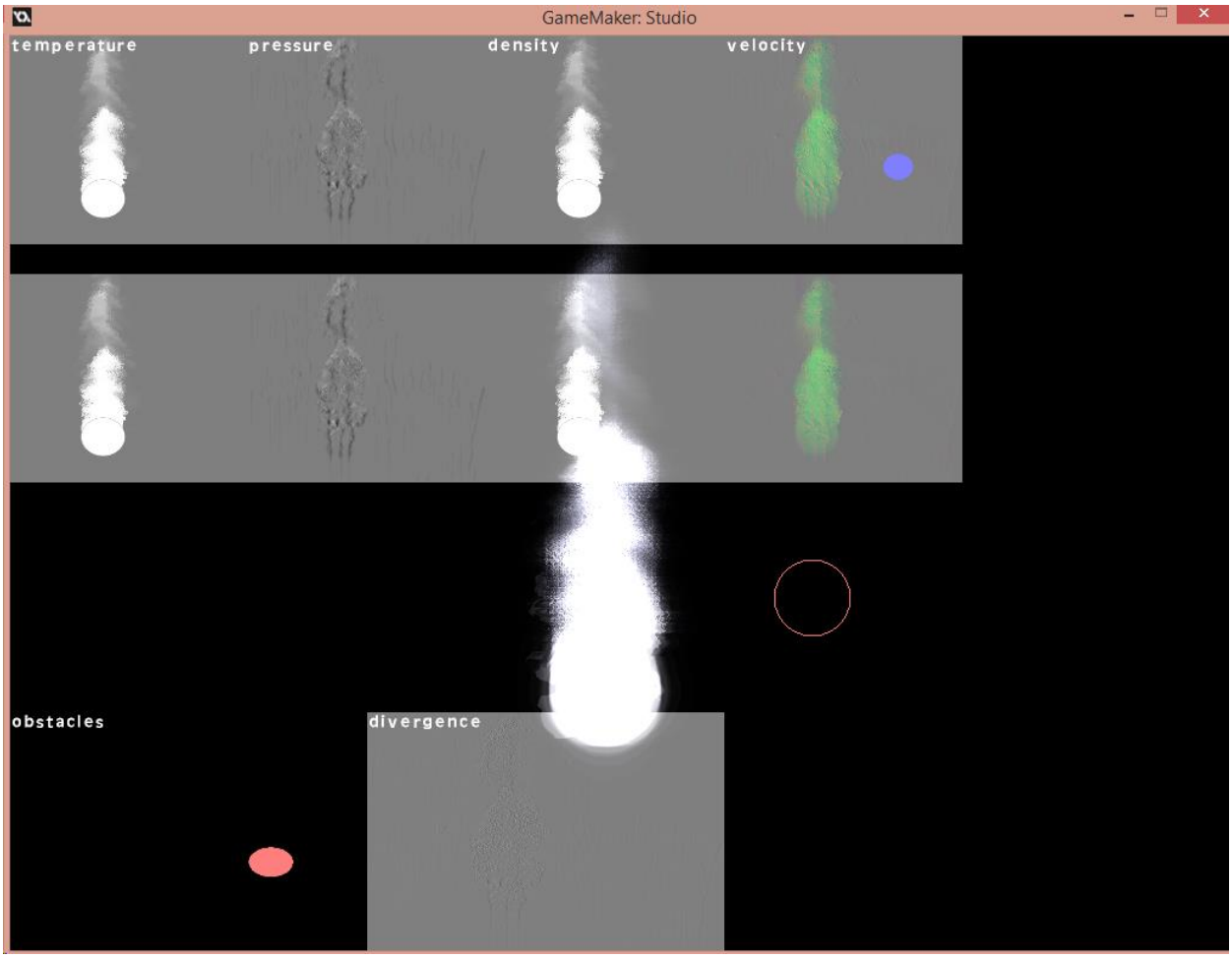
$$vec2(pE - pW, pN - pS) * GradientScale$$

Pressure to Velocity Equation

pN, pS, pE, pW are the pressures of north, south, east, west neighboring pixels. Divergence is the divergence at a given pixel.

Now that pressure is on the texture, I can apply the pressure to the velocity of the fluid. This is also fairly intuitive, and to calculate the velocity I simply take the difference in pressure between a pixel and its neighbors. The X velocity is determined by getting the difference between the east and west pixel pressures, and the Y velocity is determined by getting the difference between the north and south pixels. Larger differences mean greater velocities. This conversion does involve a constant, GradientScale, to convert between the pressure units and velocity units. This is set by the user and can be changed to create "thicker" and "thinner" fluids.

Results



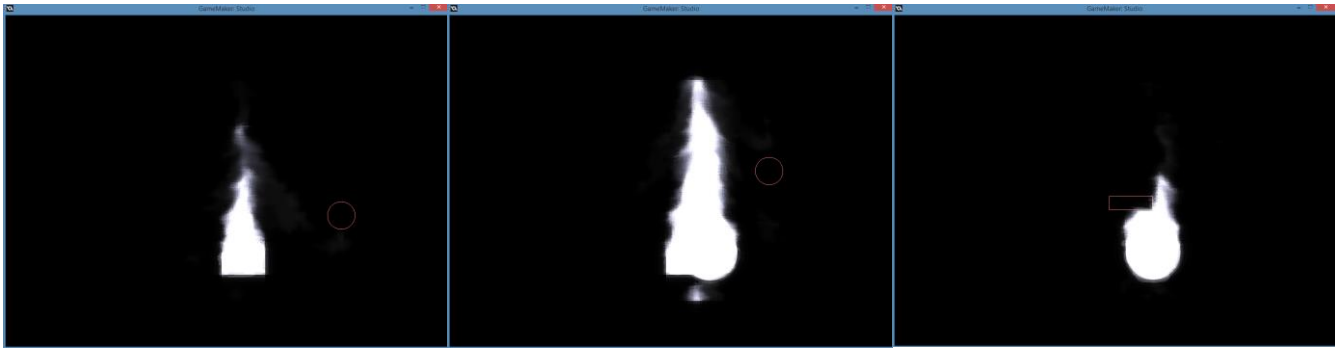
The end results of the application were very promising and would likely benefit from more time spent optimizing. The number of textures and texture lookups mean the actual performance scales very poorly with higher resolutions. At small 512 x 512 resolutions the simulation held a steady 60 frames per second. Increasing that to a larger 1024 x 768 lead to a drop to around 30 frames per second. For the simulation on its own this is acceptable, but considering the amount of GPU resources it uses, it leaves little room for anything else, limiting its usefulness in applications such as games.

To test the results, a subjective approach was taken. I simply looked at the simulation and how it interacted with itself and the obstacle, focusing mostly on how “fluid-like” the movement was. The goal of this application was not to create realistic fluids, but rather to create fluid-like behavior, so a subjective evaluation was all that was necessary.

As far as usability, the end result was very positive. Adding obstacles and sources was very simple and straightforward. In addition, the performance is not significantly hindered by more obstacles or fluid sources. Being a screen space simulation and rendering, all pixels of the

screen are always being operated on anyway, so adding some extra colors to the textures does not carry significant overhead.

The raw rendering of the final fluid was too dark. I compensated for this by adding several post process effects like refraction and bloom. Both went a long way to making the final fluid more interesting to look at. Both of these effects of course came with a performance cost, but it was marginal compared to the cost of the simulation itself.

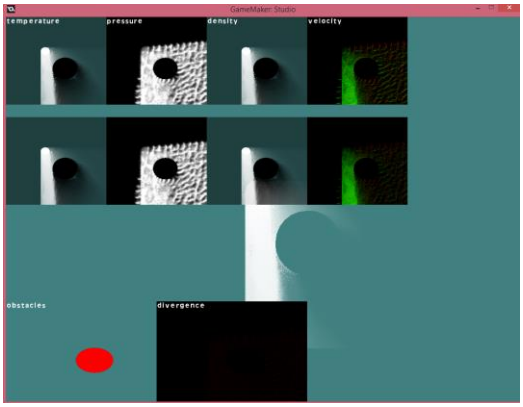


(left) Rectangle as the source shape. (center) Rectangle and circle overlapped as the source shape. (right) Rectangle as the obstacle shape.

Challenges

There was a decent amount of existing code available to look at for this project. Of course it is never as simple as dropping random code into a program and having it work.

Most of the code out there was written in newer versions of GLSL. Since I am implementing my project in the GameMaker game engine, I only have access to a very old version of GLSL ES 1.0 to write my shaders in. As a result I had to find workarounds for several things such as looking up neighboring pixels. This is not a huge problem, but just another thing to have to debug and work through.

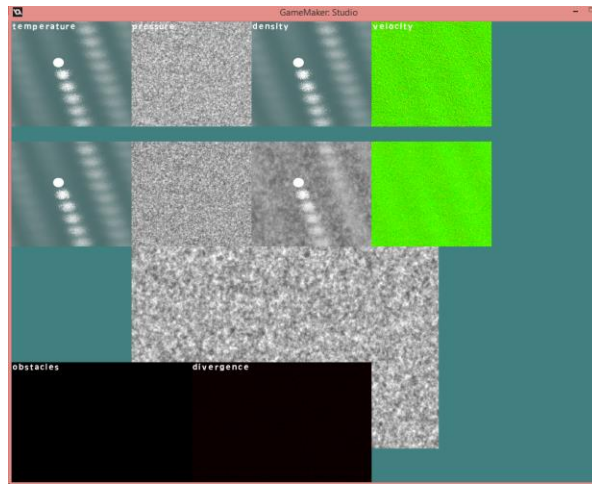


Things not only spiraled out of control, but there seem to be an axis or two missing from the movement…

GPUs are very fast, but very hard to debug and program for. Even though I was doing relatively simple things with fragment shaders, everything needs to be correct for things to work at all. Even a little mistake will yield undesirable results. While in traditional CPU programming I can use console output and debuggers to find the errors, GPU programming does not have nice tools

like that. In addition, GPUs will rarely crash at runtime, this means errors tend to spiral out of control and yield results that are hard to interpret and analyze. Most of my time was spent staring at many blocks of relatively small and simple code wondering why nothing is working. Sometimes the mistake was small like bad order of operations, other times they were more abstract issues like negative values, which brings me to my next point.

Probably the most time consuming issue to deal with was storing negative values in textures. It took a while to first get things to work, but when I did, the fluid only moved in one direction on each axis. After spending several hours combing through the code and trying to debug shaders with hacky color outputs, I discovered GameMaker could not store negative values in textures. A reasonable limitation to be sure, but a problem for my project. I kicked around several solutions, but eventually settled on encoding values on a scale from 0 to 0.5 and writing the number as a value from 0 to 1 on the texture. 0 is negative 0.5, and 1 is positive 0.5. Of course this means I need to modify the shaders to deal with this which means more shader debugging, and without a form of console print output, working through the shaders was tedious to say the least.



Errors build on themselves and just go crazy. GPUs never really “stop” they just keep going until the results don’t make sense.

Future Work

Optimization is the biggest issue that could be solved in the future. Despite the ease of use, and decent performance, the current simulation has relatively limited use in games due to the amount of resources it uses.

Texture optimization would be necessary in getting good performance. Using GameMaker’s default textures, each texture for fluid properties contains four color channels (rgba). A straightforward optimization could be manually disabling certain channels for properties that do not need them. Density for instance only needs one color channel to work, but GameMaker by default allocates four channels.

Caching neighboring pixel colors could also speed things up. Numerous shaders do four texture lookups to find the color of the same neighboring pixels. While eliminating these texture lookups during the pressure calculations would probably be impossible, caching lookups during the various advections could potentially help with performance.

Work Summary

Approximately 50 hours were spent working on this project, including research, experimentation, and debugging.

Works Cited

Akinci, Nadir, Alexander Dippel, Gizem Akinci, and Matthaias Taschner. "Screen Space Foam Rendering." *WSCG* (2013): n. pag. Web.

Harris, Mark J. "GPU Gems." - *Chapter 38. Fast Fluid Dynamics Simulation on the GPU*. Nvidia, Sept. 2007. Web. 11 May 2015.

Max N., Becker B. “Flow Visualization Using Moving Textures” 1995 Visualizing Time-Varying Data, 77-87

Van Der Laan, Sladimir J., Simon Green, Miguel Sainz, and Rijksuniversiteit Groningen. "Screen Space Fluid Rendering with Curvature Flow." *SIGGRAPH 2010* (n.d.): n. pag. Web.

Wijk, Jarke J. Van. "Image Based Flow Visualization." *ACM Transactions on Graphics* 21.3 (2002): n. pag. Web.