CPSC 490 Final Report
Fluid Simulation Based Painting
Jakub Kowalik

A screenshot of the application showing a mix of different colored paints

Overview
The goal of this project was to implement a fluid simulation system in the form of a painting application. The inspiration was that paint, when viewed through a reductionist lens, is nothing more than nicely colored fluid. Another goal was to implement GPU hardware acceleration for the simulation in order to maintain responsiveness at higher simulation resolutions, but that version of the application was not successful. The result is a Win32 C++ application that simulates the paint as fluid entirely on the CPU, and while that version is functional, the list of its shortcomings is much longer than the list of its accomplishments.

Accomplishments

Fun
The application is fun to play with. It may fail at facilitating artistry, but it is very entertaining as a sandbox. Anyone who might have opened MS Paint just to play around with the tools would likely enjoy playing with this application as well.

A Fluid and Substantive Paint
The paint appears to have depth and mass. Areas with more paint appear to be higher than areas with less paint. The paint flows, smears, and mixes somewhat realistically.
Implementation Details

Strokes

Because of the computational overhead of running the fluid simulation, this painting app is unable to poll the mouse fast enough to draw continuous-looking strokes by filling in pixel by pixel. The ability to draw a line between current and previous location is necessary for smooth-looking strokes. This application makes use of the Cairo vector drawing library for this function.

Fluid Sim

As outlined in the proposal, this application used the Stable Fluid model developed by Jos Stam to drive the core of the paint simulation. The heart of the simulation code is exactly as outlined by Stam in his paper titled “Real-Time Fluid Dynamics for Games,” with occasional changes to make the code cleaner for use with modern compilers. For example, the use of preprocessor definitions for inline functions has been removed and replaced instead by actual inline functions.

The simulation parameters were modified in two important ways for the purposes of simulating paint:

- The diffusion of density across the simulation has been completely suspended. The diffusion component of the sim tended to smooth out too much detail from the texture of the paint over time, resulting in blurry, unattractive strokes. Very thick fluids like paint should not diffuse as much as the simulation made it appear.
- The number of iterations for the back-solving components of the force diffusion code has been set to a very low number. More accurate back solving results in more wispy fluid, which is not a desirable characteristic for thick oil paint. Furthermore, the lower number of iterations translates to faster performance, making the app smoother and more usable.
Shading

One of the main goals of this project was to create a paint simulation that appeared to have substance. More than just pixels on a screen, the paint was designed to appear to have depth, mass, texture, and surface properties like glossiness. This part of the project was a relative success with the paint really having a nice physicality to it. This effect was created by combining two shading techniques (Blinn-Phong reflectance as well as normal mapping) using the density data from the simulation as the third dimension. The simulation’s density grid is used as a height field for computing the highlights and adjustments to the normals.

The Blinn-Phong shader contributes bright highlights that not only create the illusion of depth but also give the paint a nice glossy appearance.

*Depth cues are mostly from the specular highlights in this version of the shader.*

The normal mapping shader really highlights the paint’s typology giving the strokes very visible texture.

*Same simulation as above, but this version of the shader has very intense normal mapping.*

Currently this shading is implemented in the C++ code proper, but this is only for development purposes. In production, this code should be moved to the OpenGL rendering layer as GLSL shaders for performance purposes.
The OpenCL Implementation

A parallel, OpenCL implementation has been developed alongside a cpu-only implementation, but halfway through the project the OpenCL branch has been mostly abandoned. The main reason for that was the lack of any apparent performance gains from hardware acceleration—in fact, the cpu-only branch appeared more responsive in most scenarios. The lack of expected performance gains is likely due to both algorithmic limitations with adapting the current approach verbatim as well as a lack of fine tuning of the parameters. It turned out, upon actually trying to port the code into a series of OpenCL kernels that the algorithm would need to be modified slightly in order to take full advantage of the parallelization. Additionally, there are many other factors, such as local and global work group size, that might need to be more carefully considered in order to fully benefit from the hardware acceleration.

Challenges

Performance

Maintaining a responsive user experience proved to be a major difficulty. The simulation routine is very system intensive, and very slight differences in implementation can make a major difference in speed. For example, turning one commonly used function into an inline function literally tripled the speed of the application.

Integrating with Libraries

Many hours were spent finding nuanced bugs resulting from library-interfacing code. For example, the application needs to be able to access raw buffer data for Cairo textures; although the API is well documented, the documentation for the internal data representation that the library uses misses a few critical points that resulted strange bugs when initially missed.

OpenCL Versions

The OpenCL specification is unfortunately not very mature yet, with compatibility issues between hardware platforms and software versions being a constant problem. The OpenCL version of this application was initially developed for OpenCL 2.0, but then it was discovered that 2.0, as implemented by Intel, was only compatible with 5th generation and higher Core chips, significantly limiting its usability. The application was re-written for OpenCL 1.2.

Current Limitations:

Color Model

That mixing blue and yellow paint results in green is a fact that every preschooler even is sure to know by heart. It turns out, however, that modeling paint in a way where blue and yellow actually mix to make green is very challenging, and this system does not do it correctly. The current color representation model is a simple RGB based one whereas a more physically correct model must take into consideration additional physical properties of the paint. The details of how this works are described by Paul Kubelka Franz Munk in a paper titled “An Article on Optics of Paint Layers,” but the contents of that paper are by no means trivial, and an implementation of the Kubelka-Munk model proved too challenging for the scope of this project.
**Interface**

It is immediately clear that the application’s current interface was crudely and quickly-thrown together only for the purpose of testing the paint system implementation. In its current form the application is completely unusable for actual painting, requiring the use of the keyboard for even changing the color of the paint. Much interface consideration and coding would have to be done in order to turn this into a usable app.

**Fluid Simulation Approach**

The approach used for fluid simulation used here may not be the most optimum one for the task. While the stable, gird based approach definitely seems like the right technique, it is not the right type of simulation as it simulates the movement and distribution of particles within a fluid, whereas paint is really the movement of two fluids against one another—the paint and the air. Further research will be necessary to evaluate alternate techniques.

**Resolution**

The application is limited to a simulation resolution of 256 by 256 units. This is not adequate enough for large paintings.

**Canvas and Drying**

The application does not simulate a canvas and the paint never quite dries. It is fun to play with, but not useful for creating art.

**Future Work:**

Here is a list of areas where the application could be further developed in the future:

- Fixing the concurrent OpenCL branch: a rethinking and reimplementation of the simulation technique for heterogeneous computing should yield a much faster system than the current CPU-only branch.
- Increasing the resolution: certain optimizations could be made to allow the user to paint on a higher resolution canvas. Even if the simulation resolution cannot be increased, the simulation could perhaps be limited to the area around the brush while the rest of the canvas is not simulated. This would allow arbitrarily high resolutions, and it would look okay for thick paints that do not flow away too quickly from the brush.
- Improving the interface and painting tools: brush types, pressure sensitivity, actual graphical controls, etc.
- Correct, physically accurate color mixing (See. Limitations: Color Model).
- Adding a layering system. Currently there is a single layer for the entire simulation, so recently added paint has the same exact level of precedence as previously added paint. Being able to paint over previous paint completely would be much more realistic.

---
