Tilt Paint - A Viscous Fluid Simulator Over a Canvas

Rikin Gurditta
University of Toronto
Canada
rikin.gurditta@mail.utoronto.ca

Roya Shams-Zadeh-Amiri
University of Toronto
Canada
roya.shams.zadeh.amiri@mail.utoronto.ca

ABSTRACT

Fluid art painting is an abstract painting technique that uses layers of acrylic paint in a fluid consistency in order to create wavy or marbled results. In this project, we use the ideas behind particle-based viscous fluid simulation to create an algorithmic version of this painting technique. Our implementation aims to emulate paint particles on a flat canvas. When the canvas is tilted, gravitational forces pull the paint around in different directions. We evaluate our simulation behaviour and compare it with our expectations of real-life fluid art painting.

1 INTRODUCTION

One technique in fluid art painting is the "marbled" technique, achieved by placing strips of colour next to each other and rotating the canvas to disperse and mix the colours, while maintaining defined abstract lines. Relying on gravity, the paint travels to the edges of the canvas, "stretching" the mixed colours. As this is abstract art, the final result is interpreted by the artist when they determine when to stop rotating the canvas, by assessing whether or not the current result is aesthetically pleasing.

To simulate the ideas behind this technique, we used particle-based simulation of the incompressible Navier-Stokes equations. We assign each particle a colour, and place them across a grid to emulate paint at various 2D locations. We present controls to tilt the canvas using key presses, and a separate control to simulate the result once the desired canvas tilt is set. We offer fluid painting artists a prototyping technique to experimentally capture different results, giving them the ability to plan how the canvas is rotated, prior to using any paint.

2 METHODS

2.1 Setup

We begin by initializing our discretization parameters and modelling parameters. For discretization, we must decide on the grid size $nx \times ny$, the width of a grid cell Δx, the number of particles n, and a time step Δt. These parameters were chosen for visual style and feasible computation - the code is very slow with larger grids. Our physical parameters include the density of our fluid ρ, the volume of a particle V_p, and a gravity vector $\mathbf{g} = (0, -9.8, 0)$. Aside from gravity, these physical parameters are arbitrary.

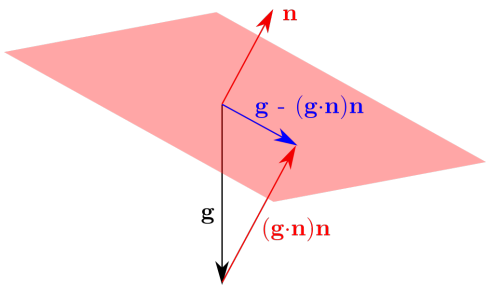
2.2 Advection

We implement our `advect_2d` function as outlined in the CSC417 lecture on fluid simulation[Levin 2020]. We utilize a simple forward Euler update to move particles around. We also clamp the particles to the edges of the canvas in order to prevent them from escaping our grid, as our pressure projection does not successfully ensure this.

We chose to implement a particle-based fluid simulation rather than a fully grid based one as presented in [Bridson 2015] because of how simple this advection step is. [Bridson 2015] presents a semi-Lagrangian backwards search technique using Runge-Kutta time intgeration in order to find and advect quantities defined on various grids. This is far more complicated than the particle strategy, which is a simple update of $\mathbf{x}^{t+1} \leftarrow \mathbf{x}^t + \Delta t \mathbf{v}^t$. However this does introduce its own small complications, as it necessitate the need for translation between particles and grids as outlined in section 2.6.

2.3 Gravity

The user is presented with options to rotate the grid using a `WASD` key configuration. In order to calculate the effect of gravity on the rotated grid, we project our gravity vector $\mathbf{g}$ onto it, then determine the grid-space coordinates of this projected gravity vector. To do this, we un-rotate the projected gravity and remove its z-component.



We keep track of the grid's rotation using a matrix $\mathbf{R}$. Let $\mathbf{n}$ be the current unit normal vector in space, and $\mathbf{P}_{2D}$ be a projection from 3D to 2D, then we can calculate the effect of gravity on the grid as:

$$\mathbf{P}_{2D}(\mathbf{R}^{-1}(\mathbf{g} - (\mathbf{g} \cdot \mathbf{n})\mathbf{n}))$$

In our implementation, we noticed that pressure projection has dominating effects to the gravity. To counteract these effects, we often used a stronger gravity vector, $10\mathbf{g}$ rather than $\mathbf{g}$, in order to increase gravity's influence on our results. With better simulation quality, we could replace this change of gravity with a manipulation of the physical parameters of our fluid. However this is not feasible with our unpredictable pressure code.

2.4 Pressure Projection

Real fluids are (more or less) incompressible, so our goal in this step is to ensure this for our simulated fluid. We do this by solving

the following PDE:

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{1}{\rho}\nabla p = 0 \tag{1}$$

$$\text{subject to } \nabla \cdot \mathbf{u} = 0 \tag{2}$$

where $\mathbf{u}$ is velocity, p is pressure, and ρ is the density of the fluid.

After discretizing equation (1) in time with a finite difference, taking the divergence of both sides, and applying the divergence free condition (2), we end up with the following PDE:

$$\nabla \cdot \mathbf{u}^t = \frac{\Delta t}{\rho}\nabla \cdot \nabla p \tag{3}$$

We can now discretize in space. The velocity $\mathbf{u}^t$ is stored as a dense vector representing flattened staggered grids. We will assume p to be stored the same way, so we can use sparse matrices for our calculus operations. We have assembled sparse matrices $\mathbf{D}$ and $\mathbf{B}$ so that $\mathbf{Df}/\Delta x = \nabla \mathbf{f}$ and $\mathbf{Bg}/\Delta x = \nabla \cdot \mathbf{g}$, for appropriate grid functions $\mathbf{f}$ and $\mathbf{g}$. We also have sparse matrices $\mathbf{P}$, $\mathbf{Q}$ to help us make grid sizes match up and help with boundary conditions. Using these operators, we can discretize equation (3) in space as well, thus fully discretizing it so that it may be solved for p with an out-of-the-box sparse system solver:

$$\mathbf{BPu}^t = \frac{\Delta t}{\rho \Delta x}\mathbf{BD}p \tag{4}$$

Typically, a conjugate gradient method solver is used, such as Eigen's `BiCGSTAB`. However, this was consistently unsuccessful for our system, so we reverted to Eigen's `SparseQR`. Lastly, we apply the velocity update as in equation (1):

$$\mathbf{u}^{t+1} \leftarrow \mathbf{u}^t - \frac{\Delta t}{\rho}\mathbf{QD}p$$

2.5 Naive Viscosity

Due to time constraints, we were not able to add a physics-based implementation of viscosity to our simulation. However, we did include a naive viscosity update, which has been used in hobby projects such as [Alex 2013]. This makes the fluid viscous by replacing each entry of each velocity grid with the one-ring average of its neighbours. Because the code does not depend on Δt, the effect is highly dependent on Δt. This smoothing is applied once every time step, so it will be applied more frequently when the simulation is run with smaller timesteps, causing greater viscosity.

2.6 Particles and Grids

In our advection step, we took advantage of each particle having its own position and velocity. In the pressure projection and viscosity steps, we took advantage of a global velocity grid defined over our simulation domain. These two representations are reconciled in our `particles_to_vel_grid_2d` and `interpolate_vel_2d` functions. The particle velocities are distributed to the grid using the method in [Bridson 2015], and the grid is interpolated to the particle velocities using regular bilinear interpolation.

3 RESULTS

To experimentally obtain our results, we tested our solution across three conditions across different rotations and simulation iterations. The first condition is the default "portrait" orientation, the second condition is a rotated "landscape" orientation, and the final condition called "multiple rotations" is a free form evaluation of how an artist would use this simulation for their purposes. We used parameters $nx = 9$, $ny = 11$, $n = 99$, $\Delta x = 0.5$, $\Delta t = 0.01$, $\rho = 0.1$, $V_p = 0.25$.

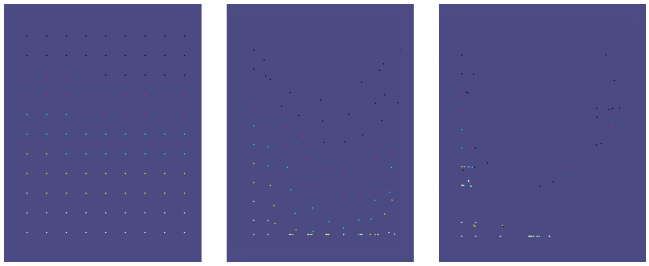


Figure 1: Default condition at iteration = 0 (left), iteration = 24 (center) iteration = 290 (right)

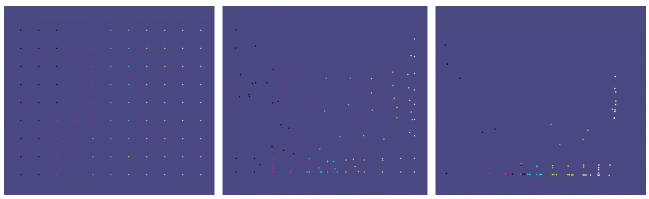


Figure 2: Rotated condition at iteration = 0 (left), iteration = 24 (center), iteration = 290 (right)

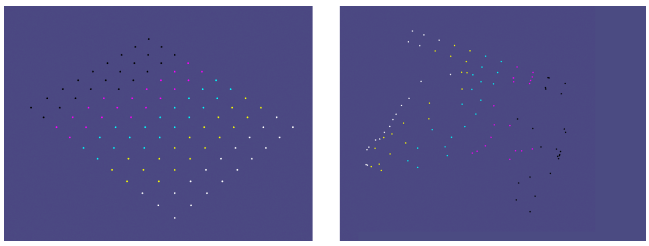


Figure 3: Mixed rotations with iteration = 0 (left), Marbled result with iteration = 42 (right)

4 EVALUATION

For the default portrait and landscape modes, it appears that the movement of the fluid at lower counts of simulation iterations, such as iteration = 24, is not too unrealistic. In real life we would expect the paint to fall to the edges of the canvas if held in one consistent orientation, and we would expect it to appear as if it is "melting". This behaviour in our simulation hints that the acceleration due to gravity is at least somewhat effective. However, at larger iteration counts such as iteration = 290, we obtain strange results as the paint particles collapse to the edges and leave the inside of the canvas sparse. The simulation seems to explode, as after a few iterations the particles gain large velocities. We suspect that

there are flaws in both the internal and boundary updates from the pressure projection. There are internal domain issues because particles that begin in the middle of the grid eventually explode outwards to the edges, and boundary issues because particles tend towards the same points on the boundaries after many iterations.

The pressure update is meant to prevent many particles from being in the same place by disallowing divergence, however that is exactly what is happening. It could be failing due to an incorrectly set up/solved linear system in equation (4), whether that be incorrect sparse matrices, inadequate accuracy from the QR decomposition based solver, or an incorrectly built velocity grid. The former two issues could potentially be sidestepped by using an iterative Gauss-Seidel solve as in [Stam 2003] (and many other fluid solvers) rather than an explicitly constructed sparse matrix solve. The latter issue could be sidestepped by using a fully grid-based approach rather than a particle-based one, as in [Bridson 2015], [Stam 2003], and most other simulators. The pressure update could also be incorrect due to inadequate simulation parameters. The computation is meant to zero out the divergence as computed over a single grid cell. The larger the grid cells, the less accurate their estimation of the divergence of the velocities of the particles. We could not find adequate grid cell sizes, however note that we did not conduct extensive trials as the running time of the simulation increases drastically with larger grid sizes.

Due to the unpredictable motion of the fluid it is difficult to evaluate whether or not our viscosity update is effective. The errors in the pressure projection cause very high velocities in all cases, so smoothing due to viscosity is difficult to measure.

REFERENCES

Dean Alex. 2013. *Javascript 2D Fluid / Turbulence Sim.* http://neuroid.co.uk/lab/fluid

Robert Bridson. 2015. *Fluid Simulation for Computer Graphics* (2nd. ed.). CRC Press, Boca Raton, FL.

David Levin. 2020. *Physics-based animation lecture 10: Fluid Simulation.* https://youtu.be/VddQZH_Ppd0

Jos Stam. 2003. Real Time Fluid Dynamics for Games. *Game Developers Conference* (2003). https://www.autodesk.com/research/publications/real-time-fluid-dynamics