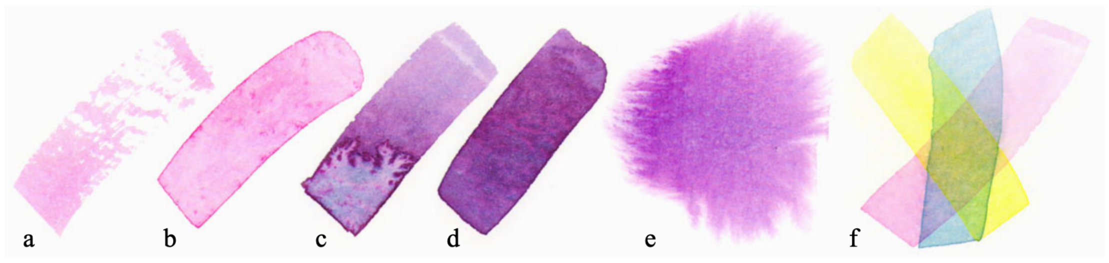
Kevin Chen
CPSC 679
Wednesday May 11

Simulating Physical Effects of Watercolor Painting

Final Project Writeup

Among drawing mediums to recreate digitally, watercolor is unique because of how it interacts with water and paper. Pigments flow and are carried by water and are affected by the amount of water. Paper texture and thickness also matter as capillary action diffuses water and pigments that are absorbed in paper. Watercolor changes and spreads over time, and its final image reveals its unique motion.

For my final project, I attempted to implement the watercolor simulation techniques described in Curtis et al. 1997, Computer-Generated Watercolor. The paper presents the following watercolor effects (a-f):



Watercolor effects: drybrush (a), edge darkening (b), backruns (c), granulation (d), flow effects (e), and glazing (f). From Curtis et al. 1997.

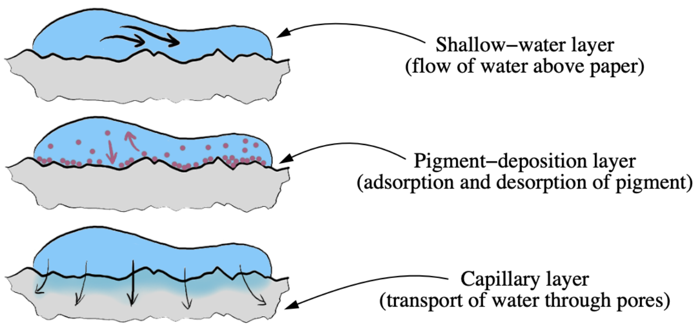
My paper implements a subset of the techniques described in the paper. Dry-brush effects were not implemented as it requires additional simulation of brush stroke textures and dynamics, and is best used with user interaction (out of scope of this project). I wanted to focus on the watercolor's unique fluid properties.

1. **Edge darkening** occurs when too much water and surface tension causes the pigment to flow and accumulate at the edges of the wet area.
2. **Backruns (a.k.a. blooms, splotchiness)** occur when excess water flows back into an uneven damp region and creates splotchy branching structures.
3. **Granulation** occurs when the natural properties of watercolor pigment cause particles to clump and settle unevenly.
4. **General flow effects** are the overall wet-on-wet effects of pigments flowing on textured paper. It creates the soft, feathery edges that flow in the direction of the water flow.
5. **Glazing** describes the color blending when multiple pigments mix or overlap.

I was able to partially recreate all the five effects described above with varying levels of success. While overall watercolor dynamics were composed of multiple components, I found that the implementation of general flow effects (wet-on-wet flows) contributed most to its realistic animation.

Implementation

I have implemented the three-layer model of simulating watercolors as described in Curtis et al. The layers are as follows: 1) the *shallow-water layer*, where water flows above the surface of the paper, 2) the *pigment-deposition layer*, where pigment is absorbed by/desorbed from the paper, and 3) the *capillary layer*, where water is absorbed via capillary action.


Shallow–water layer
(flow of water above paper)
Pigment–deposition layer
(adsorption and desorption of pigment)
Capillary layer
(transport of water through pores)

The three-layer model.

In watercolor painting, the texture of the paper itself affects the fluid flow and effects like backruns and granulation. The paper procedurally generates paper textures using pseudo-random processes; because I wanted to focus on implementing the flow of the watercolor itself rather than find myself in a rabbit hole of paper-texture generation, I skip this step. Paper textures are stored as grayscale .ppm images, which my program then reads in.

Overall more time was spent setting up data structures than in implementing the techniques from the paper. I implemented a custom staggered grid class, StaggeredGrid.h/cpp, which allows getting/setting in half (+/- 0.5) grid increments. I also implemented fast Gaussian blur functions which approximate a Gaussian blur via three linear-time box blur operations. While not currently editable, pigment information is stored in a vector to allow for user-definable colors in the future.

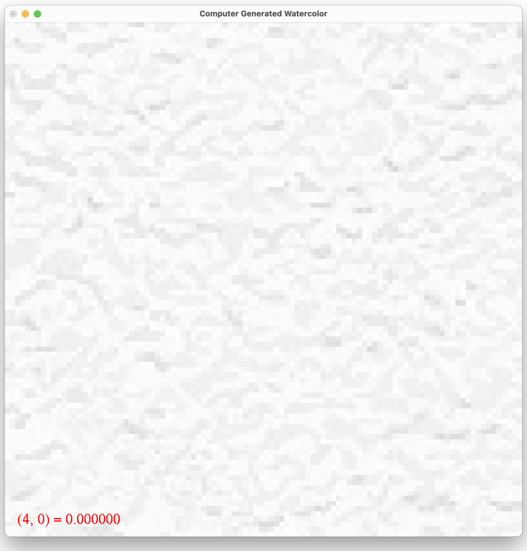
To simulate the three layers and pigment movement between them, my program's function structure matches Curtis et al's. The functions are as follows:

1. **MoveWater()** Simulates the shallow-water layer and updates water velocities. Velocities are represented in two staggered grids: one in the x direction and in the y direction. We solve forward integration using Euler's method, and dynamically set time step depending on the velocity.
2. **MovePigment()** Simulates pigment movement in the shallow-water layer. Pigment is moved by the velocity of the water.
3. **TransferPigment()** Simulates the adsorption and desorption of pigments with the paper. This also achieves granulation effects.
4. **SimulateCapillaryFlow()** Simulates backruns and capillary flow effects, allowing water to be diffused through the paper.

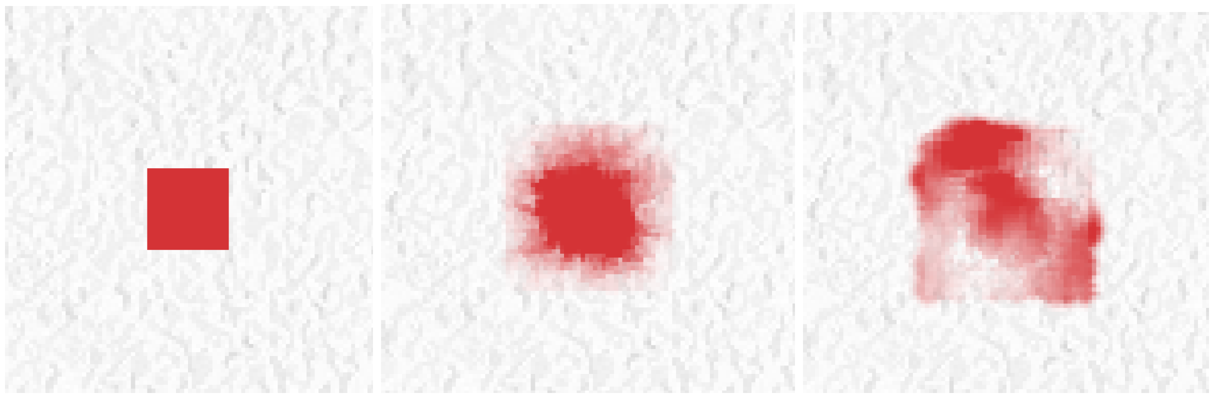
Results

First, paper textures can be uploaded and specified. Paper textures affect the flow of the watercolor.

Computer Generated Watercolor
(99, 68) = 0.000000

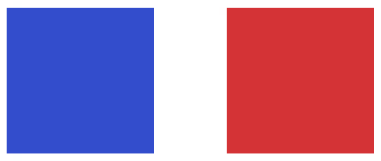


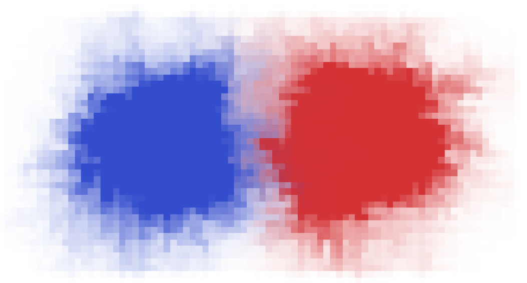
Below, we set a square of red pigment in the center with the wet area defined as a larger invisible square surrounding it. The images below are three frames taken from a full simulation (video link provided). General flow effects spread the pigment out, and in the second image we see that the pigment has begun to reach the boundaries of the wet area. In the third image, we observe pigment pooling at the edges of the wet area (edge darkening). Some granulation is achieved; the red pigment spreads unevenly and leaves splotchy traces. Capillary effects cause the pigment to spread slightly beyond the defined wet area, although not quite realistically. Flow effects and blooming can be seen in the full video.

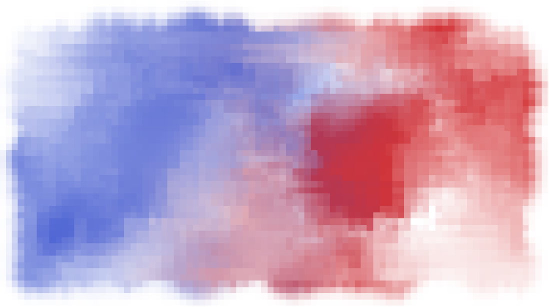


https://www.youtube.com/watch?v=ic56LUOM6j0

Next, we demonstrate two differently-colored pigments interacting with each other. The two colors here are blended by simple linear interpolation (this works, but it's not physically accurate—we would have to implement the KM model). Where the blue and red pigment touch forms a murky purple, whereas in reality, it would form a similarly-saturated purple. The future work section discusses the KM model, a more realistic color model; I did not implement that in this project. Note that the paper texture is not shown, this is for clarity.

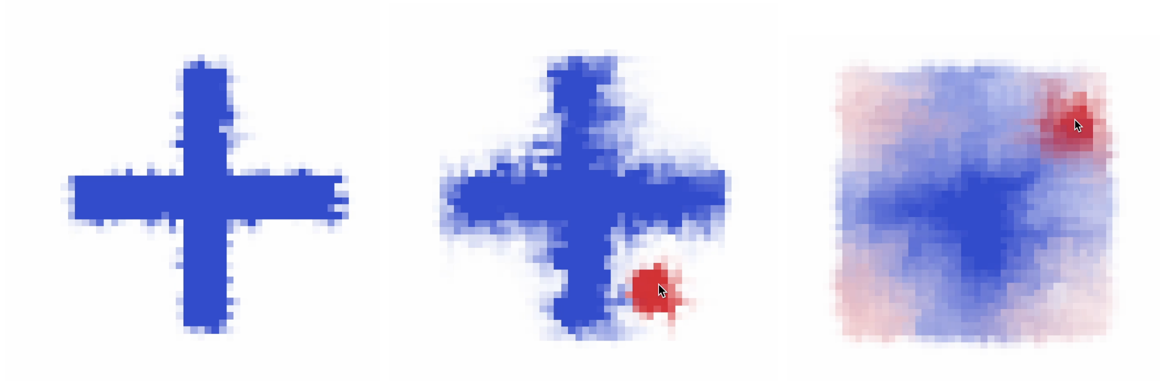






https://www.youtube.com/watch?v=JjEhPoaxUZs

Everything we're seeing is running in real-time. Let's add some basic user interaction! The next video starts with a "t" of blue pigment, and the user can click to add a small amount of red pigment, which gets dispersed and mixed with the blue.



https://www.youtube.com/watch?v=1ohEUe2xzrA

Limitations

Overall, I am pretty happy with the results I was able to achieve. However, I was unable to get capillary effects fully working, and I was unable to debug it not working (it had something to do with my forward integration shooting values to infinity—classic). I believe this was mostly due to the paper's lack of clarity. Although seemingly-detailed pseudocode of the functions were provided, there were glaring omissions: the capillary layer procedure (SimulateCapillaryFlow) introduced many constants (absorption/diffusion rate, carrying capacity, etc) which the paper never specified; this meant I had to guess these constants without understanding how they affected the simulation. I also had to double check with a patent I found of the same procedure (https://patents.google.com/patent/US6198489B1/en) as there were discrepancies even in implementation details, such as flipped signs, conflicting variable names, and different function calling structures. Because of these difficulties, I was not able to get my program to work at the same level as the paper's. This is why the recorded demos in the results section don't look perfect—the simulation would often slow down after running for too long.

Future Work

There are three main parts of the project that can be improved with future work, besides fixing the capillary layer.

First, general efficiency improvements. Access patterns should be studied and data structures may need to be refactored to improve array lookup times. Currently, pigments details are stored in a struct; each struct also contains arrays which represent pigment concentration. There are many instances in the program where it loop through the multiple pigments to render a single pixel of the paper; to reduce cache misses, it would be far better to solve for one pigment at a time and combine the colors in a separate pass. Overall, it would benefit this simulation if we refactored the code for better data read/writes.

Second, I could explore better integration methods that would not have to deal with hacks such as dynamically adjusting the timestamp based on the max velocity. Specifically, it may be worth implementing a backwards Euler solver rather than the current forward solver.

Third, I could implement the Kubelka-Munk (K-M) model of combining multiple pigments. The KM model better models the physical properties of pigments. For example, in most drawing applications, blending blue and yellow results in a muddy gray; in real life, however, blue and yellow blend to create a similarly-saturated green color. In the KM model, rendering depends on each pigments' absorption and scattering coefficients, which are functions of wavelength. Implementing the KM model would enable the next step in rendering accurate, colorful, watercolors.

Sources Cited

Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. 1997. Computer-generated watercolor. In Proceedings of the 24th annual conference on Computer graphics and interactive techniques (SIGGRAPH '97). ACM Press/Addison-Wesley Publishing Co., USA, 421–430. DOI:https://doi.org/10.1145/258734.258896

Šárka Sochorová and Ondřej Jamriška. 2021. Practical pigment mixing for digital painting. ACM Trans. Graph. 40, 6, Article 234 (December 2021), 11 pages. DOI:https://doi.org/10.1145/3478513.3480549