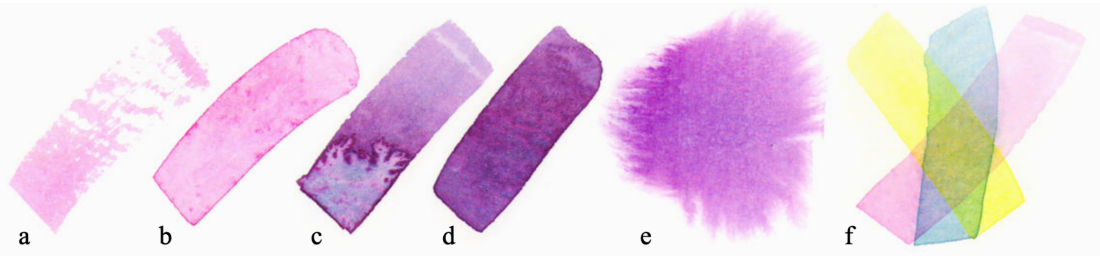
Kevin Chen
CPSC 679
Wednesday May 11

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, its final image revealing its unique motion.

For my final project, I implemented the watercolor simulation techniques described in Curtis et al. 1997, Computer-Generated Watercolor. The paper implements all the watercolor effects (a-f) in the image below.



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

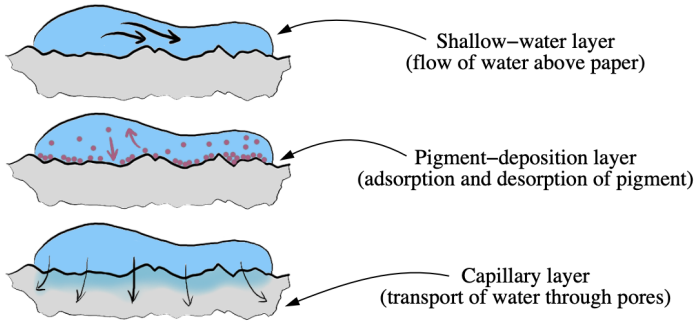
My paper implements a subset of the techniques described in the paper. Specifically, I have implemented the following watercolor effects. Dry-brush effects were not implemented because that involves specifics on user interaction with simulated brush strokes. I only wanted to focus on the movement of watercolor.

1. **Edge darkening.** This occurs when too much water causes the pigment to flow and accumulate at the edges of the wet area.
2. **Backruns/blooms.** This is when Excess water flows back into a damp region, and creates intricate tree-like structures.
3. **Granulation.** Due to the natural properties of watercolor pigment, particles clump and settle unevenly.
4. **General flow effects.** General wet-on-wet effects of pigments flowing on paper due to changes in texture.
5. **Glazing.** Color blending when multiple pigments mix or overlap.

I have achieved, for the most part, all the goals I had initially set. Overall, the goals I had set for myself were well-paced and actionable. While it took many components (the implementation will be described below) to achieve my final results, I found that the general flow effects (wet-on-wet flows) contributed most to the realistic animation of watercolor flow.

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 by capillary action.



The three-layer model.

In watercolor, the texture of the paper itself affects the fluid flow and effects like backrun and granulation. The researchers in the paper procedurally generated paper textures using pseudo-random processes. However, for my implementation, because I wanted to focus on simulating the flow of the watercolor itself rather than in accurate paper-texture generation, I simply store paper height maps as .ppm images, that my program can then read.

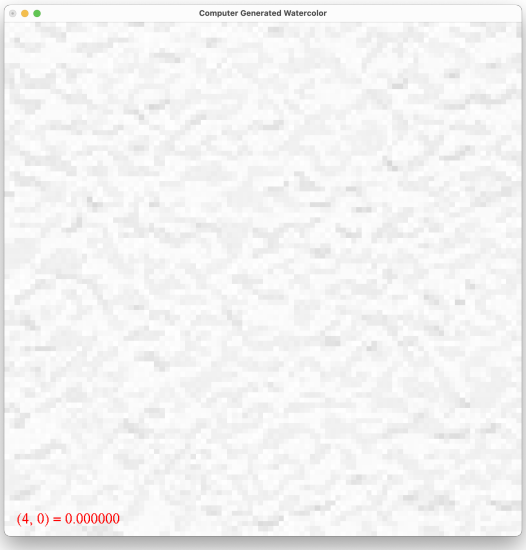
Overall, the most time spent was not implementing the procedures, but setting up the infrastructure in terms of data structures. For my project, I have implemented a custom staggered grid class, StaggeredGrid.h/cpp, which allows getting and setting in half (+/- 0.5) grid increments. I have also implemented fast Gaussian blur functions that approximate by taking three linear-time box blur operations. I used the read and write PPM functions from our previous assignments and cs478 to read custom PPMs as paper height maps. I implemented a struct to store data on each pigment; the pigments are stored in a vector to allow in the future for any number of user-definable colors.

To simulate the three layers of flow and pigment movement, my implementation matches the one described in Curtis et al. The procedure is as follows:

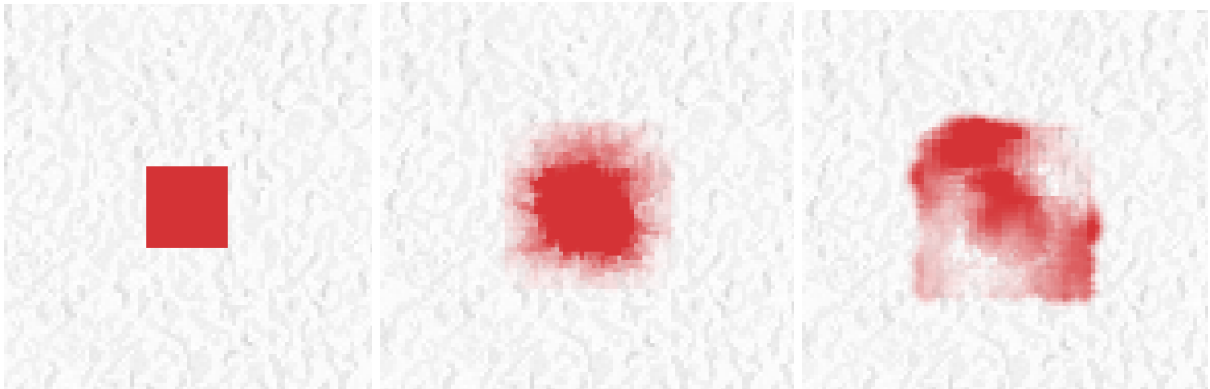
1. **MoveWater()** Simulates the shallow-water layer, and updates water velocities. Velocities are represented as two staggered grids, one for x-velocities and one for y-velocities. We solve forward using Euler's Method, and reduce the time step if the velocity is too high.
2. **MovePigment()** Simulates pigment movement in the shallow-water layer. Pigment is distributed according to the water velocities.
3. **TransferPigment()** Simulates 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, different paper textures can be uploaded and specified. These paper textures affect the flow of the watercolor.

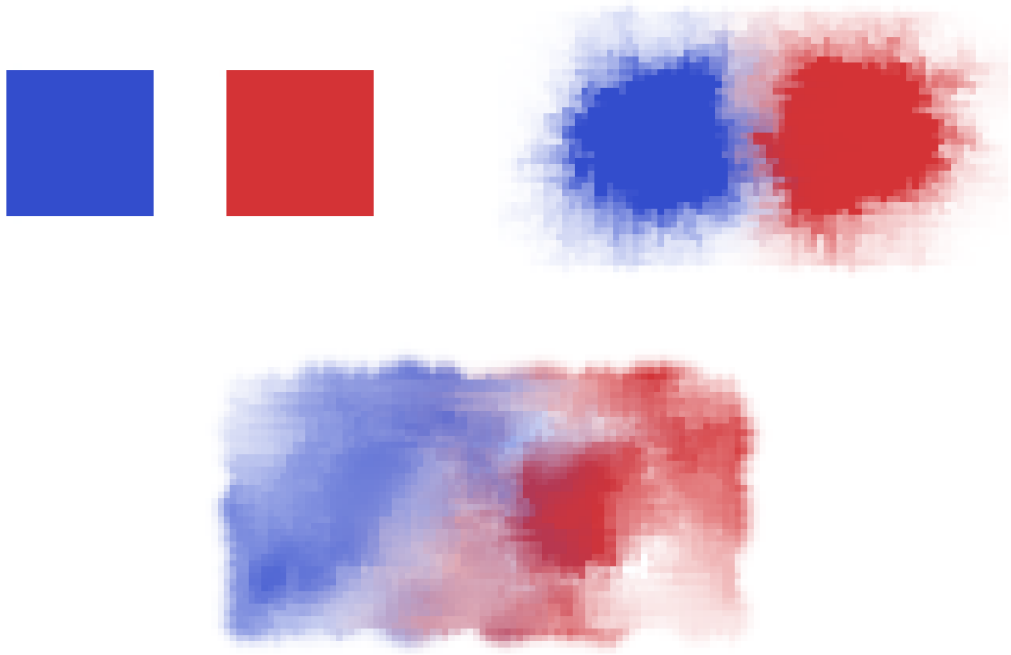


Below, we set a square of red pigment in the center, with a wet area defined as a larger square around it. The images are three frames taken from the simulation. General flow effects spread the pigment out, and in the second frame below we see the rough borders of the wet area. In the third frame we can observe the effects mentioned above. Pigment pools at the edges of the wet area to simulate edge darkening, as our implementation does not specify the amount of water, just that there is water. Some granulation is seen as the red pigment spreads unevenly. Capillary effects cause the pigment to spread beyond the defined wet area. Flow effects and blooms can be seen in the video that these frames were taken from.



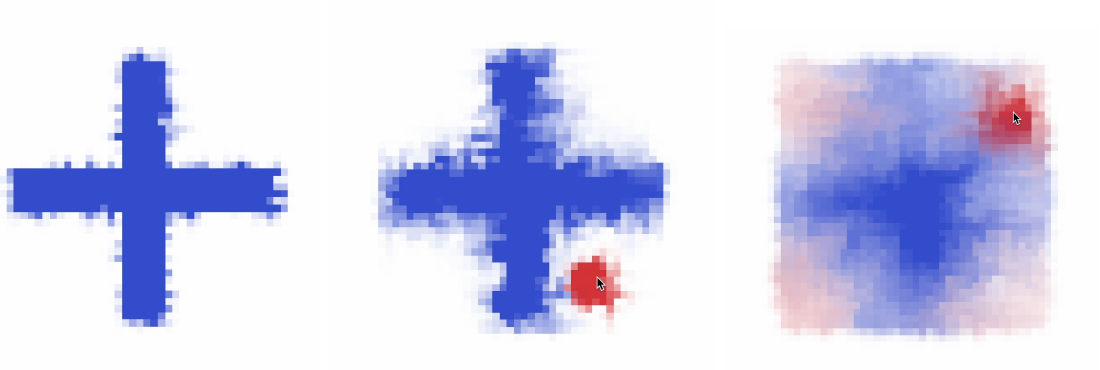
https://www.youtube.com/watch?v=ic56LUOM6j0

Next we demonstrate two pigments interacting with each other. Again, we see all the watercolor effects. Multiple colors are blended by simple linear interpolation, so blending is not fully accurate. We can see the blue and red pigment touch to form a murky purple in the center (in reality, it would form a vibrant purple). In the future work section, I detail a more realistic color model that I did not implement in this project. Note that I do not render the paper texture in this simulation and the one after; this is so we can better focus on just the flow of the pigment.



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

So far, the videos all show the simulation running without any speed-ups. Therefore, it is possible to add basic user interaction into the program. The following shows a starting amount of blue pigment, with the user then adding small squares of red pigment that get dispersed and mixed.



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

Overall, I am very happy with the results I was able to achieve with my program. The journey to getting here was not without roadblocks, though. The issues I struggled with were, surprisingly, implementing the procedures themselves. While they are listed in seemingly great detail in the paper, I found many parts lacking and some constants simply wrong. In fact, I had to double check with the patent I found (https://patents.google.com/patent/US6198489B1/en) as it had slight discrepancies in implementation details (i.e. flipped signs, different variable naming, different function ordering). Also, the capillary layer procedure involved the most constants (for absorption, diffusion, carrying capacity, etc), none of which were specified in the paper. It was mostly trial and error with tweaking these values (and part understanding how these actually affected the simulation) until I achieved visual results that were adequate.

Future Work

There are three main parts of the project that can be improved with future work.

First, general efficiency improvements. Access patterns should be studied and data structures may need to be refactored to improve array lookup times. Currently, pigments are each a struct which contain arrays that represent pigment concentration on the paper field. There are many instances in my program where I loop through the multiple pigments to change just one coordinate of the paper; this obviously would result in multiple cache misses; it would be better to solve for one pigment at a time. Overall, restructuring the code to better flow around the read/writes of data would improve efficiency.

Second, I could explore better integration methods that would not have to deal with hacks such as dynamically adjusting the timestamp. 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, 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