Artist-Centered Digital Watercolor: Water Flow, Pigment Diffusion, Optical Glazing, and Wet-in-Wet in Real Time

Daeyoung Kim
University of California, Berkeley EECS
Berkeley, CA, USA
daeyoungkim@berkeley.edu

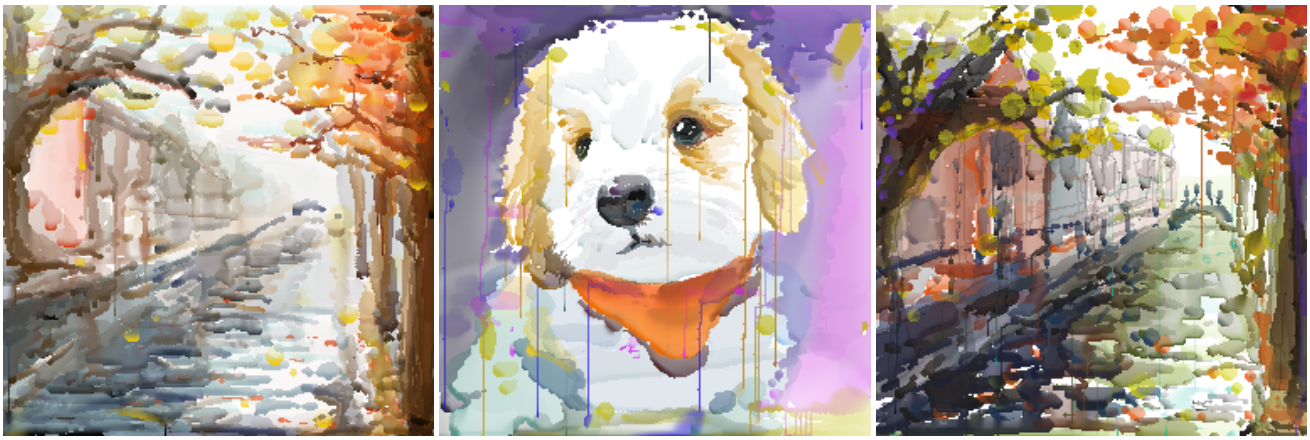

Figure 1: Paintings created with the watercolor simulation tool. Each was painted in real time using a pressure-sensitive brush and GPU-based physics.

Abstract

The motivation for this watercolor simulation is a desire to provide a rich watercolor experience digitally. Many common painting tools are built around opaque, relatively immobile media such as oil or acrylic paint. Water based medium digital tools are uncommon due to its computational complexity and difficulty of medium itself. The computational complexity involves water fluid simulation, pigment diffusion, pigment deposition in the paper all happening in real time. The fluid simulation takes gravity, surface tension, spreading force and momentum from previous frames into account. The coefficients are hand-tuned based on my personal watercolor experience as a former artist. Beer-Lambert optical mixing is used for multiple layering. This simulation is implemented using an open-source game engine Godot.

Keywords

watercolor simulation, pigment diffusion, wet-in-wet, optical glazing, real-time graphics, human–computer interaction

1 Introduction

The main challenge of digital watercolor is that pigments on the canvas constantly move and interact, even without direct user input. This dynamic behavior is essential to what makes watercolor unique, but it also makes it computationally heavy. Traditional CPU-based approaches quickly run into performance limits, as simulating fluid movement and pigment mixing across a large canvas requires massive computation every frame.

Parallel computation on the GPU is a key to making large-scale watercolor simulation possible in real time, allowing users to paint naturally and see the pigment behavior immediately, much like working with real watercolor.

Another major challenge, from an engineering perspective, is understanding the artistic side of the simulation. Adjusting physical parameters without a good sense of how watercolor behaves in the real world can easily make the simulation feel artificial. My background as both an artist and a developer helped me bridge that gap. I could interpret how real paint should look and behave, and then adjust the simulation accordingly.

2 Related Work

My simulation design was mainly inspired by the early digital watercolor model introduced by David Small [2]. His system demonstrated how a cellular automaton could be used to simulate the movement of water and pigment across paper fibers. From this foundational idea, I adopted several important principles, including force-based displacement, the separation between mobile and static pigment layers, and the role of paper absorbency in controlling pigment deposition.

Curtis et al. [1] presented a comprehensive watercolor framework that introduced the three-layer conceptual model—water, mobile pigment, and static pigment—and applied the Kubelka–Munk optical model to achieve realistic glazing effects. My implementation follows a similar structural idea but replaces the Kubelka–Munk

model with the simpler Beer–Lambert absorption law, which produces comparable translucency effects with much lower computational cost.

Xu et al. [5] expanded on pigment–paper interactions by modeling adsorption and desorption between pigment and fibers. I implemented a simplified version of this mechanism, where the deposition rate depends on the local water content and absorbency. Although I did not reproduce the full adsorption isotherm equations, my approach was guided by their conceptual model of pigment–water exchange.

Van Laerhoven and Van Reeth [4] explored real-time watercolor simulation with a focus on achieving interactive performance, targeting frame rates of around 60 FPS at 256×256 resolution. Their work motivated my decision to implement all physics calculations on the GPU, allowing large-scale watercolor behavior to run interactively while preserving the three-layer pigment model and deposition logic.

All implementations in this work were developed using the open-source Godot Engine 4.4.1 [3], which provided both the rendering framework and GPU compute pipeline for real-time simulation.

3 System Overview

The simulation uses a three-layer model for watercolor: a **water** layer for surface water amount, a **mobile pigment** layer for pigment that is still suspended in water, and a **static pigment** layer for pigment that has deposited into the paper. Each layer is *double buffered* (read/write) to avoid read–write hazards on the GPU.

Water dynamics. The water layer computes surface flow per frame based on gravity (tilt), surface tension, a local spreading term, and a simple momentum carry-over from the previous frame. These forces are evaluated in a compute pass that writes per-pixel displacement used by later stages.

Pigment transport. The mobile pigment layer advects pigment according to the water displacement field. In the same stage, I apply diffusion (to soften high-frequency pigment variation) and handle deposition into the static layer as a function of local water amount and paper absorbency. The static layer stores the finalized pigment at each pixel and is only modified by deposition (and occasional lifting when using the removal tool).

Rendering and optics. For compositing and glazing effects, I use the Beer–Lambert absorption law to approximate optical attenuation through layered pigment. This gives a simple and stable way to reproduce watercolor translucency without the full complexity of Kubelka–Munk.

GPU pipeline. All simulation steps run as GLSL compute passes on double-buffered textures. Each frame executes: (1) evaporation, (2) water displacement (gravity, surface tension, spreading), (3) advection of water and mobile pigment with inflow/momentum recording, (4) surface diffusion on the wet area, and (5) deposition from mobile to static pigment, with buffer swaps between stages.

Interactivity. User input is captured as brush dabs on a CPU preview layer, then uploaded to the GPU in small batches to keep interaction responsive. Brush pressure controls radius and strength (for removal). The system targets real-time feedback on a modest canvas while prioritizing stability and visual behavior over raw speed.

Overall, the design aims to balance physical plausibility with practical constraints, keeping the model simple enough to run interactively while still producing the key watercolor cues (flow, glazing, soft diffusion, and gradual deposition).

4 Simulation Design

4.1 Notation and Data Structures

The watercolor canvas is represented as a 256×256 grid. Each pixel stores three main values (higher resolutions can be used depending on GPU performance):

- $w(x, y, t)$ — the amount of surface water at pixel (x, y) at time t
- $p_m(x, y, t)$ — the mobile pigment suspended in water (RGBA)
- $p_s(x, y, t)$ — the static pigment that has settled into the paper

All layers are *double-buffered* to avoid read–write conflicts. Each layer has a read buffer B^t and a write buffer B^{t+1}, and the two are swapped after every simulation pass.

4.2 Water Dynamics

4.2.1 Force Calculation. Each frame, forces are calculated per pixel to determine how surface water moves. The simulation handles four directions (right, left, down, up) separately and then combines them in the inflow step.

- **Gravity (per direction).** The user can tilt the canvas, and this controls the direction of gravity. For a tilt angle θ, the force strength depends on $\sin(\theta)$:

$$F_g(x, y) = w(x, y)\, g \sin(\theta),$$

  applied toward the tilt direction.
- **Surface tension (adaptive lookahead).** Water tends to pull itself toward wetter areas. For each direction, the system looks ahead up to ten pixels (stopping at dry boundaries or the edge of the canvas). The tension force to the right is computed as:

$$F_s^{\text{right}}(x, y) = S\,(\overline{w}_{\text{right}} - \overline{w}_{\text{left}}),$$

  where $\overline{w}_{\text{right}}$ and $\overline{w}_{\text{left}}$ are average water amounts over the next and previous ten pixels, respectively. The same idea is applied for the other three directions.
- **Spreading (immediate neighbor).** This term pushes water from higher to lower regions based on the closest neighbor. For the right direction:

$$F_{sp}^{\text{right}}(x, y) = SP\,(w(x+1, y) - w(x, y)),$$

  and similar for left, up, and down.

The displacement used by the inflow step is the sum of these terms, clamped to positive values:

$$d^{\text{dir}}(x, y) = \max(0,\ F_g^{\text{dir}} + F_s^{\text{dir}} + F_{sp}^{\text{dir}}).$$

4.2.2 Inflow Model with Momentum Dampening. The inflow step moves water based on the displacement map computed above. It also uses momentum dampening to prevent water from oscillating back and forth too much. Each pixel keeps an inertia record $m^t(x, y)$

that stores how much water came in from each direction during the previous frame:

$$m^t(x,y) = (m_R^t,\ m_L^t,\ m_D^t,\ m_U^t).$$

Outflow calculation. Each pixel first decides how much water it wants to send out in each direction:

$$\text{want}^{\text{dir}}(x,y) = d^{\text{dir}}(x,y)\, w^t_{\text{movable}}(x,y)\, \Delta t.$$

Momentum dampening. Before sending out water, the simulation adjusts each direction based on the previous frame's inflow memory: - If the pixel just received water from the same side, it applies *canceling_power* (γ) to slow down the bounce-back. - If the water previously came from the opposite side, it applies *acceleration_power* (α) to keep the flow going in that direction.

For example, for the rightward direction:

$$\text{want}'_R = \begin{cases} \text{want}_R - \gamma \cdot \min(\text{want}_R, m_R^t) & \text{if inflow from right (cancel)} \\ \text{want}_R + \alpha \cdot \min(\text{want}_R, m_L^t) & \text{if inflow from left (accelerate)} \\ \text{want}_R & \text{otherwise.} \end{cases}$$

Canceling takes priority over acceleration. The same logic applies to left, up, and down directions.

Conservation. After dampening, if the total desired outflow exceeds the available water, all directions are scaled down proportionally:

$$\text{outflow}^{\text{dir}} = \text{want}^{\text{dir}\prime} \cdot \min\left(1,\ \frac{w^t_{\text{movable}}(x,y)}{\text{total}'_{\text{want}}}\right)$$

Water update.

$$\Delta w_{\text{out}}(x,y) = \sum_{\text{dir}} \text{outflow}^{\text{dir}}(x,y),$$

$$\Delta w_{\text{in}}(x,y) = \sum_{\text{nbr}} \text{outflow}^{\text{opp(dir)}}(x_{\text{nbr}}, y_{\text{nbr}}),$$

$$w^{t+1}(x,y) = w^t(x,y) - \Delta w_{\text{out}}(x,y) + \Delta w_{\text{in}}(x,y).$$

Inertia update. Finally, the new inflows are stored for use in the next frame:

$$m^{t+1}(x,y) = (m_R^{t+1},\ m_L^{t+1},\ m_D^{t+1},\ m_U^{t+1}),$$

where each value represents how much water was received from that side.

4.2.3 Evaporation. Water gradually evaporates over time. The rate depends on how exposed a pixel is compared to its neighbors. I estimate the exposed area from local height differences:

$$A_{\text{exposed}}(x,y) = 0.1 + \sum_{n \in \{\text{neighbors}\}} \max\big(0,\ w^t(x,y) - w^t(n)\big).$$

The small base term (0.1) guarantees a minimum drying rate even in flat regions (prevents water from lingering forever), while the neighbor differences make protruding or edge pixels dry a bit faster than interior pixels.

The update is:

$$w^{t+1}(x,y) = \max\big(0,\ w^t(x,y) - A_{\text{exposed}}(x,y)\, k_{\text{evap}}\, \Delta t\big).$$

Notes. (1) This is simple to compute on the GPU (few texture reads, early-outs on dry pixels). (2) The same formulation works well with my momentum/inflow scheme and reduces oscillation. (3) The evaporation boost can be temporarily increased from the UI for quick tests or drying (Q key).

Edge effect. Because boundary pixels are more "exposed," they evaporate a little faster than the interior. In practice, that gently concentrates pigment near the stroke perimeter and, as deposition progresses, leaves a slightly darker edge. This small asymmetry was intentional—it reproduces the subtle outline that often appears in real watercolor as a wash dries.

Figure 2: Brush strokes at different water amounts

4.3 Pigment Transport and Deposition

4.3.1 Pigment Transport with Water Flow. Pigment moves together with water during the inflow step. When water flows from one pixel to another, it carries pigment proportionally based on three factors:

(1) **Movable pigment fraction:** Only the pigment suspended in movable water (not absorbed into the paper) can flow. If a pixel has water amount w with capacity c, then movable water is $w_{\text{movable}} = \max(0, w - c)$. The movable pigment mass is:

$$m_{\text{movable}} = m_{\text{total}} \cdot \frac{w_{\text{movable}}}{w}$$

(2) **Outflow fraction:** If total outflow is Δw_{out} and movable water is w_{movable}, the fraction of movable pigment that leaves is:

$$f_{\text{out}} = \frac{\Delta w_{\text{out}}}{w_{\text{movable}}}$$

(3) **Pigment carried:** The incoming pigment mass at a receiving pixel is:

$$m_{\text{incoming}} = m_{\text{movable,neighbor}} \cdot \frac{\Delta w_{\text{in}}}{w_{\text{movable,neighbor}}}$$

Pigment from multiple inflow directions is mixed optically using the Beer–Lambert absorption model (see Optical Compositing section below).

4.3.2 Deposition. Mobile pigment gradually deposits into the static layer based on water content and paper absorbency. The deposition rate is calculated as:

$$k_{\text{dep}}(x,y) = k_{\text{base}} \cdot A(x,y) \cdot \left(\frac{w_{\text{scale}}}{w(x,y) + w_{\text{scale}}}\right)^2$$

where $A(x,y)$ is the local paper absorbency and w_{scale} controls how much water feels "wet." The squared term makes deposition much

slower when there's lots of water, mimicking how real watercolor stays workable longer on a wet surface.

The fraction of mobile pigment mass that deposits per frame is:

$$f_{\text{deposit}} = 1 - e^{-k_{\text{dep}} \cdot \Delta t}$$

If the mobile layer has pigment mass m_{mobile}, the deposited mass is:

$$\Delta m_s = m_{\text{mobile}} \cdot f_{\text{deposit}}$$

4.3.3 Diffusion. Pigment spreads laterally along the wet surface based on concentration gradients and water contact area. For each neighbor pair, the contact area is:

$$A_{\text{contact}} = \min(w(x, y), w(x_{\text{nbr}}, y_{\text{nbr}}))$$

The diffusion flux is driven by the mass gradient:

$$\text{flux} = D \cdot A_{\text{contact}} \cdot (m_{\text{neighbor}} - m_{\text{here}})$$

where D is the diffusion rate. Positive flux means pigment flows toward the current pixel. To prevent excessive spreading, each transfer is capped at a fraction (`diffusion_limiter`) of the source pixel's mass. This creates soft, natural gradients while keeping the pigment somewhat localized.

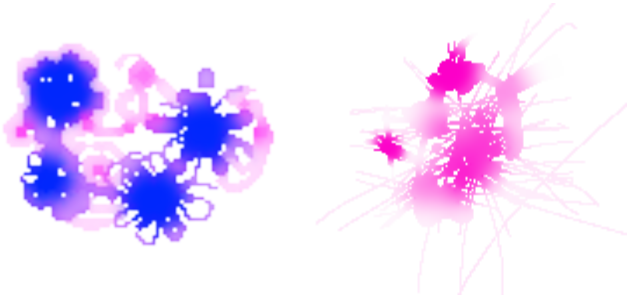

Figure 3: Diffusion example

4.4 Optical Compositing

4.4.1 Beer–Lambert Absorption Model. Instead of treating alpha as a direct opacity value, the simulation converts between alpha and pigment *mass* using the Beer–Lambert absorption law. This captures a key property of real watercolor: you need more pigment than you'd expect to create intense color, because pigment concentration affects opacity exponentially, not linearly.

The conversion formulas are:

$$\text{mass} = -\frac{\ln(1-\alpha)}{K_{\text{abs}}}, \quad \alpha = 1 - e^{-K_{\text{abs}} \cdot \text{mass}}$$

where $K_{\text{abs}} = 0.5$ is the absorption constant. When mixing two pigments optically (e.g., during inflow or diffusion):

(1) Convert both alphas to mass
(2) Add masses: $m_{\text{total}} = m_1 + m_2$
(3) Weight hues by mass: $\text{hue}_{\text{mix}} = \frac{m_1 \cdot \text{hue}_1 + m_2 \cdot \text{hue}_2}{m_{\text{total}}}$
(4) Convert total mass back to alpha: $\alpha_{\text{mix}} = 1 - e^{-K_{\text{abs}} \cdot m_{\text{total}}}$

This approach gives gradual color buildup that feels closer to real watercolor glazing.

4.4.2 Layer Compositing Order. The final image is rendered from back to front:

(1) White paper background
(2) Static pigment layer (deposited, permanent)
(3) Mobile pigment layer (wet, suspended pigment)
(4) Water visualization layer (optional blue tint, toggled with Tab key)
(5) Pencil layer (top, opaque blending)

4.5 GPU Pipeline

4.5.1 Compute Shader Dispatch. All physics steps run as GLSL compute shaders dispatched with 8×8 threads per work group. For the 256×256 canvas, this means $(32, 32, 1)$ work groups total. Each thread processes one pixel using `gl_GlobalInvocationID.xy`.

4.5.2 Texture Format. All layers use `RenderingDevice.DATA_FORMAT_R32G32B32A32_SFLOAT` (32-bit floats per channel):

- **Water:** R = water amount, GBA unused
- **Pigment (mobile/static):** RGBA = color with alpha as concentration
- **Displacement:** RGBA = forces (right, left, down, up)
- **Inertia:** RGBA = inflows from (right, left, down, up)

4.5.3 Frame Pipeline. Each frame executes these steps in order:

(1) **Evaporation:** `evaporation.glsl` reduces water based on exposed surface area
(2) **Displacement:** `calculate_displacement.glsl` computes directional forces
(3) **Inflow:** `apply_inflow.glsl` moves water and pigment, applies momentum dampening, records inertia
(4) **Diffusion:** `diffusion.glsl` spreads mobile pigment along wet surface
(5) **Deposition:** `deposition.glsl` transfers pigment from mobile to static layer
(6) **Buffer swap:** Read/write texture pairs swap for next frame

Parameters like S, SP, γ, k_{evap}, etc. are passed as push constants each frame. The entire simulation runs on the GPU with no CPU readback during painting.

4.6 User Interaction

4.6.1 Brush System. Brush strokes are previewed immediately on the CPU using Beer–Lambert compositing for instant feedback. Each stroke maintains a mask to prevent double-painting the same pixel. After a short interval, accumulated dabs are batched and uploaded to the GPU via `add_paint.glsl`.

Tablet pressure controls brush radius between $0.1r_{\text{base}}$ and $2.0r_{\text{base}}$. The system falls back to full pressure (1.0) for mouse input.

4.6.2 Interactive Controls. The interface supports real-time parameter adjustment and canvas manipulation:

- **Settings window:** Sliders for all physics parameters (S, SP, γ, k_{evap}, etc.). Changes apply immediately via GPU push constants without shader recompilation.
- **WASD keys:** Tilt canvas to adjust gravity direction during painting
- **Q key:** Temporarily boost evaporation for quick drying

- **Tab key:** Toggle water layer visibility (shows only the water amount as a blue overlay)

4.7 Stability and Boundaries

To keep the simulation stable:

- Dry pixels ($w < 0.0001$) require a minimum force ($F_{\text{hold}} = 5.0$) to become wet, preventing unwanted spreading
- Water and pigment amounts are clamped to $[0, w_{\max}]$ to avoid overflow
- Canvas edges act as no-flow boundaries (forces don't propagate beyond the edge)

4.8 Performance Considerations

To keep interaction responsive, the implementation uses: (1) batched uploads, (2) early exits on dry pixels, and (3) double buffering to avoid read–write stalls. See Section 5.1 for runtime measurements.

5 Results (Prototype)

5.1 Runtime Performance

All observations are qualitative. Tests were done on an Intel Core i5-14400 with integrated graphics (no discrete GPU). At 256×256, the system feels real-time in normal use.

- **Brush responsiveness.** User input is handled immediately; brush strokes feel responsive and natural.
- **Upload batching.** Batching stroke uploads was key for smooth interaction during fast strokes. The CPU preview shows dabs right away, and the GPU catches up smoothly as batches arrive.
- **Water blobs under tilt.** When the canvas is tilted, strokes naturally form water blobs. This is intentional and matches the way washes pool in real watercolor.
- **Hold threshold and dripping.** Once the local force reaches the `HOLD_THRESHOLD`, those blobs break and start to drip downward. The resulting motion is a bit unpredictable on purpose and feels like the real medium.

Overall, the prototype captures the behavior I care about for painting: quick strokes, visible wet-on-wet flow, pooling into blobs, and occasional drip events when the threshold is crossed.

6 Evaluation Plan

This system is not only a technical simulation but also an interactive tool meant to feel natural to artists. To see if it actually provides a believable watercolor experience, I plan to compare how artists work with real watercolor versus how they work with my digital tool.

6.1 Approach

The main idea is simple: I will invite a few watercolor artists and ask them to paint a small piece using real watercolor first. After that, I will ask them to recreate the same piece using my tool under similar conditions. The goal is not to match the result perfectly, but to see how the digital tool feels in comparison and how it supports their normal painting habits.

6.2 What to Observe

During each session, I want to watch for:

- how easily artists adapt to the tool,
- whether they recognize watercolor behaviors such as blending, pooling, or edge darkening,
- how much control they feel they have over water and pigment,
- and whether they describe the experience as natural or mechanical.

I will also record how long each painting takes and note any repeated attempts or hesitations, since those can show where the interaction breaks down.

6.3 After Each Session

After painting, I will ask a few short questions. For example:

- "Did the paint flow and mix the way you expected?"
- "Did the brush feel responsive?"
- "Which parts felt most like real watercolor?"
- "Which parts felt off?"

Responses will be rated on simple 1–7 scales (e.g., realism, control, responsiveness) and followed by open comments so participants can explain their thoughts in their own words.

6.4 What I Hope to Learn

From this comparison, I expect to learn which features make the digital medium feel alive and which parts still feel digital. The main thing I want to test is whether the unpredictable yet natural flow of real watercolor like water pooling into blobs and breaking when gravity wins also gives artists that same sense of spontaneity in my system. If the artists say "this feels like watercolor," that would be the most convincing result.

7 Limitations and Future Work

This is an active prototype. It works well enough to paint and to study the behavior, but there are a few gaps I want to address.

7.1 Water Blob Shape and Drying Look

Right now, water blobs do not hold a clean curved profile as they settle and dry. The edge forms, but the curvature is not as convincing as real puddles on paper. This reduces the visual effect during drying. I likely need a more advanced fluid model (or at least a better pressure/height coupling) to capture that shape and the way it collapses over time.

7.2 Color Feel and an Immersive Palette

In real watercolor, pigments from different sources feel different because of particle size and other physical properties. As an artist, I learned to "find" a color by mixing what I have, and that personal mixing process is a big part of an artist's style. A flat digital color picker misses that experience. My plan is to build an *immersive palette*: users can drop any pigments they want onto a palette area and mix them directly, then paint from that mixture. The goal is to bring back the tactile, exploratory feeling of building a color rather than picking a hex value.

7.3 Residual Oscillations

Even with the canceling/acceleration terms, there is a small oscillation in some flows. It's not disruptive, but it's visible. This likely needs better fluid equations or a different momentum treatment to settle more cleanly without killing the lively motion.

7.4 Brush–Canvas Exchange (Bidirectional Flow)

My current model assumes the brush gives water/pigment and the canvas absorbs it (and stops when saturated). In real watercolor, the exchange is bidirectional: a relatively dry brush can lift water or pigment back from the paper. Artists use this intentionally for corrections and for dry-brush textures. Adding a proper brush–canvas exchange (with brush moisture and local canvas wetness) would unlock dry brush strokes and more nuanced lifting. This should give artists a wider range of expression without hacks.

7.5 Broader Fluid Improvements

Beyond the points above, I want to explore: (1) a more explicit height/pressure model for puddle profiles, (2) finer control of capillary effects at paper edges, and (3) pigment-specific behavior (e.g., granulation) to push the look closer to real paint while keeping the system interactive.

References

[1] 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*. 421–430.

[2] Small David. 1991. Modeling watercolor by simulating diffusion pigment and paper fibers. *Proceedings of SPIE, Bellingham, Washington* 1460 (1991), 140–146.

[3] Godot Engine contributors. 2025. *Godot Engine.* https://godotengine.org Release 4.4.1 (26 Mar 2025).

[4] Tom Van Laerhoven and Frank Van Reeth. 2005. Real-time simulation of watery paint. *Computer Animation and Virtual Worlds* 16, 3-4 (2005), 429–439.

[5] Songhua Xu, Haisheng Tan, Xiantao Jiao, Francis CM Lau, and Yunhe Pan. 2007. A generic pigment model for digital painting. In *Computer Graphics Forum*, Vol. 26. Wiley Online Library, 609–618.