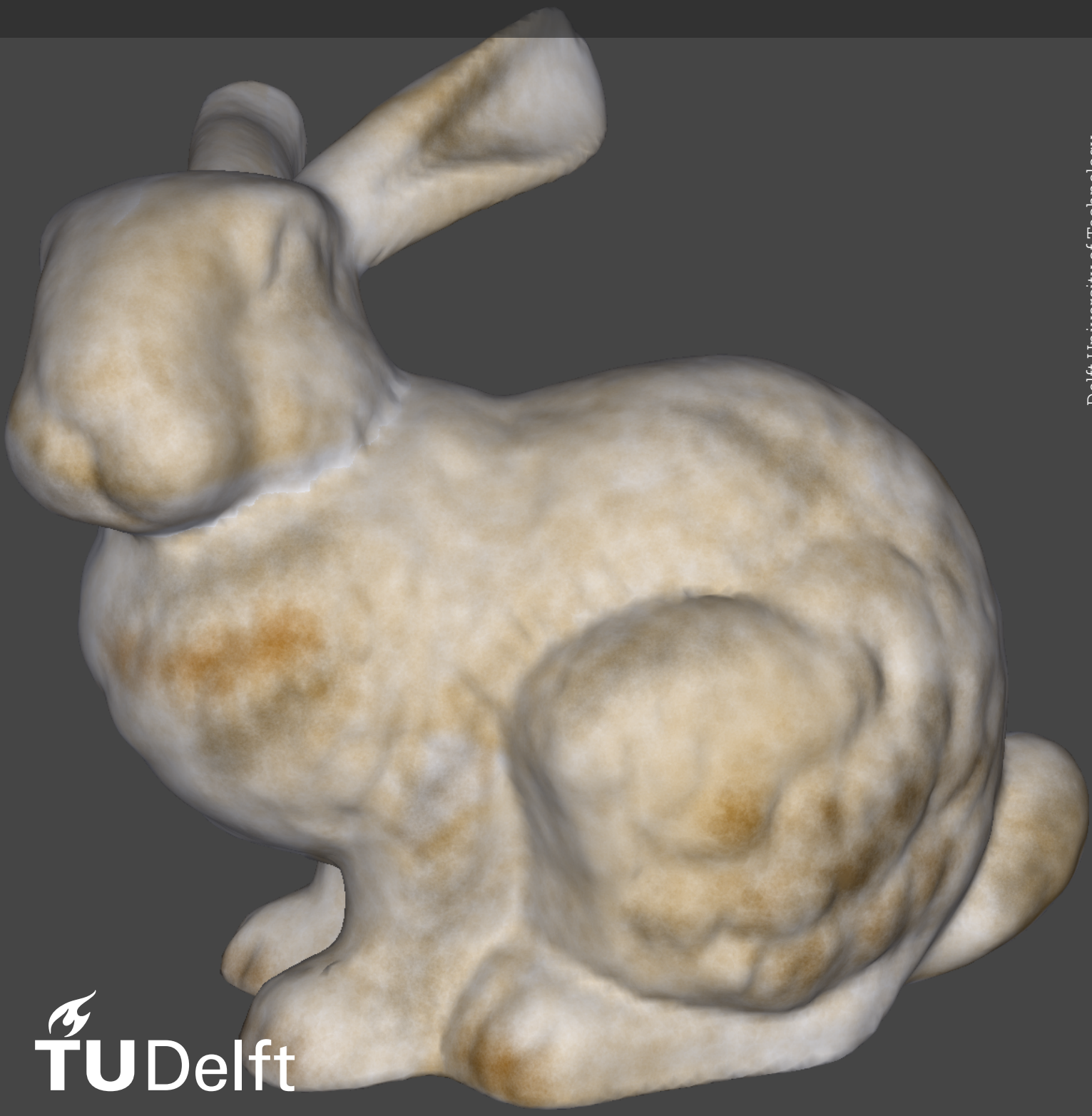
Dynamic Watercolour Painting on 3D Surfaces

Master Thesis

Medard Szilvasy



Delft University of Technology

TU Delft

Dynamic Watercolour Painting on 3D Surfaces

by

Medard Szilvasy

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday 14 October at 14:00.

Student number:	5267455	
Project duration:	18 December 2024 – 14 October 2025	
Thesis committee:	Dr. R. Marroquim	TU Delft, supervisor
	Prof. dr. P. C. Garcia	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

We propose a new method for simulating and rendering watery paint, which provides artists with a tool to produce watercolour-like textures on 3D models. Our particle-based system replicates most effects and techniques associated with watercolour, interactively and intuitively, as it is based on a physical fluid simulation. We show how our system can be applied to 3D surfaces by accurately compensating for distortions in the surface parametrisation.

Contents

1

Introduction

Digital tools allow artists to apply the same skills from traditional art forms to entirely new kinds of media, while also imitating the familiar experiences of drawing and painting on canvas or paper. 3D graphics are an obvious example, bringing new forms of artistic expression in modelling, texturing, and animation.

Texturing, for example, bears many similarities to real-world painting, but it also requires artists to familiarise themselves with material parameters and properties. A common approach is to map a 2D image onto the surface of a 3D object using a texture map. Software designed for texture editing allows the user to paint "what you see is what you get" textures directly on a preview of the 3D model [6], and not have to concern themselves about the structure of the texture map.

One could also paint a texture in 2D painting software and apply that to the mesh, but this approach has a shortcoming: the scale and orientation of the faces of the mesh can vary, and projecting a 2D image with details or patterns produced with a flat surface in mind, generally leads to those patterns appearing distorted in the 3D render. Any tool that replicates a complex brush or generates procedural details for the purpose of texturing, must therefore also warp those details to match the surface the texture is meant for.

Performing simulation in texture space adds another layer of complexity, as the elements of the simulation should move and interact with one another in a way that conforms to the geometry of the surface, and not its (effectively arbitrary) parametrisation. 3D simulation can work for 3D effects – acting in model space rather than parameter space. But if we specifically want to emulate a medium which is functionally 2D, such as watercolour paint, then we should have a 2D simulation which correctly warps scales and distances.

In this work, we introduce a watercolour simulation well-suited for 3D texturing. Watercolour is an art form normally associated with painting on a flat piece of paper. Hence, it should preserve all of its 2D characteristics during the simulation. Many existing approaches to watercolour are cell-based using a grid [3, 15], and can be expected to face caveats, causing watercolour effects and features which look normal on a flat plane to appear distorted on a surface, as the projection warps the grid. Our approach is instead particle-based, and performs rendering and simulation in a way that not only conforms to the surface being painted but also allows rendering at arbitrary levels of magnification. Figure 1.1 shows an example painting in our particle-based system.

DiVerdi *et al*. also proposed a "particle-based" watercolour simulation method [4]. We compare our method with theirs and how they differ in representing watercolour effects.

1.1. Concepts of Watercolour

We now give a brief introduction to some of the most important characteristics and effects of real-world watercolours that we seek to reproduce. We describe further characteristics and how we achieve them in Section 6.3.

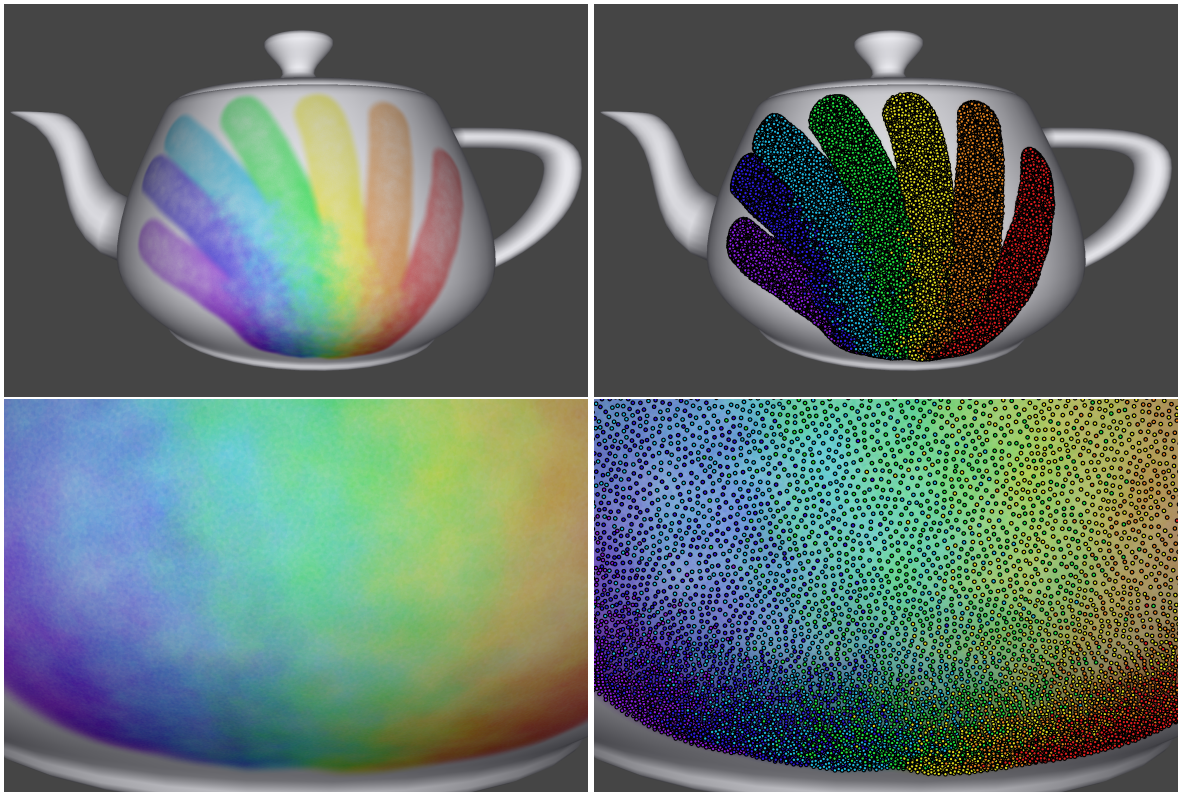


Figure 1.1: Top: watercolour painting on a 3D model; Bottom: a zoomed-in view on the lower part of the scene. We show the actual render on the left, while the right column shows the individual particles of paint drawn as circles. Notice the mixing of pigments where the different strokes intersect.

Edge darkening

Darkened edges are a typical characteristic of wet-on-dry watercolour painting, where pigments accumulate around the edges of a brush stroke as it dries.

Rewetting

The addition of water to a pigment which has not yet fully been absorbed by the paper can cause it to be advected by the water away from its initial location. An artist may add water around the edges of a brush stroke to spread the pigment out, giving it a feathered appearance.

Backruns

A backrun, or a *bloom*, is caused when a new drop of water spreads on a drying region of paint, desorbing some of the pigment from the paper and carrying it as it spreads [3]. This is often unintentional, and it is hard to quantify the exact effect that we expect to see, but in real-world watercolours, one can generally observe complex, branching patterns where the pigment gets deposited.

Glazing

Colour glazing involves adding a thin, pale layer of paint after the previous layers have already dried. This is used to tint areas of a painting without causing the paint to mix together.

1.2. Contributions

Our method treats the water and pigment application separately, as their artistic uses differ. Wetted parts of the model are handled by polygonal hulls, while pigments are treated as particles representing packets of pigment and water. Handling both parts concurrently makes up the main contributions of this thesis.

When water is applied to a 3D surface, we construct a polygonal hull in the texture space and store it in a way similar to silhouette maps [13] and vector texture maps [12]. A main characteristic of our method

is that it allows an arbitrary amount of overlapping strokes and discontinuities.

In the watercolour simulation, the hull is analogous to the part of the surface made wet by adding water, and is also used as the boundary for sampling pigment particles.

We also present a particle-based model for the appearance and motion of pigments suspended in water. Our aim is not about accurately modelling all of the physical properties of watercolour paint. It is rather about capturing the workflow, dynamic properties, and pleasing aesthetics associated with the medium. We model pigments as fluid particles in a Smoothed Particle Hydrodynamics (SPH) system [7, 10] and demonstrate emergent effects including edge darkening, feathering, and backruns.

Further, by using a Lagrangian (particle-based) pigment model, all associated transformations can be warped to revert any distortions caused by the surface parametrisation. The resulting texture both conforms to the surface's geometry and can be magnified in great detail. Such properties are much harder to achieve with an Eulerian (cell-based or similar) fluid model.

The remainder of this report is structured as follows. Chapter 2 discusses previous work on watercolour models and texture mapping. We discuss the aforementioned discontinuity mapping algorithm and pigment model in Chapters 3 and 4, respectively. Chapter 5 gives implementation details, and Chapter 6 experimental results and comparisons. Finally, we conclude our contributions and findings in Chapter 7.

2

Related Work

2.1. Digital Watercolour

There have been numerous efforts to faithfully reproduce the texture and dynamics of real watercolours in digital painting. We consider two broad approaches to watercolour simulation: the cell-based approach and the particle-based approach. Our method falls into the second category but, in this chapter, we discuss some representative examples of either approach.

2.1.1. Cell-Based

Cell-based watercolour simulation uses a cellular automaton to model the flow of water and subsequent pigment advection. Early work by Curtis *et al*. [3] (Figure 2.1) demonstrated impressive, realistic watercolour effects using a simulation of shallow water and pigment absorption in paper represented by a grid, although (on contemporary hardware) it was not truly a real-time simulation. Subsequent work by Van Laerhoven *et al*. [15] achieved interactive frame-rates by solving for time implicitly rather than explicitly, and also demonstrated oriental black ink and gouache paint styles by varying the parameters of the physical simulation.

It is unclear how a 2D cell-based paint simulation could be adapted to textures in a way that the flow of fluid conforms to the surface instead of the underlying grid.

2.1.2. Particle-Based

DiVerdi *et al*. created a procedural watercolour engine based on small, polygonal splats of pigment and an underlying "wet map" which constrains their motion [4]. Splat particles may take complex shapes as each vertex is subjected to an advection algorithm independently. On a zoomed-out scale, the underlying vector representation becomes unnoticeable, but the variedness of the particles produces rich watercolour-like textures. Watercolour effects are achieved by adding water to the wet map and using pre-designed brushes that affect different parameters of the advection algorithm.

Our method is also particle-based, but instead of polygonal pigment particles with moving vertices,

Figure 2.1: Computational watercolour effects presented in the seminal work of Curtis *et al*. [3]

our particles represent packets of pigment and water within a Lagrangian fluid model. We opt for this representation because by using a physically based fluid simulation, we can observe watercolour effects as an intuitive product of flow processes that exist in the real world. We give a more detailed comparison of our method to DiVerdi *et al*. in Section 6.3.

2.2. Discontinuity Mapping

Texture maps are widely supported in graphics applications and effectively model local changes in material properties. They are also raster-based and have to be filtered to render at an acceptable quality when magnified. To prevent a typical bilinear filter from blurring intentional discontinuities in the texture, Sen [13] stores boundary information between adjacent texels, producing a hard edge at all resolutions where a boundary is present and applying a bilinear filter elsewhere. Similar work includes contributions by Ray *et al*. [12] to map discontinuities represented by cubic curves onto a texture map, drawing complex vector graphics on 3D surfaces.

2.3. Procedural Textures

Key contributions in procedural texturing include Perlin's inaugural work on solid textures [11], where a 3D texture is evaluated on the surface points of an object, and procedural surface noise [8], which projects directly onto the surface. Both methods produce undistorted infinite-resolution textures, and they do not even require a parametrisation. Our work also produces procedural, noise-like textures on a surface, but we also provide a simulation to give the artist dynamic control.

3

Discontinuity Mapping

The first stage of our method is to demarcate the outline of the user's brushstrokes. The next stage (Chapter 4) relies on this mapping to sample particles and set their boundaries. The resulting hull can also be rendered directly by performing a texture look-up within the map. Unlike traditional textures, this map is resolution-independent and maintains the hard edges of the hulls at all levels of zoom.

This follows a previous line of work on discontinuity mapping, the main difference being that our method allows arbitrarily many hulls and discontinuities to exist in the same area.

3.1. Painting

The user draws a stroke which is first rasterised, and then has its outline extracted using marching squares [9]. This is followed by a smoothing step where we resample the vertices of the outline by traversal: starting at an arbitrary location, and placing new vertices at uniform arc length increments.

The brush appears circular on the 3D surface but needs to be warped to maintain its form when processed in UV space, as shown in Figures 3.1. The process which we use is illustrated in Figure 3.2: the outer ellipsoid on the right is a circle in UV space, but it has to be reduced to a smaller ellipse in order to appear circular on the surface (left). Essentially, we want the geodesic length of the orange line to be the desired radius of the circle.

To do this, we trace the direction of the geodesic in UV space, starting with a desired length, and scale the remaining length of the line according to the size of the triangle face it is on. If the scaled line crosses a boundary into a different triangle, then the part of the line which is inside of the other triangle should

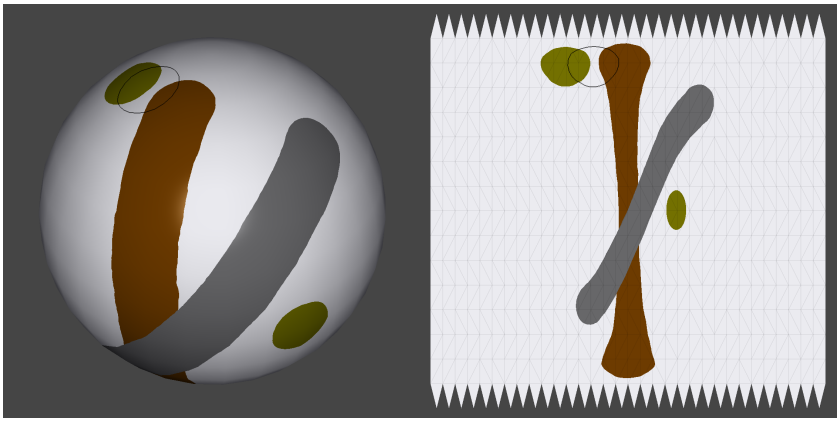


Figure 3.1: Left: different brush strokes with the same width on a UV sphere. Right: the corresponding strokes in parameter space.

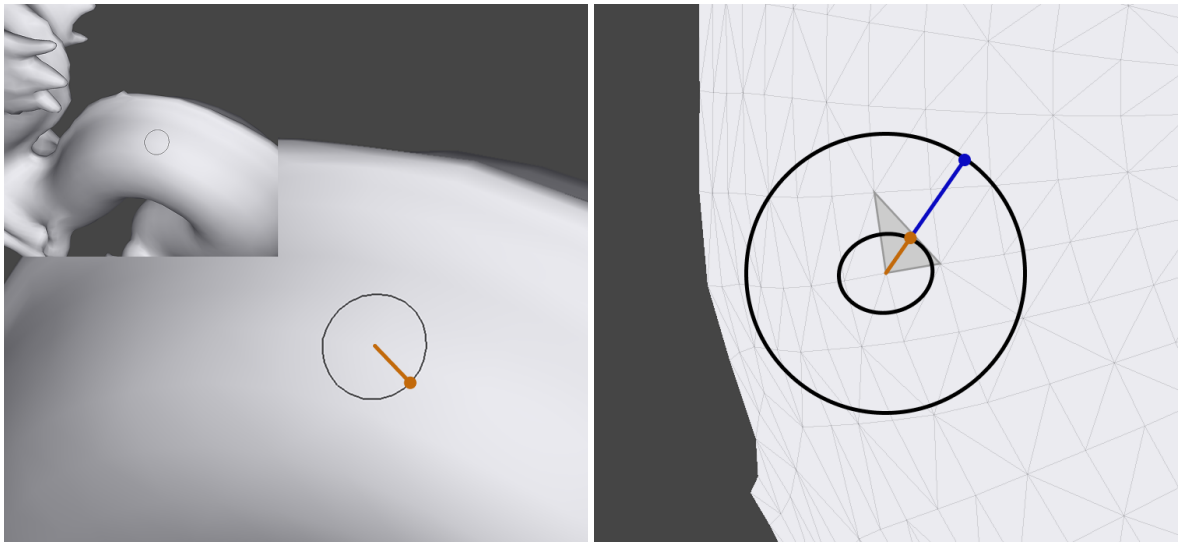


Figure 3.2: A circular brush must be warped in UV space (right) to remain circular on the 3D surface (left): to do this, we scale the radius of the initial circle (blue line) to the desired length (orange line) by traversing the triangle mesh.

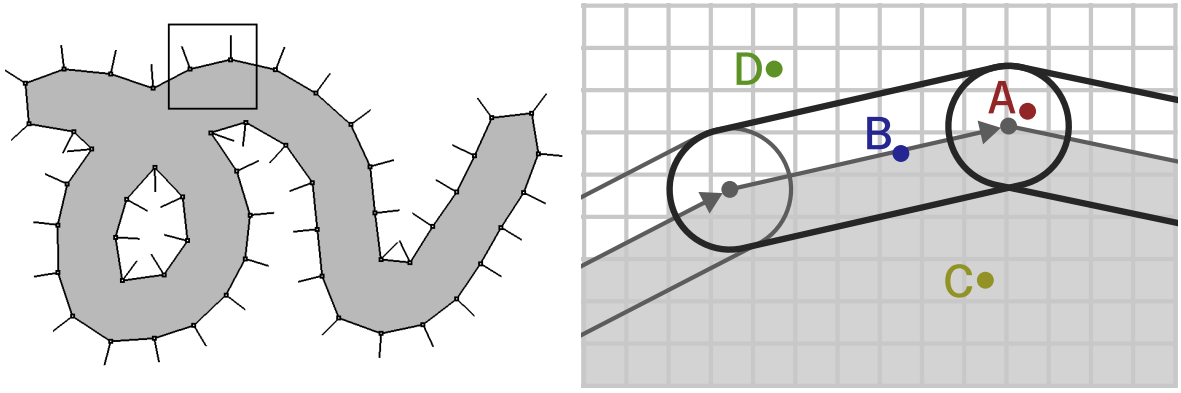


Figure 3.3: Left: a stroke that might be rendered within our system, showing all vertices and their normals. Right: a close-up of a particular line segment. The cell centred at A is at the overlap between two segments, while B is only inside one. C is internal to the stroke but not near the boundary, and D is external without any incident line segments.

be scaled to that triangle face instead. This process is repeated iteratively until we find the end point of the line – it is greatly accelerated by maintaining a data structure with adjacency information between triangles.

Generally, this algorithm only has to traverse a small number of triangles before it reaches the end of a geodesic: in Figure 3.2, only the triangle we filled-in needs to be traversed.

3.2. Discontinuity Map Construction

The discontinuity map is a grid in which each cell stores information about what hulls are close to it. For each hull, it stores any of the edges that a point in the cell should be compared against to see if it is inside the hull or not. We also want to compute the shortest distance from a point to its closest hull outline – which requires us to store edges in more than just the cells that they intersect, but also in any other cells that are close enough to be taken into consideration.

When constructing the map, we draw edges as buffered line segments (see Figure 3.3, right) to include all cells in their neighbourhoods when rasterised.

We give more detail about the system of linked-lists that we used to implement arbitrarily large buckets per cell in Section 5.2.

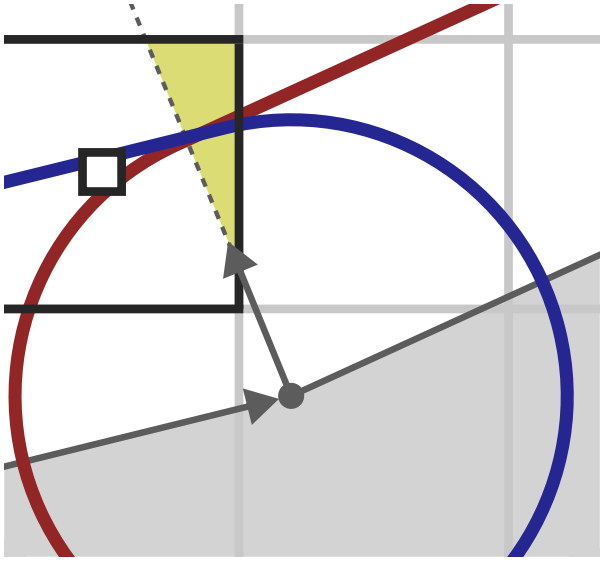


Figure 3.4: An example of an ambiguity that can happen at the intersection of two line segments, where the marked cell (top-left) is covered by the buffer of one line segment (blue) but not the other (red). The area highlighted in yellow should not be checked against the left segment, because it is on the other side of the dividing line, but it cannot be checked against the right segment either.

3.3. Handling Stroke Boundaries

For the simple case when there is only one line segment incident on a cell, as is the case for the cell at B in Figure 3.3, it is easy to determine if a point is inside the respective stroke. We compute the cross product of the point and the line segment and determine whether it is inside based on the sign of the result, and the distance to the boundary based on its magnitude.

We have to be careful when treating multiple incident line segments from the same stroke, however. We always consider the segment that has the shortest distance to the point being evaluated, and update it whenever another segment from the same stroke with a shorter distance is found. To prevent adjacent line segments with a shared vertex (such as cell A in Figure 3.3) from intruding on one another, we use the vertex normal as a dividing line, so that only the line segment on one side of the normal is evaluated for a given point.

There is an important edge case that needs consideration, pictured in Figure 3.4. Here, the centre of the cell in the top-left is covered by the buffer of the line segment on the left (blue outline), but not the buffer of the right line segment (red). Using the aforementioned rasterisation process, this cell would not be "aware" of the stroke on the right, but part of the cell is to the right of the normal dividing line between the two segments. When we want to determine which side of the boundary a point is on – in an area such as this one (marked in yellow) – we check it against the tangent to the vertex: the point is exterior if it is above the tangent, and interior if it is below it.

The process we just described can be used to find every hull that a given UV coordinate falls inside of. In the next chapter, we will describe the pigment model, which makes use of the discontinuity map for its sampling and boundary conditions. The hulls can also be drawn on their own (as in Figure 3.1), in which case the discontinuity map is accessed by the fragment shader similarly to performing a texture lookup.

4

Pigment Model

Our goal is to procedurally reproduce watercolour characteristics with an approximate simulation of pigment advection in water. The wetted parts of a model in our simulation are represented by vectorised hulls, as described in the previous chapter. For the pigments, we adapt the SPH-based approach to fluid simulation described by Müller *et al*. [10].

Our approach is to model parcels of pigment as particles of fluid, with boundary conditions set by the hull, so that pigments gradually diffuse down the concentration gradient where water is present. By using an appropriate sampling method and carefully tuning the parameters of the simulation, the paint produces varied, cloudy textures from variations in pigment density. Watercolour effects emerge naturally from the simulation, reproducing characteristics such as darkened edges and feathering (Figure 4.1).

These particles act as a metaphor for water as much as they do for the pigments suspended in them. For instance, the backrun in Figures 4.3 and 4.4 was induced by adding transparent particles in the area, as though more water was being added to the canvas, but without pigment.

We will first describe our method of sampling particles (Section 4.1), then introduce the fluid model (Section 4.2), and finally describe how we composite and render the results in real-time (Section 4.3).

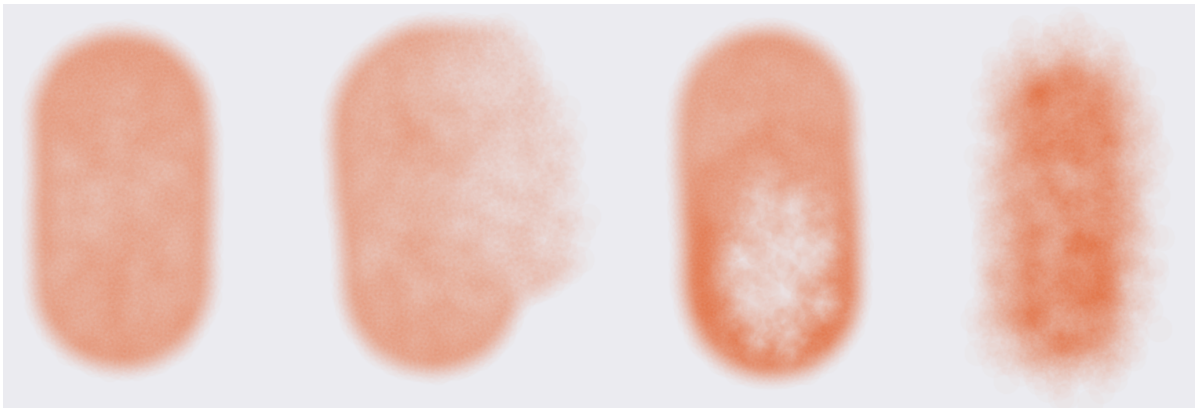


Figure 4.1: Some strokes made with our algorithm demonstrating different paint effects. From left to right: the first stroke was added to a dry surface and left untouched as it dried. The second has a feathered edge where extra water was added. The third stroke features an intentional bloom effect. Finally, the rightmost stroke was drawn on a wet surface to demonstrate wet-on-wet painting.

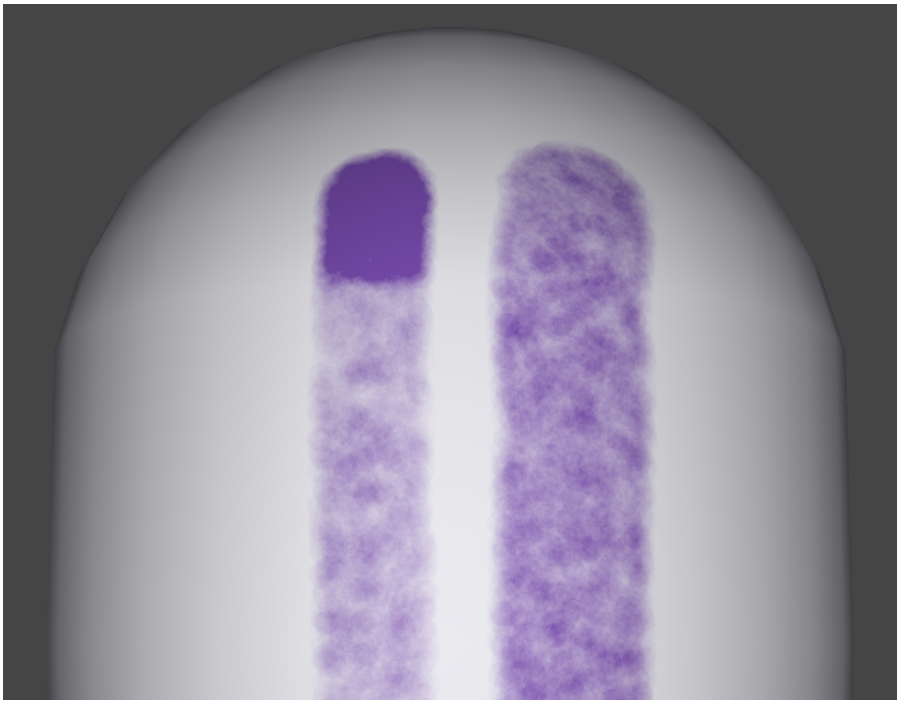


Figure 4.2: Appropriate and inappropriate samples of points on a sphere which is heavily distorted at its equator. The line on the left is oversampled above the equator and undersampled around it, while the line on the right is sampled correctly according to the surface geometry.

4.1. Sampling

When the user draws a stroke, and after the hull is vectorised and mapped, we sample a uniform random distribution of points inside its boundary. We invoke a shader program on each cell of the discontinuity map to generate new points using rejection sampling.

Here, we have to take care if we want a sample that conforms to the surface, and not to the parametrisation. Figure 4.2 shows how, if we naïvely try to sample the same density of points across the grid, then faces with a higher relative size in parameter space will have a higher density of points on the surface itself than those of a smaller scale.

We analyse each triangle in a cell separately, establishing a bounding box around the part of it that is inside the bounds of the cell. The surface area of the bounding box is A_{uv} in parameter space and A_{surf} in 3D. Let C_{uv} be the area of the cell in parameter space. Then the mean number of points inside a bounding box is

$$\lambda = \lambda_{\text{uv}} \frac{A_{\text{surf}}}{A_{\text{uv}}} \cdot \frac{A_{\text{uv}}}{C_{\text{uv}}} \tag{4.1}$$

for a common λ_{uv}, but since all grid cells have the same size, we use

$$\lambda = \lambda_{\text{ref}} A_{\text{surf}} \tag{4.2}$$

where the user chooses λ_{ref} before drawing the stroke. We draw k samples from the Poisson PMF with mean λ to determine the number of points in each box.

$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{4.3}$$

These points may still be rejected if they fall outside of the triangle. This finally gives us a distribution that is uniform over the surface.

4.2. Simulation

We apply a simulation in four stages similar to Harada *et al*. [7]. The first stage is to generate buckets of nearby particles, which we elaborate on more in Section 5.2. The properties of the particles are updated in the next three stages: we first compute the density of particles around a pigment, then we apply the forces from nearby particles and update its momentum, and finally we move it to its new position.

We use the same kernels and formulae to compute density and forces as Müller *et al*. [10], except that we do not include a mass. A pigment particle instead has a wetness parameter w_i, which decreases with time and forces it to slow as it dries. The velocity of a particle is

$$\mathbf{v}_i = w_i \mathbf{p}_i \tag{4.4}$$

where $\mathbf{p}_i$ is its momentum.

4.2.1. Density

The density of pigment at location $\mathbf{r}$ is given by

$$\rho(\mathbf{r}) = \sum_j w_j W(|\mathbf{r} - \mathbf{r}_j|, h) \tag{4.5}$$

where we apply the following weight kernel on all nearby particles.

$$W_{\text{poly6}}(r, h) = \frac{315}{64\pi h^9}(h^2 - r^2)^3 \tag{4.6}$$

Where r is the distance of the point from the centre and h is the radius of the kernel. Note that we have to scale these distances to match the surface distortion, using the same method mentioned in Section 3.1.

4.2.2. Forces

Nearby particles may repel one another, forcing a fluid to spread to where its density is lower. This is modelled by a local pressure

$$P_i = k\rho_i \tag{4.7}$$

where k is constant. Then the force applied on a particle by all others with which it interacts is:

$$\mathbf{F}_i^{\text{press}} = -\sum_j w_j \frac{P_i + P_j}{2\rho_j} \nabla W_{\text{spiky}}(\mathbf{r}_i - \mathbf{r}_j, h) \tag{4.8}$$

using

$$\nabla W_{\text{spiky}}(\mathbf{r}, h) = \frac{45}{\pi h^6}(h - |\mathbf{r}|)^3 \frac{\mathbf{r}}{|\mathbf{r}|} \tag{4.9}$$

By having a carefully tuned viscosity force, we preserve random details such as clusters of particles that formed during sampling.

$$\mathbf{F}_i^{\text{visc}} = \sum_j w_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W_{\text{visc}}(\mathbf{r}_i - \mathbf{r}_j, h) \tag{4.10}$$

$$\nabla^2 W_{\text{visc}}(\mathbf{r}, h) = \frac{45}{\pi h^6}(h - |\mathbf{r}|) \tag{4.11}$$

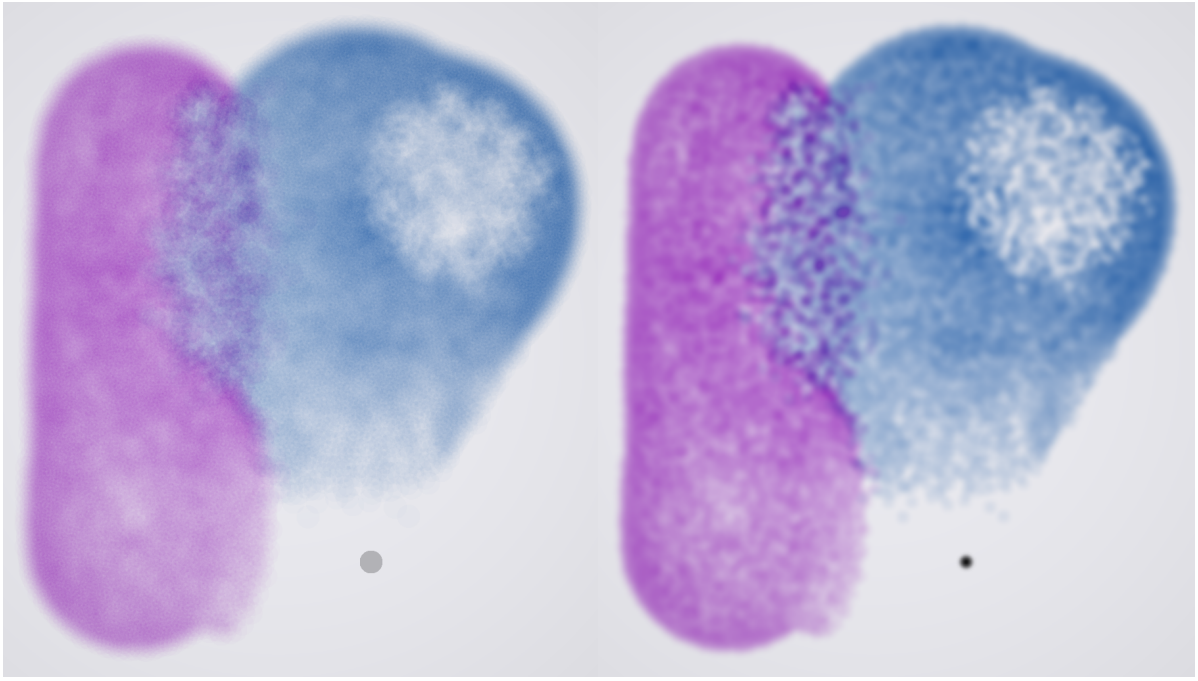


Figure 4.3: Dried paint rendered using a disc-shaped kernel with sharp edges (left) and the smooth W_{poly6} density kernel (right). Note the single black particle drawn with the kernel in the bottom-right corner of either image. All particles have the same size, but the disc appears to produce details at various scales.

This helps prevent such features from diffusing completely.

We additionally apply a friction force if (and only if) a pigment falls outside of all hull boundaries; it is on a dry surface, so it rapidly comes to a stop.

$$\mathbf{F}_i^{\text{frict}} = -\mu\rho\mathbf{p}_i \tag{4.12}$$

4.2.3. Motion

Once all forces have been applied, the change in momentum is

$$\frac{d\mathbf{p}_i}{dt} = \frac{\mathbf{F}_i}{\rho_i} \tag{4.13}$$

And we apply Equation 4.4 to find the velocity. Before we update the particle's position, we must also scale the displacement according to the parametric distortion using the same method we described previously.

4.3. Rendering

Arguably, the most faithful way to render pigments is to use the same W_{poly6} kernel (Equation 4.6) for the local intensity as is already the direct metaphor for local pigment density in the fluid simulation. However, there is an entirely different kernel we can use for rendering to enhance the texture of the paint at no extra cost.

In his work on Spot Noise [16], van Wijk noted how the convolution of a Poisson point process and a simple, disc-shaped noise kernel with sharp edges could produce noise with a cloudy texture and fractal detail. We also use a Poisson point process in our method to sample initial pigment locations. Although the simulation affects this distribution, it still remains mostly unstructured. Therefore, our model also benefits from using this kernel, as seen in Figure 4.3.

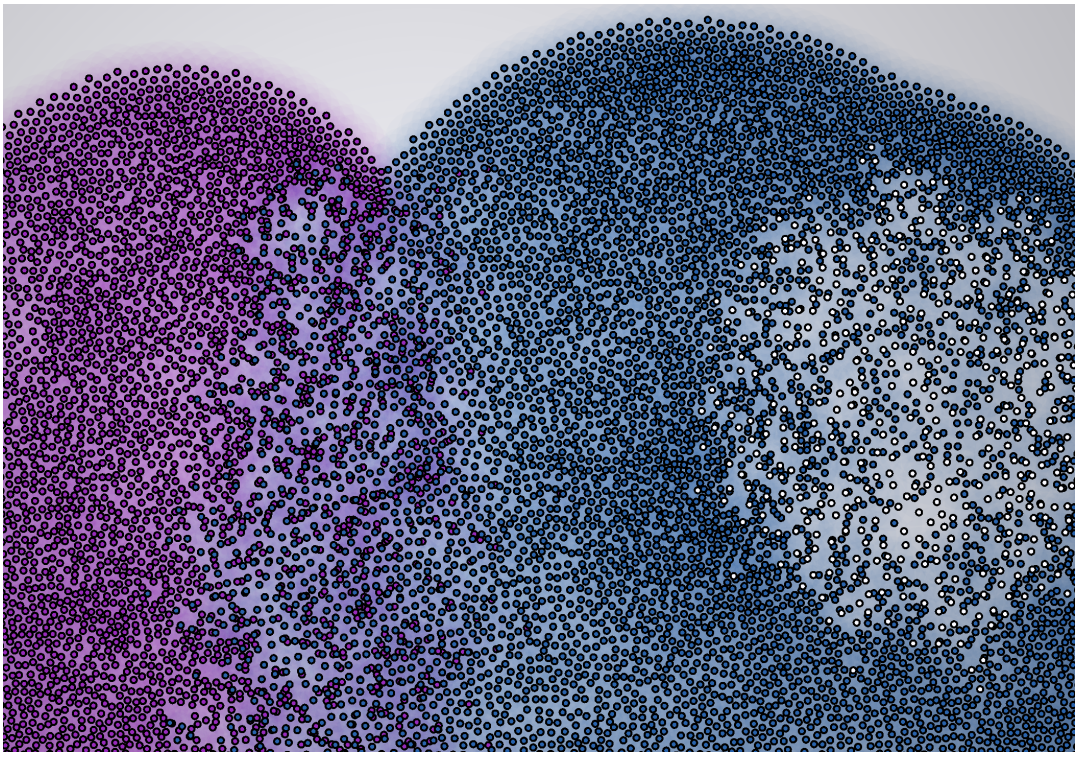


Figure 4.4: A close-up of the upper part of Figure 4.3 with pigment particles explicitly drawn as coloured circles. The white points on the right-hand side of the image were added after the blue stroke and made completely transparent, but they influence the flow of the paint nonetheless.

These properties are strongly desired. We expect to see fractal detail stemming from turbulent flow in the water, which occurs at all scales, but our fluid model only deals with particles that are relatively large compared to the scale at which they are rendered. Previous work [2], tracing back to Perlin's turbulence textures [11], achieves the same effect by compositing several successive octaves of noise. To attempt something similar, we would have to sample and simulate pigments at multiple scales, with an exponentially growing sample size as we add more octaves. Eventually, it would render real-time simulation prohibitively expensive. Our alternative rendering approach imitates what we would expect to see in a more fine-grained model, even though the resulting detail does not come from the simulation.

Regardless of the choice of kernel, particles are composited by accumulating the intensity of nearby kernels and averaging their colours weighted by intensity. Because this operation is order-independent (as are the other simulation steps), the buckets used to track nearby particles do not have to be sorted.

5 Implementation

We created an interactive application using C++ and OpenGL 4.6 for painting, rendering, and simulating watercolour on the GPU, using the methods described in this thesis. In this chapter, we present the details of the implementation, and in Chapter 6 we will give an overview of its performance.

Figure 5.1 shows our application in use. The user may set the size, density and color of pigments, as well as several properties of the particles in the fluid simulation: the coefficients of pressure and friction, the viscosity, and the drying rate. It is also possible to change the diffuse colour and specular properties of the paint hull.

The program provides some debug views, which were used to make many of the images in this report. These include toggles for explicitly drawing pigment particles (as seen in Figure 4.4), drawing hull outlines and vertices, or highlighting which pigments are in the kernel range of the cursor. The user can also switch between the default 3D view and a 2D view showing the unpacked texture as it exists in parameter space.

5.1. Mesh Setup

The program can load a triangular mesh with a pre-existing parametrisation given by per-vertex UV coordinates. The UV map is subject to certain limitations under which our technique is well-defined.

We make the (reasonable) assumption that the mapping is invertible with no overlaps, such that any point on the two-dimensional UV map can be traced back to a single location on the surface of the model. We require this for distortion compensation, as we cannot conform a transformation to the surface if multiple different faces of it share the same UV coordinates.

We also do not handle distortion compensation in degenerate triangles, where barycentric coordinates are undefined.

After the mesh is loaded, it is normalised, and we construct a grid of linked lists for quick triangle lookup in parameter space. We also store adjacency information for each triangle by saving the index of the neighbour at each of its edges. As we mentioned before, this helps speed up processes where we trace a line in parameter space and apply scaling from every triangle it crosses – a common operation in distortion compensation.

5.2. Linked Lists on the GPU

Several stages in our method demand quick look-ups of all nearby elements of a type around a location: to determine which surface triangle covers a UV position, to discern the outline of a paint hull, or to apply a kernel on all particles surrounding a spot. The number of elements is arbitrary.

Modern graphics APIs, including OpenGL 4.6, provide atomic memory operations that are useful for constructing a linked list, or a grid of them with a shared buffer, entirely on the GPU [17]. List nodes are placed in a buffer using an atomic counter to assign indices, and each node stores the index of the next

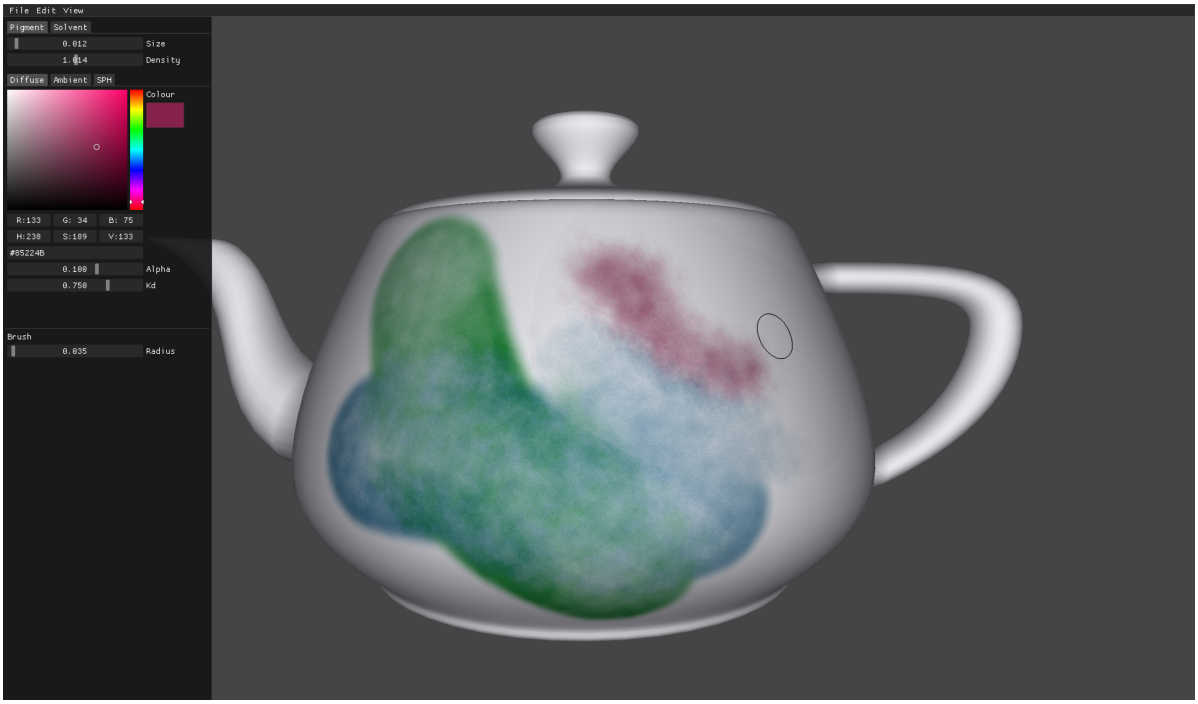


Figure 5.1: The painting application in use. The green paint was added first, followed by the teal paint with the same parameters. A large stroke with no pigments (only the wet hull) was then added to make the teal paint diffuse, and the purple stroke was added last – in the water – with a higher viscosity and faster drying to produce a wet-on-wet stroke that does not diffuse as much. Not all pigments had dried when this screenshot was taken.

one in the list. A grid only has to store the head of each list, and we can construct a map that lists all elements incident at a grid cell, with the only limit being the size of the shared node buffer.

5.3. Data Structures

We now give a brief summary of how we set up the data structures related to the paint model. The memory usage of our method can be quite substantial, so we will justify how this configuration helps reduce the memory load. We provide more detailed metrics of memory usage in Chapter 6.

Table 5.1 shows the types related to the model and how they are structured. The types with `Node` in their names are linked-list nodes (Section 5.2). Pointer attributes are shown with an arrow followed by the type that they point to. Here, a "pointer" is simply an index to the element being pointed to in the related buffer. Each type has its own buffer.

The most numerous instances in a typical session are the linked list nodes – a single hull, line segment, or pigment, usually has multiple nodes pointing to it from all grid cells in its area. For example, there may be hundreds of thousands of populated cells in the grid of `PigmentNode` lists, each of them tens of nodes in length to keep track of all pigments inside or close to the cell. To keep memory usage reasonable, the size of a linked list node is therefore the bare minimum needed to point to the element(s) it pertains to and to the next node in the list.

Individual `Pigment` and `HullVertex` instances are much less numerous in comparison to the list nodes that refer to them. They store the properties related to that element in the paint model. A `Pigment` also points to the `Triangle` it is inside, which is information that is reused between simulation stages.

The entry point for discontinuity mapping is the `HullNode`, which points to a `Stroke` instance storing the optical properties of that hull, as set by the user. The `Stroke` type is also used for properties that are common to all pigments added in the same stroke, including their colour and simulation parameters. A new `Stroke` instance is only created when the user draws a stroke, so the size of its buffer is truly negligible. A `HullNode` can also point to the head of a `LineNode` list, which stores nearby line segments on the edge of the hull (see Chapter 3).

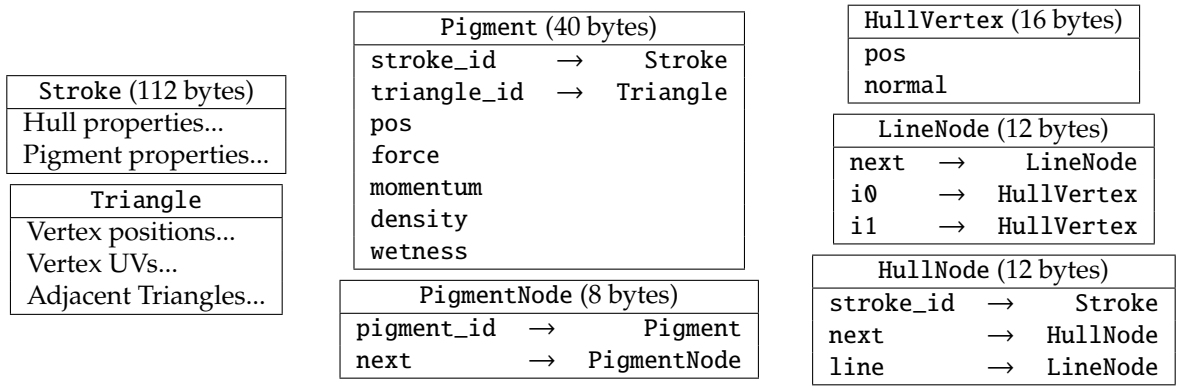


Table 5.1: Most of the structs used in our paint model. Pointers (which are just buffer indices) are shown with an arrow, followed by the type being referenced. We give the size of each struct under OpenGL's `std430` layout rules in bytes.

5.4. Limitations

There are certain limitations in our implementation that we believe are not too complex to overcome, but were outside the scope of this work.

Dried particles, whose wetness has reached 0, should no longer be included in the simulation – they do not have an effect on nearby pigments, and their positions do not need to be updated or redrawn. Once a pigment has dried, it could safely be drawn to a dried pigment grid, which is only involved in rendering but not simulation. We have not implemented this, but it would significantly bring down the runtime of the program while working on complex scenes.

Similarly, we do not delete water hulls at any point, which makes wetted parts of the model remain wet indefinitely. Ideally, the hulls would also dry over time using a timing set by the user.

We also do not allow brush strokes or particles to cross seams, where the UV mapping is not continuous. However, including adjacency information along seams could allow the user to draw over them without restriction.

6

Results

We conducted some experiments to evaluate the performance of our implementation. The main metrics we tested were the speed of the individual simulation and rendering stages, and the time-memory trade-off associated with the pigment model.

All tests were run on a laptop with Radeon 780M integrated graphics on an AMD Ryzen 7 8840U CPU. Rendering was on a viewport 2560 pixels by 1494 pixels in size.

6.1. Experiments

We created a series of simple scenes, each with approximately double the complexity of the previous, and timed the durations of those stages of the model that run once per frame. This includes the three stages of the fluid simulation described in Chapter 4, the creation of buckets (linked lists as described in Section 5.2) for looking up nearby pigments, and rendering the paint and the surface in 3D. Results are listed in Table 6.1 and plotted in Figure 6.3.

The scenes used, as seen in Figure 6.2, were set up by drawing each spot of paint with the same parameters. Pressure and viscosity coefficients were set to 4e−10, the friction coefficient to 1, and the drying rate to 0, to ensure that all of the pigments remained in motion as the test continued. The simulation was left to run for a minimum of 2 minutes each time a new set of spots was added, to allow the new pigments to have time to diffuse.

Scene	Pigment Count	Density	SPH Force	Motion	Buckets	Render	Total
1	2,028	0.73	0.75	0.02	2.04	4.33	7.87
2	4,117	1.35	1.37	0.03	3.7	5.24	11.69
3	8,208	1.58	1.48	0.04	3.74	5.03	11.87
4	16,382	2.41	2.24	0.06	5.73	5.7	16.14
5	32,406	4.32	4.62	0.14	10.76	8.26	28.1

Table 6.1: Duration (in milliseconds) of each simulation and rendering step on the scenes in Figure 6.2. All times are averaged over 100 frames.

We noticed that bucket generation is often the slowest task. As previously mentioned, buckets are part of a uniform grid of pointers to the head of each list – having a denser grid will bring a proportionally higher memory usage, but we also expect it to improve the speed of kernel lookups, shortening the runtime of the fluid simulation and rendering.

To test this theory, we rendered the scenes in Figure 6.1 with 5 different dimensions of the bucket grid; the previous experiment involved a 1024×1024 grid. The results (Table 6.2) suggest that 1024×1024 to 2048×2048 are good choices of grid size for these models, with modest-to-low memory demands for storing the pointer grid and all of the `PigmentNode`s.

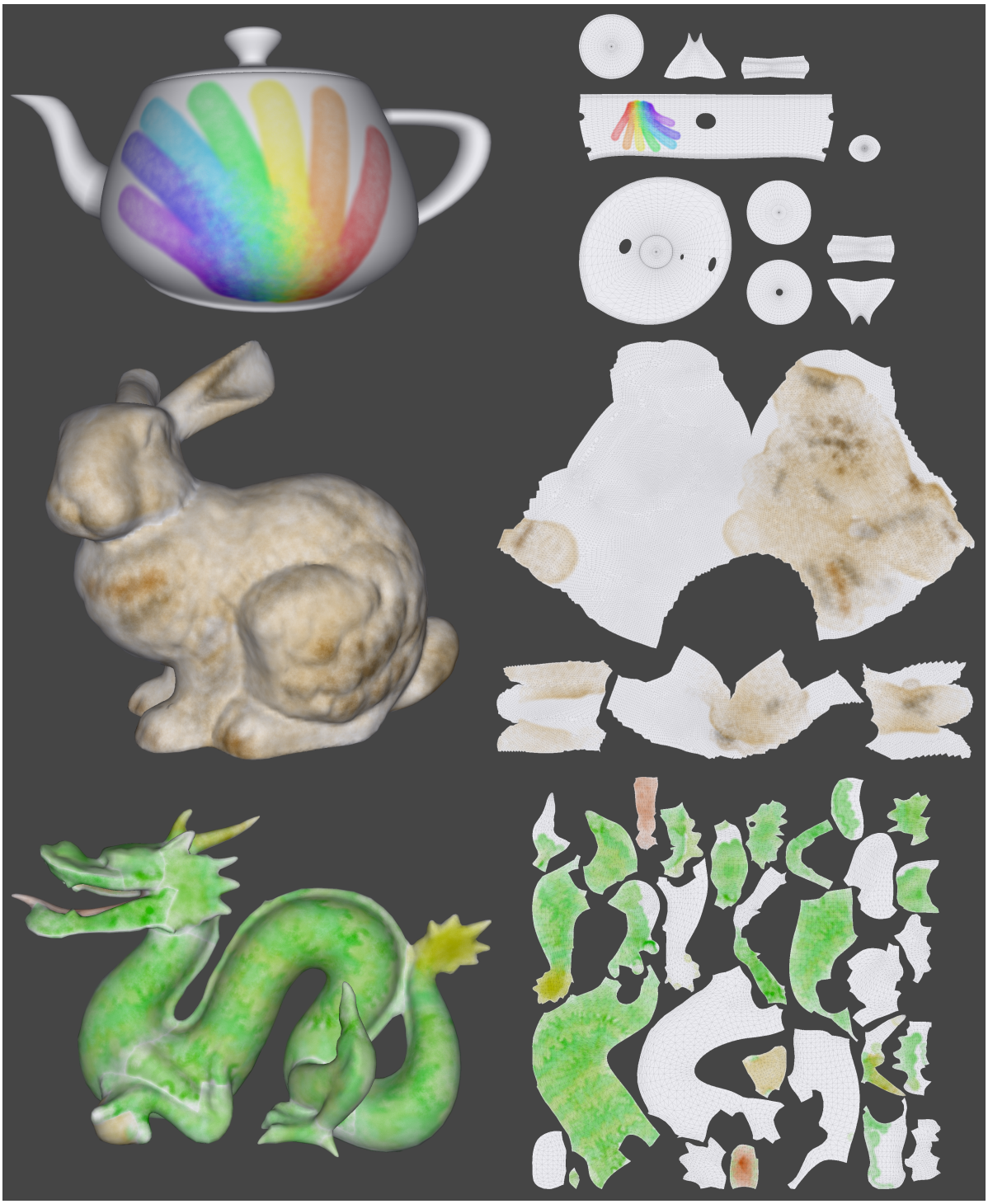


Figure 6.1: Some models which have been partially painted using our implementation, with unwrapped views of the underlying textures.

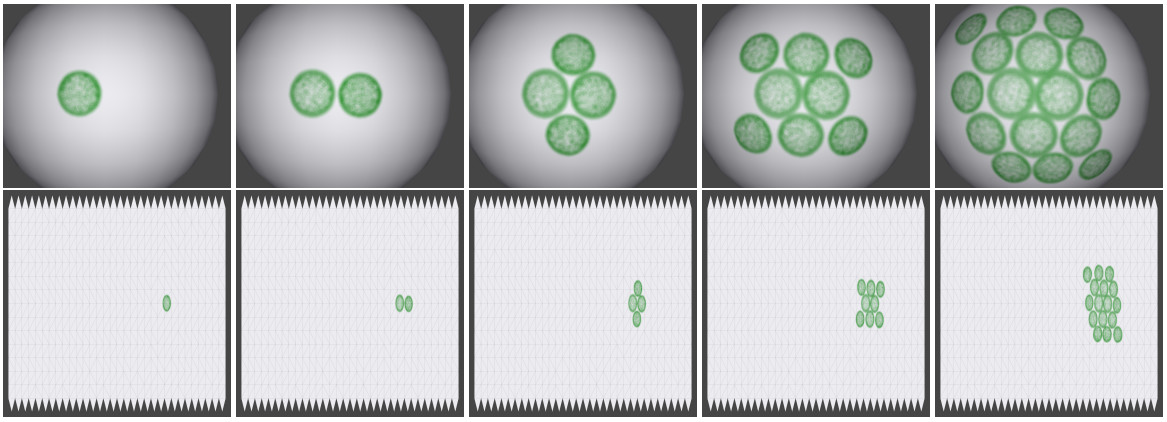


Figure 6.2: Scenes 1-5 (from left to right) used for evaluating the performance of our application. The images show the state of the scene when screenshots were taken of the program's metrics. The top row shows the view that was rendered, and the bottom row shows the texture map for each.

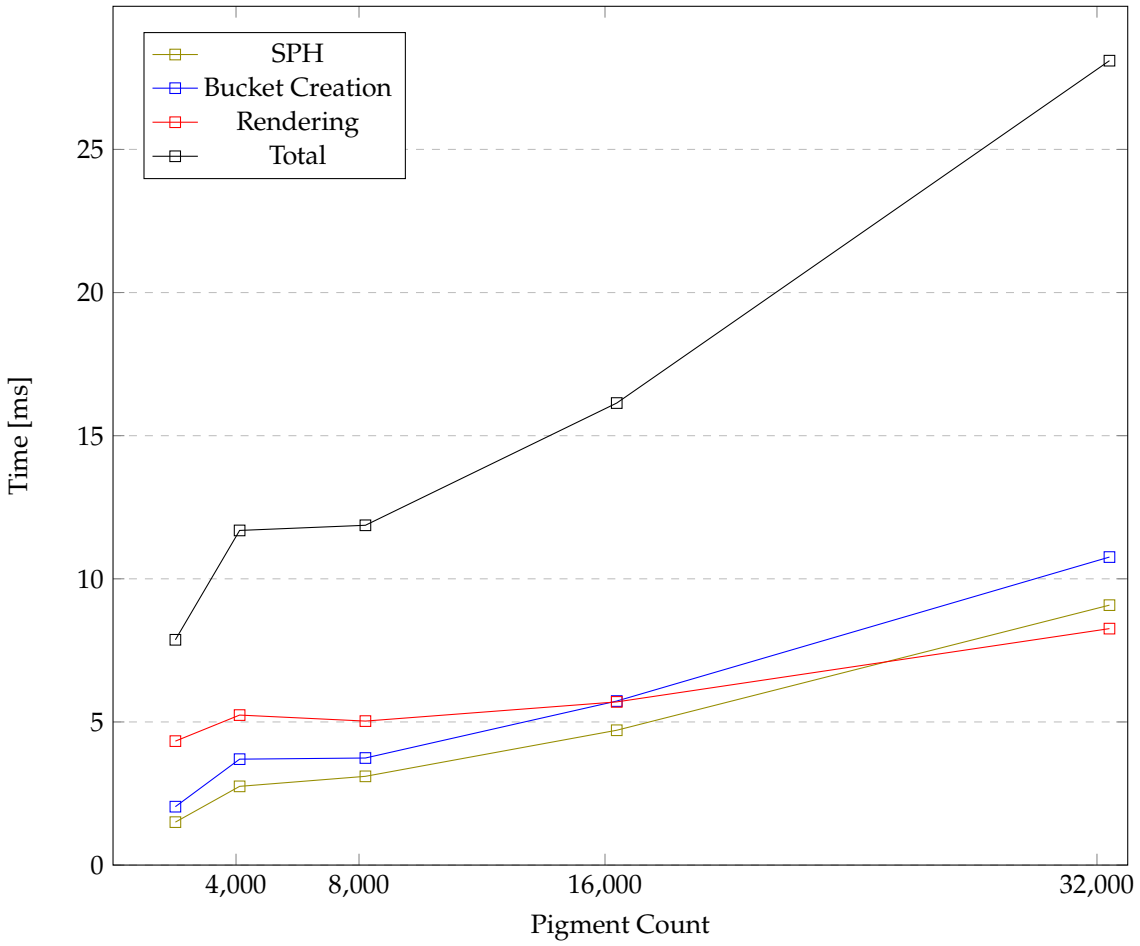


Figure 6.3: Runtime of each simulation/rendering step during the experiment shown in Figure 6.2. Apart from one outlier, we observe a linear relation between the number of pigments and simulation time, while rendering time is affected less by increasing complexity.

Teapot (15,704 triangles; 29,651 pigments)

Grid Dimensions	Node Count	Memory Cost		SPH			Buckets	Render	Total
		Nodes	Grid	Density	Force	Motion			
256x256	239,780	1.9MB	2.1MB	24.68	24.74	0.04	14.43	28.75	92.64
512x512	460,223	3.7MB	8.4MB	11.7	11.82	0.04	12.71	15.63	51.9
1024x1024	1,118,362	8.9MB	33.6MB	8.09	8.17	0.04	12.15	11.57	40.02
2048x2048	3,298,436	26.4MB	134MB	**7.37**	**7.41**	0.04	**12.11**	**11.44**	**38.37**
4096x4096	11,115,935	88.9MB	537MB	9.63	9.62	0.04	14.24	14.97	48.5

Bunny (65,630 triangles; 81,467 pigments)

Grid Dimensions	Node Count	Memory Cost		SPH			Buckets	Render	Total
		Nodes	Grid	Density	Force	Motion			
256x256	1,495,627	12.0MB	2.1MB	110.55	104.08	0.12	88.34	90.67	393.76
512x512	3,035,853	24.3MB	8.4MB	49.03	47.98	0.13	74.03	43.34	214.51
1024x1024	7,773,177	62.2MB	33.6MB	29.9	30.53	0.12	62.25	28.59	151.39
2048x2048	23,837,803	191MB	134MB	**23.94**	**24.62**	0.12	**59.93**	**25.8**	**134.41**
4096x4096	82,316,167	659MB	537MB	30.62	30.41	0.12	68.8	41.16	171.11

Dragon (19,332 triangles; 197,915 pigments)

Grid Dimensions	Node Count	Memory Cost		SPH			Buckets	Render	Total
		Nodes	Grid	Density	Force	Motion			
256x256	2,097,991	16.8MB	2.1MB	104.32	99.13	0.31	115.53	63.15	382.44
512x512	4,347,896	34.8MB	8.4MB	49.28	48.53	0.32	99.91	34.02	232.06
1024x1024	11,237,701	89.9MB	33.6MB	34.53	35.12	0.32	**91.33**	**25.54**	**186.84**
2048x2048	34,620,697	277MB	134MB	**31.74**	**33.56**	0.33	96.29	26.81	188.73
4096x4096	119,824,351	959MB	537MB	33.48	36.21	0.34	104.49	39.33	213.85

Table 6.2: Comparison of memory usage and runtime while performing the full simulation on the models in Figure 6.1. Times are given in milliseconds. We tested different sizes for the pigment bucket grid; these are shown in the first column.

6.2. Interpretation

The simulation times in Table 6.2 may seem concerningly high, but as mentioned in Section 5.4, this is a result of all particles being involved in the simulation; even those which have dried. This shows the need to remove dried particles from the simulation, as these are not interactive frame rates. Regardless, the rendering time is low and lends itself to real-time rendering of complex textures once the simulation is stopped.

The experiment shown in Figures 6.2 and 6.3 is indicative of simulation times in a scenario where only a (comparatively) small number of pigments are simulated at once, which is what we would expect during the typical workflow in a more complete program. We observe interactive simulation and rendering times scaling linearly with the complexity of the simulation.

6.3. Comparison with Polygonal Watercolour

Our method is suitable for painting on a flat 2D plane just like any existing system. We previously mentioned that cell-based watercolour algorithms may not be suitable for texturing. On the other hand, a different particle-based model, such as that introduced by Diverdi *et al*. [4] (which we hereby refer to as "Polygonal Watercolour"), may also be adaptable to painting on 3D surfaces. For this, we would suggest applying our discontinuity mapping method from Chapter 3 to draw the polygons.

Given these similarities, we now give a comparison of this method to our own (Figure 6.4) and describe how they produce certain effects.

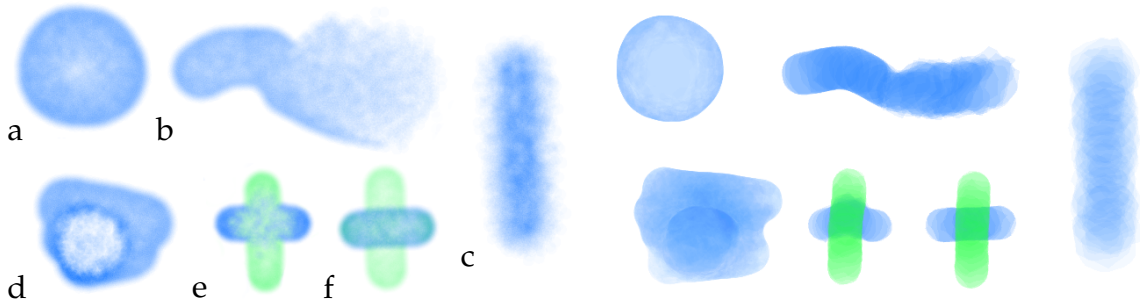


Figure 6.4: Similar effects produced with our algorithm (left) and an implementation of [4] (right).

Effect	Fig 6.4	DiVerdi *et al.*	Ours
Edge Darkening	a	Pre-defined *wet-on-dry* brush	Consequence of the simulation
Rewetting	b	Discretised wet map	Polygonal hulls
Wet-on-Wet	c	Intuitive	Intuitive
Backruns	d	Limited	New fluid brings increased pressure
Colour Blending	e	Overlapping strokes rewetted	Same, but particles also interact
Glazing	f	Front-to-back compositing	Limited
Granulation	-	Paper texture	Random particle clustering

Table 6.3: Both our method and [4] reproduce similar watercolour characteristics, but the way these behaviours emerge in the two systems are different. This table gives a brief overview of the differences.

Edge darkening

In real-world watercolours, darkened edges are observed as a result of evaporation occurring faster around the boundary of a pool of water and getting replenished by water from the inside of the pool, which carries more pigment, causing it to concentrate around the edges of the stroke.

Our method does not explicitly model this process, as there is no evaporation or surface tension in our fluid model. Edge darkening instead occurs because the outside of a stroke has a lower fluid density than the inside, forcing particles outwards. The added friction force when a particle goes outside of the stroke boundary means they do not go too far, causing a few layers of pigment to accumulate around these edges.

In contrast, edge darkening does not occur as a result of physical simulation in the Polygonal Watercolour system. DiVerdi *et al.* instead describe a *wet-on-dry* brush specifically designed to produce this effect by placing splats with an outwards motion bias from the centre of the brush, so that these splats accumulate around the edges of the stroke. This method does have the advantage of producing thinner and more explicit outlines than ours. While the effect is more subtle in our system, it does not require conscious brush choices from the artist and instead occurs on all wet-on-dry strokes, closer to how it does in reality.

Rewetting

Vectorised polygonal hulls, which represent wetted parts of paper, are an important element of our simulation. They are essentially the container for pigment particles, and adding more of them to a stroke will give the particles more room to diffuse.

The Polygonal Watercolour counterpart to this is actually a rasterised "wet map": despite the pigments being vector-based, water is represented by a map with fixed resolution. It could be possible to use our approach instead. However, their wet map also stores a water velocity component, which would have to be adapted to be compatible with a vectorised wet map.

Wet-on-Wet

Both our system and Polygonal Watercolour can faithfully reproduce wet-on-wet painting, where the paper is wetted in advance of adding the paint. DiVerdi *et al.* also describe a special brush which intentionally places a larger pool of wetness around a pair of concentric inner splats, which allows the user to produce a feathered stroke without wetting the paper in advance.

We find that this style of painting benefits from a shorter drying time in our simulation. Using a long drying time, which we find necessary for pleasant-looking wet-on-dry strokes, could cause wet-on-wet strokes to diffuse too much.

Backruns
With our particle-based model handling pigments and water as one, we found that we often observed even accidental backruns in our simulation when mixing paint together. We could reproduce the effect intentionally as well, by adding transparent particles to a stroke when it has almost dried. This acts like water without (or with fewer) pigments, which forces other particles away by pressure. This is not the same as rewetting the canvas by adding new hulls as mentioned above, since wetness and water pressure are properties of the particles and not the hulls.

It is not clear the extent to which backruns occur in Polygonal Watercolour. DiVerdi *et al*. make brief mention of it, but the effect appears much less profound in their examples than in our work. This does, however, mean that unintentional blooms are unlikely to occur within their system.

Colour Blending
When two wet strokes of different colours overlap, we expect to see some mixing from the pigments of either stroke being advected by the water in both. Our model additionally has the characteristic that the particles from both strokes interact, forming unique textures at the intersection.

However, the way in which we mix colours in our model (averaging over RGB space), does not appear to follow the patterns of real-world paint, and we believe it may be possible to achieve more realistic results with physically-based compositing such as the Kubelka-Munk model [5]. The same could also benefit Polygonal Watercolour.

Glazing
Because DiVerdi *et al*. use front-to-back compositing, the glazing effect can be achieved naturally in their work.

This sort of order-preserving compositing is missing from our implementation, but it should also be possible to implement front-to-back compositing within our method: we would have to differentiate between pigments that should be blended together, and pigments that should be composited in order. A user-friendly approach could be as simple as having multiple layers, or "washes", which the user can add, move, delete, or choose which one to paint on, just like in most common 2D image editing software.

Granulation
The unique texture of watercolour paint is largely due to the way in which pigment gets deposited unevenly on paper. DiVerdi *et al*. mention using a rasterised paper texture to drive this effect, although they do not use it in their iPad application, and we left it out of our implementation that was used for Figure 6.4. The primary concern that was expressed with regard to the paper texture was that it could not be represented in vector image format.

Our system does not use a paper texture, but we do observe patterns that are similar to granulation, nonetheless. This occurs in part due to pigments forming random clusters during sampling and simulation, and in part thanks to the complex details emerging from our choice of kernel, as mentioned in Section 4.3.

7

Conclusion

We have developed a system for painting and manipulating watercolour textures on polygon meshes in real-time. Our method is particle-based and has the benefits of infinite resolution rendering and distortion compensation to conform to the geometry of the 3D surface.

Our solution comes in two parts. In the first (Chapter 3), we process user strokes in model space and convert them into a polygonal representation in texture space. On its own, this already provides a useful painting system where polygonal strokes can be rendered on the surface at any magnification.

The second part of our system (Chapter 4) is a pigment model. We use the *discontinuity map* from the first part to represent the wetted parts of the surface and delimit the sampling area of pigment particles. We apply a physically based fluid model to drive particle motion and observe similar effects to those associated with real-world watercolour paint.

7.1. Future Work

Some further improvements would render the method more robust, such as a method for strokes and particles to cross seams in the texture map, using a physically based model for pigment composition [5], and ordering particles so that artists can reproduce the glazing effect.

We note that one of the shortcomings of our particle model is that all particles in the same brush stroke have the same shape and size, which harms the fidelity of small-scale details – particularly darkened edges, which are notably thicker in our model than in other models and in the real world. Stam *et al*. [14] bring more detail out of fluid models by warping the particles, and DiVerdi *et al*. [4] use polygonal particles with several independent vertices. Ideally, we could reach a compromise that allows anisotropic particles shaped by physical simulation, without going so far as to model each vertex of the particles separately.

We previously mentioned that cell-based paint models may not be suitable for this type of texture generation due to their reliance on a square grid. However, future work could explore this possibility. One possibility could be to represent the medium of the surface as a network of connected nodes instead of a grid, with the density of these nodes being scaled to the geometry of the surface.

We would also like to explore the avenue of research in the context of non-photorealistic rendering. This would involve drawing particles of paint in screen space rather than texture space, similar to some other methods [1, 2], but with the benefit of the effects being deliberately produced by an artist using an interactive approach, rather than being automatically generated from an existing texture. This would come with the challenge of dynamically projecting features such as blending or darkened edges from the surface onto the screen.

References

[1] Pierre Bénard et al. “A dynamic noise primitive for coherent stylization”. In: *Computer Graphics Forum*. Vol. 29. 4. Wiley Online Library. 2010, pp. 1497–1506.

[2] Adrien Bousseau et al. “Interactive watercolor rendering with temporal coherence and abstraction”. In: *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*. 2006, pp. 141–149.

[3] Cassidy J Curtis et al. “Computer-generated watercolor”. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, pp. 421–430.

[4] Stephen DiVerdi et al. “Painting with polygons: A procedural watercolor engine”. In: *IEEE Transactions on Visualization and Computer Graphics* 19.5 (2012), pp. 723–735.

[5] Chet S Haase and Gary W Meyer. “Modeling pigmented materials for realistic image synthesis”. In: *ACM Transactions on Graphics (TOG)* 11.4 (1992), pp. 305–335.

[6] Pat Hanrahan and Paul Haeberli. “Direct WYSIWYG painting and texturing on 3D shapes”. In: *ACM SIGGRAPH computer graphics* 24.4 (1990), pp. 215–223.

[7] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. “Smoothed particle hydrodynamics on GPUs”. In: *Computer Graphics International*. Vol. 40. SBC Petropolis. 2007, pp. 63–70.

[8] Ares Lagae et al. “Procedural noise using sparse Gabor convolution”. In: *ACM Transactions on Graphics (TOG)* 28.3 (2009), pp. 1–10.

[9] William E Lorensen and Harvey E Cline. “Marching cubes: A high resolution 3D surface construction algorithm”. In: *Seminal graphics: pioneering efforts that shaped the field*. 1998, pp. 347–353.

[10] Matthias Müller, David Charypar, and Markus Gross. “Particle-based fluid simulation for interactive applications”. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2003, pp. 154–159.

[11] Ken Perlin. “An image synthesizer”. In: *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296.

[12] Nicolas Ray, Xavier Cavin, and Bruno Lévy. “Vector texture maps on the GPU”. In: *Inst. ALICE (Algorithms, Comput., Geometry Image Dept. INRIA Nancy Grand-Est/Loria), Tech. Rep. ALICE-TR-05-003* (2005).

[13] Pradeep Sen. “Silhouette maps for improved texture magnification”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2004, pp. 65–73.

[14] Jos Stam and Eugene Fiume. “Depicting fire and other gaseous phenomena using diffusion processes”. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 1995, pp. 129–136.

[15] Tom Van Laerhoven and Frank Van Reeth. “Real-time simulation of watery paint”. In: *Computer Animation and Virtual Worlds* 16.3-4 (2005), pp. 429–439.

[16] Jarke J Van Wijk. “Spot noise texture synthesis for data visualization”. In: *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*. 1991, pp. 309–318.

[17] Jason C Yang et al. “Real-time concurrent linked list construction on the GPU”. In: *Computer Graphics Forum*. Vol. 29. 4. Wiley Online Library. 2010, pp. 1297–1304.