Vector Graphics Depicting Marbling Flow

Ryoichi Ando[a], Reiji Tsuruno[b]

[a]*Graduate School of Design, Kyushu University, 4-9-1, Shiobaru, Minamiku, Fukuokashi, 815-8540, Japan*
[b]*Faculty of Design, Kyushu University, 4-9-1, Shiobaru, Minamiku, Fukuokashi, 815-8540, Japan*

Abstract

We present an efficient framework for generating marbled textures that can be exported into a vector graphics format based on an explicit surface tracking method (see Figure 1). The proposed method enables artists to create complex and realistic marbling textures that can be used for design purposes. Our algorithm is unique in that the marbling paint on the surface of water is represented as an enclosed contour and is advected by fluid flow to deform the marbling silhouette. In contrast to previous methods, in which the shape is tracked with a concentration density field in Eulerian grids, our approach facilitates greater complexity that is free from grid resolution and per-pixel computation while retaining real-time performance. To forestall the propagation of large vertices, we adaptively resample the contours, exploiting the curvature and the turbulence of the fluid as criteria. At the convection phase, we parallelly advect contour particles on a Graphics Processing Unit (GPU) in addition to applying volume corrections. Finally, we quickly remove extremely thin lines in shapes to remove dozens of vertices. We performed our method with an interactive prototype to demonstrate the robustness of the proposed method in several scenarios.

Keywords: marbling, surface flow, front tracking

1. Introduction

Marbling is a traditional technique that is used for decorating papers with paints floating on a liquid. Marbled patterns are created by dropping paints onto the surface of water and stirring the surface with brushes. The colors are then transferred to a sheet of paper by laying the paper on the surface of the water. Today, due to its ease of use and the vivid and unique patterns it produces, the marbling designs are printed worldwide onto various media, such as booklets and tissue boxes. [13, 18].

One of the most challenging aspects of simulating marbling is to retain the clarity on the surfaces between the different paints and the liquid to depict the features of flow streams precisely. Such clear surfaces are maintained with ox gall and water in real marbling. Several researchers have attempted to simulate this marbling effect with grid-based advection schemes. However, the Eulerian approach comes with built-in "numerical diffusion," which is notorious for blurring clear outlines. A higher-order accurate advection scheme can be used to prevent the dissipation, but it is computationally expensive and suffers from instabilities known as Gibb's oscillations. Among the computer graphics community, a great variety of fluid phenomena, such as smokes and

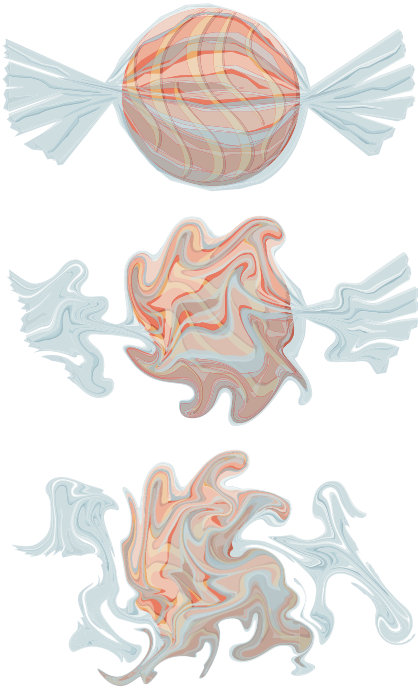


Figure 1: **Marbleized candy clip:** This candy was deformed interactively using our marbling simulator, and it can be described in terms of vector graphics.

liquids, have been explored [4]. However, these techniques are not directly applicable to marbling flow because they are carefully engineered to focus on producing specific types of visual properties.

Our method is related to an explicit surface tracking method, also known as front tracking, which is a technique for tracking propagating interfaces. Front tracking works with Lagrangian surface particles connected to triangles or piecewise linear curves, and it utilizes the underlying motion to capture deforming surfaces. Front tracking often outperforms other Lagrangian methods because the particles are placed only on the surface rather than filling the volume. However, the algorithm tends to be complicated because surfaces can be tangled.

In the proposed method, we track the deformation of the marbling shape with explicit surface particles based on the principle that the contour of the paint region rarely collides by advection due to the divergence free property of free surface fluid flow, as shown by Ando & Tsuruno [3]. Note that the ignorance of topology is only true for free surface flow in continuum level. In contrast to liquid animation, where the liquid domain merges or splits vividly over time, the liquid domain of free surface flow is usually fixed. This strategy makes the algorithm simpler and intuitive because the topological changes can basically be ignored. However, if no topological changes are taken into account, the number of vertices grows limitlessly as the contour stretches, which significantly slows the simulation. To permit the simulation of proportionately larger surfaces, we run a fundamental algorithm on the GPU in a parallel manner, and we resample the contour adaptively, watching for the local vorticity and curvature of the surfaces to approximate the shape with fewer vertices without losing much visual detail. Even though the contour is essentially collision free, because the surface is discretized over space and time, collisions can be produced due to numerical error. We found that this error does not produce significant visual artifacts, but as an option we also remove thin line regions that are almost invisible to remove a large number of vertices and collisions. Consequently, our algorithm runs reasonably fast in terms of shape complexity. Our marbling simulation runs on an underlying velocity field of fluid, which is generated in real-time in response to the user's interactions. The characteristics of marbling deformation are controlled by the behavior of the velocity field. However, because the resolution of fluid flow is rather coarse, the tracker produces slight volume loss at each time step, which accumulates over time. To maintain concentration constant volume of the fluid, we slightly move surfaces in normal directions to effectively correct the error.

1.1. Simulation Overview

For each time step, our marbling simulator takes the following five steps in order.

1. **Convection:** We start the simulation by generating the underlying fluid velocity with a uniform grid and semi-Lagrangian method. We advect the contour points explicitly by fetching velocities from sixteen grid points using cubic spline interpolation with the fourth-order Runge-Kutta method. We also subdivide the stretched contour by rewinding time to find a more accurate subdivision point than that obtained with linear subdivision.
2. **Adaptive Sampling:** We resample the contour points according to the local curvature, vorticity and distance from the opposite contour to depict the shape with fewer vertices.
3. **Volume Error Correction:** If we advect the marbling shape under the fluid motion with a coarse grid, the volume error accumulates slowly over time. We quickly correct this error by inflating or shrinking the entire shape toward normal directions with slight changes.
4. **Shape Simplification:** We dynamically remove extremely thin lines that are nearly invisible so the simulator will run more smoothly. To do this, we detect and cut such regions, and then reconstruct the contour connections. The simplification test can be triggered at any time, particularly when the user desires, because the collision can basically be ignored.
5. **Rendering and Export:** The deformed marbled shape is rendered through the graphics hardware or exported in an editable vector graphics format.

2. Previous work

Our work is related to two categories of studies: the artistic expression of fluids and surface tracking methods.

The most relevant field is the direct simulation of marbling. Acar & Boulanger [2] attempted to reproduce visual marbling effects using a physically derived flow model. They observed surface flow based on mesoscale dynamics, and they produced fluctuation effects at different scales. To advect clear silhouettes under Eulerian grids, they employed B-spline interpolation and extended the range of concentration temporarily in the semi-Lagrangian advection phase. However, Eulerian grid approximation is limited in terms of the degree of resolution if we wish to obtain a reasonable simulation.

If we simulate with high resolution, a great deal of per-pixel computation and memory is required. Zhao et al. [33] developed a real-time marbling simulator that fully runs on the GPU. They employed a third-order accurate, but fast, unsplit semi-Lagrangian constrained interpolation profile method to reduce the numerical dissipation while retaining stability. Although they achieved 24 frames per second (FPS) at 1680×1050 grid resolution, the method was still inadequate for printing large materials because the resolution was approximately 5×3 square inches at 350 dpi. Acar [1] also proposed a level set-based system that provides a flexible environment for the user to generate traditional marbling patterns in high resolution. However, real-time feedback is still computationally expensive with this method.

Eden et al. [6] proposed a method for rendering liquids in a cartoon-style manner. By exploiting a physically created fluid surface, they rendered the effect by emphasizing the properties of the liquid's shape and motion, which were inspired by the abstraction and simplification of cartoon animations. This method resembles our own in that both have clear silhouettes and few colors. However, they used an implicit contouring method as the underlying liquid animation. Hence, the thin line detail is inherently smoothed out before it is stylized.

Selle et al. [28] introduced a technique for generating cartoon-style animations of smoke. Based on a physically-based simulated output, they traced marker particles and rendered them using depth buffer differences to generate clear smoke animations. McGuire & Fein [19] extended this technique and developed a system for rendering real-time animations of smoke in addition to introducing a novel self-shadowing algorithm. However, these methods cannot track precise thin lines with the level of detail seen in marbling-like fluid flow because each particle (primitive) is visibly large.

Witting [30] presented a system that uses computational fluid dynamics to produce two-dimensional animated films. Compressive fluid dynamics were employed and were restricted to two dimensions to develop user-controlled fluid experiences. The work presented in this paper shares his motivation in that it offers artists fluid environments for design purposes, we focus particularly on a vector graphics format.

Pedersen & Singh [26] developed the automatic synthesis of labyrinthine and maze structures with curves on 2D manifolds which are updated by adding Brownian motion, fairing, and attraction-repulsion forces. Although this type of simulation is quite different from ours, the nature of the strategy is similar to ours in that a simple closed path is evolved with adaptively resampled curve points and underlying forces. In our model, curves are rendered as solid polygons, which depict clear marbling silhouettes.

Our work is also comparable to surface tracking techniques used in physically derived liquid animations. This topic has a long historical background in the literature. We briefly review these works because they mainly focus on a well-designed topological change algorithm, which is orthogonal to our approach. Roughly, a free surface is tracked using three approaches: Eulerian grids, Lagrangian particles and explicit surfaces.

Among the Eulerian approaches, Hirt & Nichols [16] proposed a *volume-of-fluid* method that constructs an approximation of the interface from cells that contain portions of the interface. Osher & Sethian [25] proposed a level set method, which has become dominant in industrial applications. In the level set method, a signed distance function ϕ from the interface is advected, and the surface is implicitly located where $\phi = 0$. The advantage of these two Eulerian based approaches is that they do not require a special post-process to handle topological changes. However, they do suffer from numerical diffusion or loss of mass conservation [8].

Among the Lagrangian methods, Enright et al. [7] extended the level set method with Lagrangian particles and increased accuracy. Harlow & Welch [15] advected particles in *marker-and-cell* grids to identify deforming surfaces. Müller et al. [24] and Harada et al. [14] incorporated *Smoothed Particle Hydrodynamics* (SPH) and applied it to visual simulations. These methods introduced blobby or splitting artifacts to thin details.

Meanwhile, a number of explicit front tracking methods have been proposed, both for computational dynamics [5, 11, 22, 29, 31] and for image processing [17, 20]. In front tracking methods, a surface interface is constructed with explicit surface elements and is advected by the underlying motion. The front tracking method offers a precise representation of the interface free from grid resolution or numerical diffusion; however, it suffers from self-intersections and complexity. For example, they detected topological changes by uniform grid traps or by searching close edges, a process that can be challenging to execute without losing surface detail. Except for the fact that they tackle complex topological changes, their method is comparable to ours. If desired, our method can also be applied to topological changes for extremely thin lines. In our case, however, we only split thin lines so that little detail will be removed.

3. Method

Before we discuss the concrete algorithm, let us first briefly outline the basic workflow, which is shown in

Figure 2: **Workflow of our method:** We first place closed contours of a painted region in the fluid field, then we advect or stretch them along the fluid flow. The rendering of the region is performed the same way a concave polygon is rendered.

Figure 2. To depict vector graphics, we first place closed contours in a fluid flow and then advect them along the fluid flow.

3.1. Contour Advection

To advect contours, we must generate the underlying fluid motion. For simplicity, we use the finite differential grid and semi-Lagrangian advection method to generate the fluid velocity field. Once we have the fluid flow, we can place closed contours on the surface of the water and advect them. In our system, we represent contours as a sequence of discrete points that are connected, and we advect them in the Lagrangian manner. In a numerical method, it is inevitable that advection will cause collisions. To prevent significant collisions, we employ a fourth-order accuracy Runge-Kutta scheme. We denote this scheme as

$$\boldsymbol{p}^{t+\Delta t} = \phi(\Delta t, \boldsymbol{p}^t, \boldsymbol{u}^t), \tag{1}$$

where $\boldsymbol{p}^t$, $\boldsymbol{u}^t$ and ϕ denote the position of the contour vertex, the velocity of the fluid field and the Runge-Kutta scheme at time t, respectively. The velocity of the fluid at vertex point $\boldsymbol{p}$ is obtained by interpolating the sixteen surrounding grids using cubic interpolation. Note that the velocity beyond the grid wall is set to zero. This type of interpolation is more accurate than bilinear interpolation. However, our interpolation sometimes overshoots due to the inherent oscillation. We improve this instability by clamping the value to one of the velocities of the four surrounding grids.

After all of the vertices have been advected ($t \leftarrow t + \Delta t$), we measure the distances between the connected vertices. If the distance exceeds a threshold d, we insert a new vertex $\boldsymbol{p}_{\text{new}}$ midway between the relevant pair of vertices at the previous time step, and we advect it such that

$$\boldsymbol{p}_{\text{new}} = \phi\Big(\Delta t, \frac{1}{2}(\boldsymbol{p}_0^{t-\Delta t} + \boldsymbol{p}_1^{t-\Delta t}), \boldsymbol{u}^{t-\Delta t}\Big), \tag{2}$$

where $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$ denote the positions of each vertex in the pair. This time-rewinding method helps position the subdivided vertices more accurately than would linear subdivision with rapid advection, as seen in Figure 3. The effect may be slight. However, it also helps to reduce faceted edges or collisions, which allows us take larger time steps, but, if the distance is less than $d/2$, we collapse the vertex. This process is repeated until all connected vertices are separated by distances between d and $d/2$

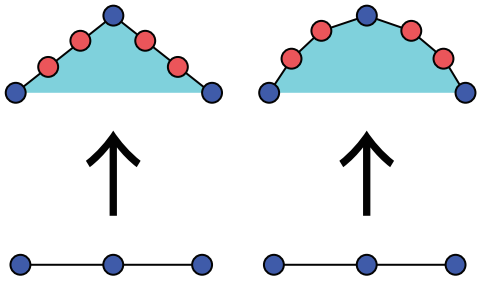


Figure 3: **Comparison with linear subdivision and time-rewinding subdivision:** The red oval represents the subdivided points, and the blue oval represents the original advected points. Unless the contour is subdivided only once per segment, our time-rewinding subdivision mimics the round feature of the marbled contour more effectively than linear subdivision.

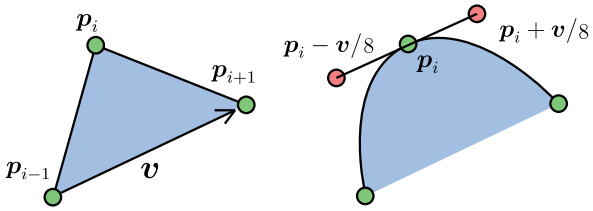


Figure 4: **Procedure for setting anchor points:** The bezier description helps blur faceted edges. The two anchor points are set at the front and back direction of the tangent vector of the point.

3.2. Rendering and Export

To export a region as a vector graphic, we write the shape as a regular concave polygon. Starting from an arbitrary vertex, we move to the next connected vertex and write its position in sequence. In our prototype, we use the scalable vector graphics format with a path entry to export the vector graphics file into an actual file. The rendering of the concave polygon is performed efficiently by the stencil method [32]; for every pair of connected contour points $(\boldsymbol{p}, \boldsymbol{q})$, we draw a triangle polygon $(\boldsymbol{0}, \boldsymbol{p}, \boldsymbol{q})$ onto a framebuffer while inverting the existing values between 0 and 1. Finally, the solid region is filled with value 1. Rendering solid regions using the GPU is described in a later section. The contours can be roughly described with a Bézier curve by giving each point a

pair of anchor points to avoid faceted curves. For each point $\boldsymbol{p}$, a pair of anchor points $\boldsymbol{q}_0, \boldsymbol{q}_1$ are computed as $p_i \pm v/8$, where $\boldsymbol{p}_i$ denotes the position of the i th vertex on a contour, and $\boldsymbol{v} = \boldsymbol{p}_{i+1} - \boldsymbol{p}_{i-1}$. Figure 4 illustrates the graphical procedure. This interpolation is actually a family of Catmull-Rom splines. Note that the B-spline interpolation may seem natural, but it is not widely used in the editable vector graphics format.

3.3. Adaptive Refinement

With the aforementioned steps, creating a marbling silhouette with vector graphics may be possible. However, the implementation of this method can easily result in excessive computation due to the rapidly increasing propagation of vertices over time. In this section, we introduce an "Adaptive Refinement" method to suppress this propagation.

From our observations, we found that we could omit vertices where the contour is not strongly curved, and, in places where the contour is tightly curved or the fluid is eddying, we needed to insert more points. Hence, we tuned the distance threshold d, and we controlled it by taking its product with the local curvature $c_i^{\text{curvature}}$ and the turbulence $c_i^{\text{turbulence}}$ so that

$$d_i = d_{\max} c_i^{\text{curvature}} c_i^{\text{turbulence}} + \varepsilon, \tag{3}$$

where $d_{\max}$ denotes the maximum space between vertices. The value ε is a limit constant to avoid 0. Typically, $d_{\max}$ would be around 5ε to 10ε and ε is approximately the size of a pixel. To determine the local curvature, we employed a normalized second derivative:

$$c_i^{\text{curvature}} = \exp(-|\kappa|), \tag{4}$$

where κ denotes the finite differential curvature value. For local turbulence, we considered the local vorticity

$$c_i^{\text{turbulence}} = \exp(-|\nabla \times \boldsymbol{u}(\boldsymbol{p}_i)|), \tag{5}$$

where $\boldsymbol{u}(\boldsymbol{p}_i)$ denotes the velocity of the fluid at position $\boldsymbol{p}_i$. Using such an adaptive d, we can unnoticeably remove a large fraction of the vertices while increasing the quality where contours have high curvature or turbulence. Notice that both $c_i^{\text{curvature}}$ and $c_i^{\text{turbulence}}$ range between 0 and 1. This strategy works well for most cases, although when d is sufficiently large, it often fails to capture small perturbations of the fluid flow. To compensate for this drawback, we also diffuse d_i along the contour. After we calculated each d_i value at $\boldsymbol{p}_i$, we iteratively assigned to d_i a value as follows:

$$d_i \leftarrow \sum_{n=-w}^{w} G(\alpha, n) d_{i+n}, \tag{6}$$

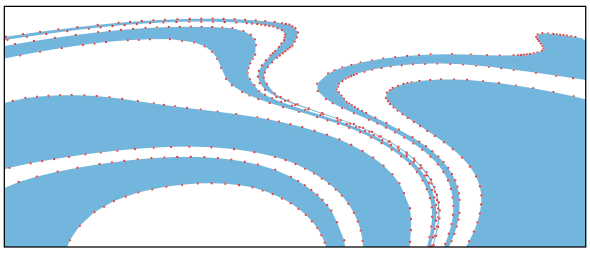


Figure 5: **Adaptive Refinement:** The small, red circle marks represent vertices to track. Note that the space of vertices is sensitive to curvature. To avoid collisions, more vertices are inserted where the contours are highly curved.

where w and $G(\alpha, x)$ denote the window radius and a Gaussian function, respectively. Typically, $w = \text{floor}(kd_{\max}^{-1})$ where k is a scaling parameter. By diffusing d_i, a rapid agitation around the vertices triggers its neighbors to have a small d_i value; as a result, the rapid motion is well captured by the neighboring vertices. We illustrate the effect of Adaptive Refinement in Figure 5.

The Adaptive Refinement technique prevents large numbers of unnecessary vertices from being inserted. However, it produces intersections around thin or adjacent regions. Even though such collisions should not occur in the continuum level, because the numerical model is discrete, such collisions are inevitable. Fortunately, we found that slight contour collisions do not contribute significantly to visual artifacts. However, for the purpose of generating high DPI images such as large posters, one may want to remove collisions as much as possible. In such cases, we add a proximity term $c_i^{\text{proximity}}$ in equation (3) as

$$c_i^{\text{proximity}} = 1 - \exp(-d_i^{\text{proximity}}), \tag{7}$$

where $d_i^{\text{proximity}}$ denotes the distance function from the nearest opposite contour. This may magnify the contour subdivisions, but it reduces collisions.

3.4. GPU Acceleration

Although our algorithm runs at an interactive rate on a CPU, it is limited to only around 40,000 vertices (10 FPS). In this section, we introduce a GPU accelerated algorithm to boost real-time performance. Similar to particle markers, contours can be tracked with a collection of particles. In our model, however, the number of particles varies, and explicit connection information is dynamically reconstructed. A straightforward approach may be to store each initial vertex in individual kernels and watch them propagate, however, the computational

cost varies among kernels because contour growth is uneven, resulting in a slowdown. To disperse the computation evenly, we introduce a method for task diffusion.

Our GPU acceleration strategy consists of the following three steps: (i) contour advection, (ii) contour subdivision and (iii) task diffusion. In our system, each vertex contains information about its position and a reference to the next connected vertex. We refer to this kind of vertex as a "task." On a GPU, a contour is decomposed into a collection of tasks, each one referring to the next task. The tasks are then stored in separate kernels (Figure 6 (a)(b)). The devices memory storage is pre-allocated and mapped onto kernels.

In the advection phase, the positions of the tasks stored in each kernel are advected in parallel, as in section 3.1. In the contour subdivision phase, each kernel probes every stored task for its distance from the referenced task and then subdivides or collapses it if necessary, as described in section 3.1,3.3. Inserted vertices are stored in the kernel of the originating task. The detailed workflow of this advection is illustrated in Algorithm 1.

In this phase, the number of tasks stored in the kernels become uneven. In the task diffusion phase, we choose random pairs of kernels, and we compare their numbers of stored tasks. Where the opposite kernel holds a smaller number of tasks, we move the tasks into the opposite kernel. For consistency, we choose a pair $(\text{kernel}_i, \text{kernel}_{(i+r) \bmod n})$, where kernel_i denotes the i th kernel, r is a shared random integer and n is the number of kernels. We avoid conflict by letting i be even and r be odd. This task diffusion phase averages out the number of tasks among the kernels, which disperses the advection and subdivision costs evenly. The detailed procedure for the task diffusion is illustrated in Algorithm 2.

To evaluate equation (7), vertex indices are sorted into uniform grids as described in [12], and we constrain the searches within the neighboring grids. Solid regions are rendered using the stencil method by randomly writing task polygons $(\mathbf{0}, \boldsymbol{p}, \boldsymbol{p}_{\text{reference}})$ into a vertex array, where $\boldsymbol{p}$ denotes the position of a task and $\boldsymbol{p}_{\text{reference}}$ denotes the position of the referenced task. Empty slots are filled with null polygons (Figure 6 (c)). In our case, the procedures, including a description of the underlying fluid, was ported using Compute Unified Device Architecture (CUDA).

3.5. Volume Correction

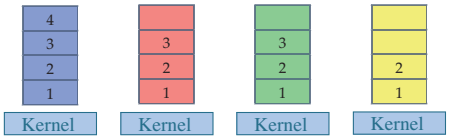
In recent years, preserving volume has been a common problem for deformable objects in the field of computer graphics, including character animation and phys-

Algorithm 1 GPU_Advect

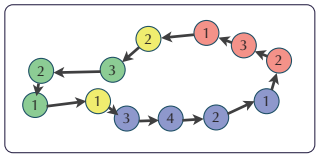
```
// Advection Phase
for all kernel k_i in parallel do
    for all p_i mapped with k_i do
        p_i^{t+Δt} = φ(Δt, p_i^t, u^t)
    end for
end for
// Compute d_i^*
for all kernel k_i in parallel do
    for all p_i mapped with k_i do
        d_i^* = d_max c_i^curvature c_i^turbulence + ε
    end for
end for
// Compute Diffused d_i^* as d_i
for all kernel k_i in parallel do
    for all p_i mapped with k_i do
        d_i ← Σ_{n=-w}^{w} G(α, n) d_{i+n}^*
    end for
end for
// Subdivision Phase
for all kernel k_i in parallel do
    repeat
        for all p_i mapped with k_i do
            if ||p_i^reference − p_i|| < d_i/2 then
                collapse p_i^reference
                reconstruct connection
                pop p_i^reference from k_i
            else if ||p_i^reference − p_i|| > d_i then
                subdivide p_new = Eq. (2)
                insert p_new into k_i
                reconstruct connection
            end if
        end for
    until no subdivision or collapse made
end for
```

Algorithm 2 GPU_Diffuse

```
r ← 2*random()+1
for all even number of kernel k_i in parallel do
        for all p_i mapped with k_i do
            n_i ← number of mapped tasks with k_i
            n_{i+r} ← number of mapped tasks with k_{i+r}
            if n_i > n_{i+r} + 1 then
                move a task from k_i to k_{i+r}
            else if n_i < n_{i+r} − 1 then
                move a task from k_{i+r} to k_i
            end if
        end for
end for
```



(a) Kernels and stored vertices



(b) Connection Map

0	1	2	0	1	2	0	1	1	0	1	3
0	2	1	0	2	3	0	2	3	0	2	3
0	3	4	0	3	1	0	3	2	0	0	0
0	4	2	0	0	0	0	0	0	0	0	0

(c) Vertex Buffer

Figure 6: **Device memory map of task vertices:** This example is configured with four kernels and four maximum slots for each kernel. The arrows in the connection map indicate the references task (vertex). The vertex buffer is filled based on the references and where the vertices belong.

ically based animations, where a number of models have been proposed to achieve natural transformation while correcting volume errors well.

In our approach, we extended the method of Funck et al. [10], Müller [23] and Rohmer et al. [27] to fit our 2-dimensional applications, and solve the problem by stretching the vertices in their normal directions with the minimal sum of movement so that the initial volume is recovered, as illustrated in Figure 7. Let $\boldsymbol{p}$, $\boldsymbol{n}$ and $\mathcal{V}(\cdot)$ be vertices array, displacement normal vector and area function, respectively. As derived by Green's law, the volume (Area) of $\boldsymbol{p}$ is given by

$$\mathcal{V}(\boldsymbol{p}) = \frac{1}{2}\sum_i (\boldsymbol{p}_i \times \boldsymbol{p}_{i+1}) \cdot \boldsymbol{e}_z, \tag{8}$$

where $\boldsymbol{e}_z$ denotes z-direction of unit vector. Let $\boldsymbol{p}_{\text{initial}}$ and $\boldsymbol{p}_{\text{correct}}$ be initial vertices array and corrected vertices array where each component is denoted by $\boldsymbol{p}_{\text{correct}}|_i = \boldsymbol{p}_i + x_i\boldsymbol{n}_i$. Normal vectors for each vertex are given by $\boldsymbol{n}_i = (\boldsymbol{p}_{i+1} - \boldsymbol{p}_{i-1})/\|\boldsymbol{p}_{i+1} - \boldsymbol{p}_{i-1}\| \times \boldsymbol{e}_z$. Similarly to Müller [23], the volume change of correction is well approximated by

$$\Delta\mathcal{V}(\boldsymbol{p}_{\text{correct}}, \boldsymbol{p}) = \frac{1}{2}\sum_i x_i \|\boldsymbol{p}_{i+1} - \boldsymbol{p}_{i-1}\|, \tag{9}$$

where $\Delta\mathcal{V}(\boldsymbol{p}, \boldsymbol{q}) = \mathcal{V}(\boldsymbol{p}) - \mathcal{V}(\boldsymbol{q})$ denotes the difference of volume between two sets of closed contour vertices. If we compact a known part into a new vector $\boldsymbol{c}$, where

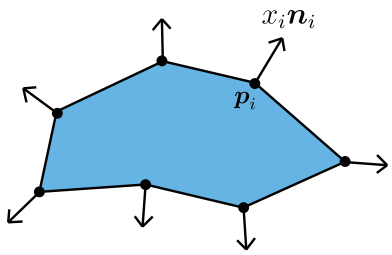


Figure 7: **Our volume correction strategy:** The volume is corrected by stretching or shrinking series of vertices toward their normal directions in such a way that the sum of every movement at the vertices is kept to a minimum.

each component is denoted by $c_i = \|\boldsymbol{p}_{i+1} - \boldsymbol{p}_{i-1}\|$, the volume correction is written with the following vector product:

$$\Delta\mathcal{V}(\boldsymbol{p}_{\text{correct}}, \boldsymbol{p}) = \boldsymbol{c} \cdot \boldsymbol{x}/2, \tag{10}$$

where $\boldsymbol{x} = [x_0, x_1, \cdots, x_i]$. Because we want this correct function to actually cancel the produced error,

$$\boldsymbol{c} \cdot \boldsymbol{x}/2 = -\Delta\mathcal{V}(\boldsymbol{p}, \boldsymbol{p}_{\text{initial}}) \tag{11}$$

in such a way that the sum of the vertices movement should be kept to a minimum. Hence, our problem is formulated as

$$\begin{cases} \text{minimize} & \|\boldsymbol{x}\|^2 \\ \text{subject to} & \boldsymbol{c} \cdot \boldsymbol{x} + 2\Delta\mathcal{V}(\boldsymbol{p}, \boldsymbol{p}_{\text{initial}}) = 0, \end{cases} \tag{12}$$

We solve this constraint minimization problem by translating it into an unconstrained format using Lagrangian multipliers. Finally, the analytical solution of the equation is expressed as

$$\boldsymbol{x} = -\frac{2\boldsymbol{c}}{\|\boldsymbol{c}\|^2}\Delta\mathcal{V}(\boldsymbol{p}, \boldsymbol{p}_{\text{initial}}). \tag{13}$$

This approach runs fast because the equation is evaluated explicitly. After this correction, the initial volume is recovered approximately. If we have the proximity value computed in equation 7 beforehand, we may scale the normal vector to ensure that the vertices will not pass through the opposite contours around the adjacent regions as $\boldsymbol{n}_i \leftarrow \left\{1 - \exp(-d_{\text{proximity}})\right\}\boldsymbol{n}_i$. Our consequent volume correction algorithm is illustrated in Algorithm 3. When compared to that of Müller [23], our method is slightly more complicated. However, the volume error is guaranteed to be corrected with smaller amount of changes of vertices than that of Müller [23]. Our volume conservation can be coupled with multiple separated regions without special modification, but, in such cases, although the volume of an individual island may not be corrected, the sum of all areas will be corrected.

Algorithm 3 CorrectVolume($\boldsymbol{p}, \boldsymbol{p}_{\text{initial}}$)

$e \leftarrow \Delta\mathcal{V}(\boldsymbol{p}, \boldsymbol{p}_{\text{initial}})$
Allocate $\boldsymbol{c}$
$c_{\text{sum}} \leftarrow 0$
for all c_i **do**
    $c_i \leftarrow \|\boldsymbol{p}_{i+1} - \boldsymbol{p}_{i-1}\|$
    $c_{\text{sum}} \leftarrow c_{\text{sum}} + c_i^2$
end for
for all $\boldsymbol{p}_i$ **do**
    $\boldsymbol{p}_i \leftarrow \boldsymbol{p}_i - 2\boldsymbol{n}_i(ec_i/c_{\text{sum}})$
end for

3.6. Shape Simplification

In practice, we found that the number of vertices propagates quickly over time as it develops swirly curls. Because the simulation cost scales linearly with the number of vertices, it is highly preferable that we remove the vertices in extremely thin lines that are hardly visible to maintain real-time interaction as long as possible. In this section, we will provide a detailed outline for performing this simplification. Note that this step can be omitted at the user's discretion because it eliminates detail to varying degrees. Our approach for removing and sewing the mesh is a modified version of the method of Brochu et al. [5], where the meshes are reconstructed by finding close surfaces under a predefined tolerance. However, we add some additional operations to remove thin lines, as requested by the user. The user may also consider choosing other methods that offer precise topological changes under explicit surfaces, as mentioned in section 2. Because we already have computed the proximity value in equation (7), this method is much easier and less costly to employ than other grid trap based methods. In the first stage, we start by finding vertices that are close enough to each other and mark them with reference to the closest vertex (Figure 8a).

These close vertices will hopefully be marked together. Unfortunately, some pinched vertices at the thin line regions remain unmarked due to the noisy distributions of the vertices. We then decide whether the unmarked vertices are those at the thin regions by searching other marked vertices from that position with backward and forward walking connections within the user's defined distances. If marked vertices are found in both the backward and forward directions, we mark them too. (Figure 8b). In the second stage, we detect splittable vertices by searching a vertex, which is marked, and the next unmarked forward vertex, as shown in Figure 8c. We then collapse the forward connections of the splittable vertices and replace them with the references to the proximity vertices. Finally, after unmarking the

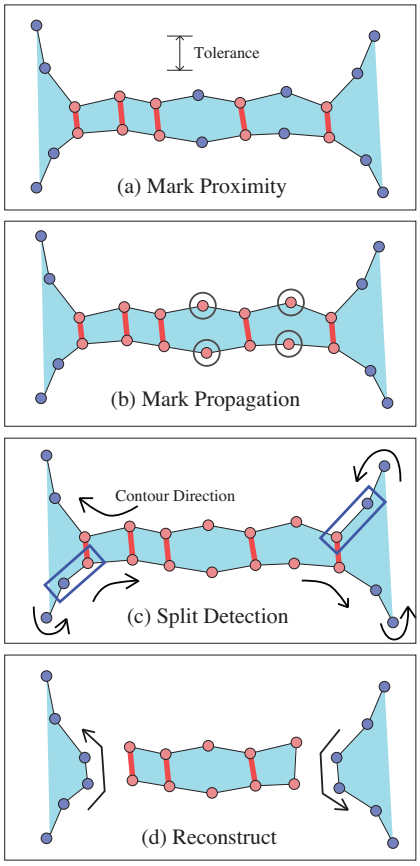


Figure 8: **Simplification Procedure:** (a) Vertices that are closer than a predefined tolerance are marked. The red circle represents marked vertices, and the blue circle represents unmarked vertices. The red bold lines indicate the proximity references. (b) The marked vertices are propagated. (c) Marked vertices that have unmarked vertices on their next connection are labeled as splittable vertices. (d) The forward connections of the splittable vertices are collapsed and replaced with references to the proximity vertices.

splittable vertices, the rest of the marked vertices are removed to remove the thin regions. This simplification is fast because the algorithm is parallel, but it sometimes fails to sew new contours. To detect such failures, we check whether the connection is valid, and we abandon the simplification if we find invalid connections. In our observation, this kind of failure takes place if the regions that have numerous branches suddenly approach each other. The algorithm is still practical though, we can use it to extend the limits of the runnable time steps.

4. Results

A comparison with competing methods is shown in Figure 9. The particle level-set result was generated

Our method with 5742 Vertices | Particle level-set method | Eulerian Advection with Acar's Edge Accentuation

Figure 9: Competing methods for a shared fluid field. The particle level-set method breaks the thin lines, and the Eulerian advection filters out details, whereas our method retains detailed features.

with an existing library [21]. The Eulerian advection was generated with a semi-Lagrangian advection scheme. The clear edges were enforced using the concentration transformation function introduced by Acar & Boulanger [2]. All of these results were computed for shared fluid motion. Our computation took seven seconds, while the others took more than ten minutes for a 1,600x1,600 grid. The thin line property was well maintained with our method, whereas the particle level-set method broke the thin lines, and the Eulerian advection filtered them out, even at high resolutions. Note that a simplification test was not performed in this comparison but will be described in a later section. The performance with a single threaded CPU and GPU is illustrated in Figure 10(a). Our implementation for a complex scene on a GPU runs several times faster on a GPU than on a CPU. Note that the performance of the GPU depends on the size of the pre-allocated GPU memory because the entire memory space is sent into the rendering pipeline. The maximum size of this example was around 100,000 vertices.

However, Figure 10(a,b) reveals two critical limitations of our method. First, the tractable number of time steps is limited to around a few thousand (though this number depends heavily on the environment), which can be easily reached. Second, the contours are interactively tracked only up to around 100,000 vertices. Despite these limitations, we found that our method is quite practical for creating organic fluid components in terms of the number of vertices (See Figure 10(c)).

The effects of volume correction are shown in Figure 11. In this example, the two figures are generated under a 16x16 coarse flow grid with the left one uncorrected and right one corrected. These may look similar at a glance; however, the volume of the left figure has been

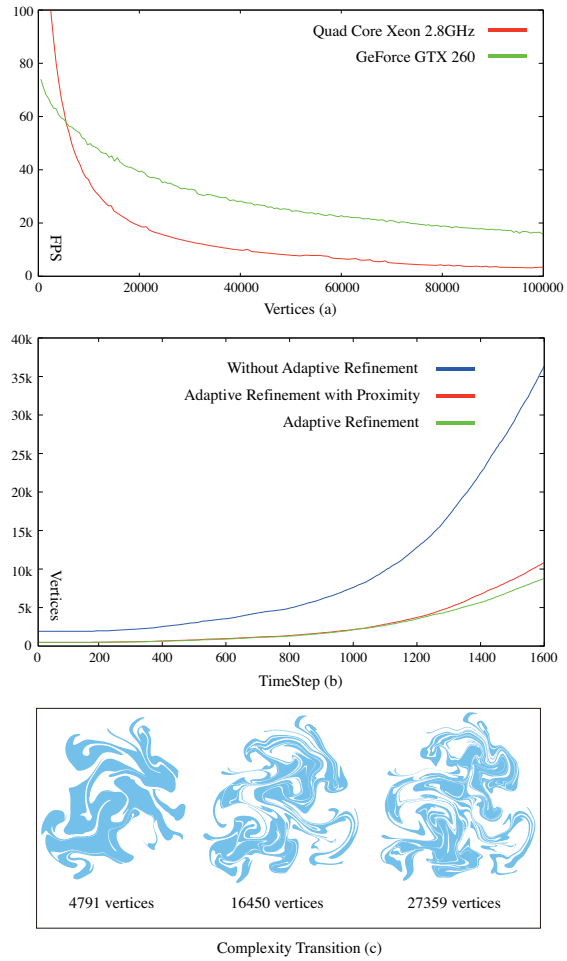
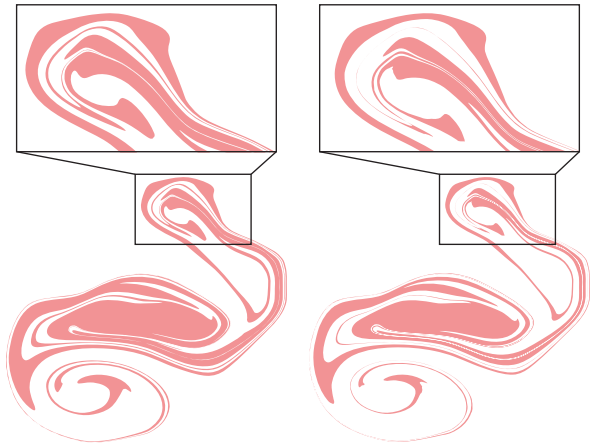


Figure 10: The performance of our method with Adaptive Refinement: (a) At 10,0000 vertices, the CPU recorded 4 FPS while the GPU recorded 17 FPS. (b) Vertex propagation in time. The number of time steps that can be interactively simulated is limited due to the sharp increase in the number of vertices. (c) Generated vector graphic and its number of vertices.

inflated to a size that is 40% larger than that of the initial state. In the right figure, the correction was performed at every time step, but we did not experience critical lags in terms of real-time interaction. As shown in Figure 12(b), the correction only slows performance between 5% and 20%. From our observations, however, the cost is almost ignorable if the number of vertices exceeds 10,000.

The quality and precision of the correction is illustrated in Figure 12(a). Without the correction, the volume error varies unsteadily during the simulation, whereas with the correction, the volume error is completely canceled. However, our correction step only adjusts the volumes without accounting for the local fea-



(a) Without Volume Correction (40.4 % Volume Increased)

(b) Volume Corrected (0.000 % Volume Error)

Figure 11: **Comparison of volume changes between uncorrected and corrected deformation:** The two shapes were advected under a shared motion except that the volume of the right shape (b) was corrected at every time step.

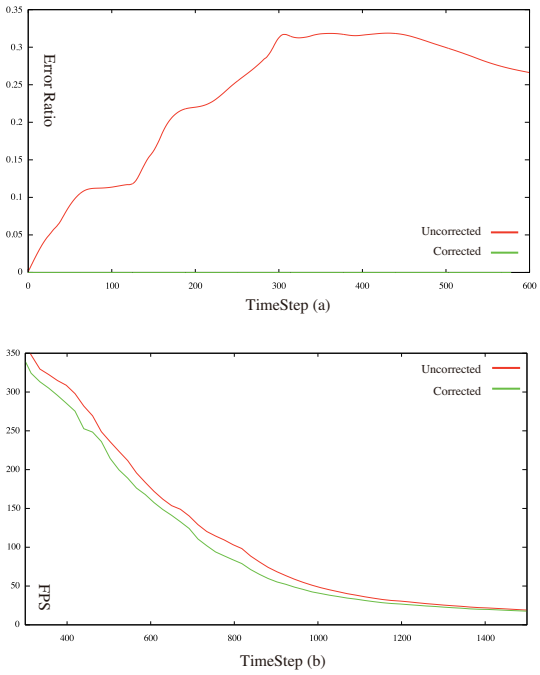


Figure 12: **Volume error, the cost of correction induced by deformation and the complexity table:** (a) Under the coarse flow grid, the volume error in the deformation accumulates slowly over time if no correction is made. With the volume corrected, such volume errors are completely canceled. (b) Our simulation of the volume correction only has a cost between 5% to 20% of that without correction. When it comes to hundreds of time steps, its latency is almost ignorable. (c) These marbling silhouette show how the shape develops as it subdivides with number of vertices.

tures. As a result, the user may observe unwanted feature changes around curves or shared boundaries between two regions. For such special local curves, we can suppress the artifacts by scaling normal vectors by zeros, but then volume correction may not work effectively.

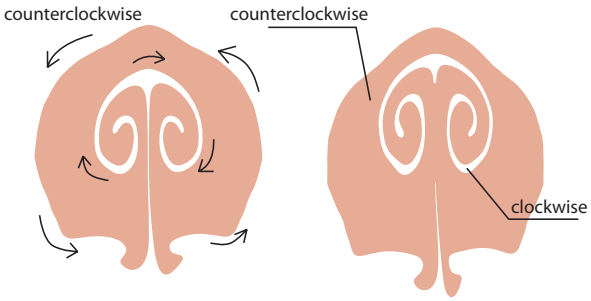
The effect of the simplification step is shown in Figure 13. In this figure, the tolerance is set to either $\varepsilon/2$ or $\varepsilon/4$, where ε denotes the minimum sampling rate defined in equation 3. If the tolerance is set to the larger value, as in Figure 13(a), a significant portion of thin details will be removed, which results in featureless stencil art. However, if tolerance is set to the smaller value, as in Figure 13(b), the thin details may be kept, but the simplification will rarely be triggered. Therefore, we designed the tolerance degree to be controlled by the user, allowing him or her to choose the amount of detail. This simplification test can be completed at arbitrary time steps, especially when the user desires it, because our marbling flow is essentially collision free. However, the test should be processed at least once at fixed, large intervals because numerical advection produces collisions and increases chance of failure for sewing new separate contours. Note that our simplification algorithm is not stable because it can fail to detect adjacent contours and special care should be paid, as described in section 3.6. Our simplification algorithm can be naively coupled with the volume correction step because, even though the topology has changed, we can still differentiate the volumes with equation 8. It is also still naive to export the topological complex shape into a vector graphics file using complex path attributions. For instance, some vector based programs treat clockwise paths as "holes" and counterclockwise paths as "solid regions" and vice versa. Fortunately, our simplification algorithm naturally handles this problem, as shown in Figure 14. In this example, the direction of connection is counterclockwise before merging (a). After merging (b), the connection of the solid region is counterclockwise, but the connection of the hole region becomes clockwise.

5. Applications

We built an interactive prototype both on a Quad Core Xeon 2.8 GHz processor and a GeForce GTX 260 running Linux. The underlying fluid was computed with a 64×64 grid resolution. With our prototype, users are allowed to drop pigments on the surface of the water and disturb it by simply dragging the mouse. In this sec-

(a) Tolerance = $\varepsilon / 2$ (b) Tolerance = $\varepsilon / 4$

Figure 13: **Simplification at various tolerance:** If the tolerance is large (a), it often kills the interesting stripe patterns of the marbled shape. In contrast, if tolerance is small (b), the effect would be hardly be noticeable, but the simplification would rarely be triggered. ε is the same value used in equation (3).



(a) Before Merging (b) After Merging

Figure 14: **Directional changes in simplification:** The simplification event changes the direction of vertices connections from counterclockwise to clockwise, which determines whether the path is a "hole" or "solid" in a vector graphics format.

tion we introduce some interesting applications of our method, and we explore the feasibility of our method.

5.1. Marbling and Sumi-nagashi

Figure 15 shows an example of a complex vector fluid generated with our method. When creating this silhouette, the artist experienced responsive interactions just as with real marbling. As can be seen from the figure, our method is powerful enough to design stylish curly shapes similar to those found in marbling or sumi-nagashi. One particularly convenient characteristic of our vector fluid method is that such closed contours are directly translatable into vector-based programs. Figure 16 shows an example of a shape imported into Adobe©Illustrator©and Adobe©Flash©. Because

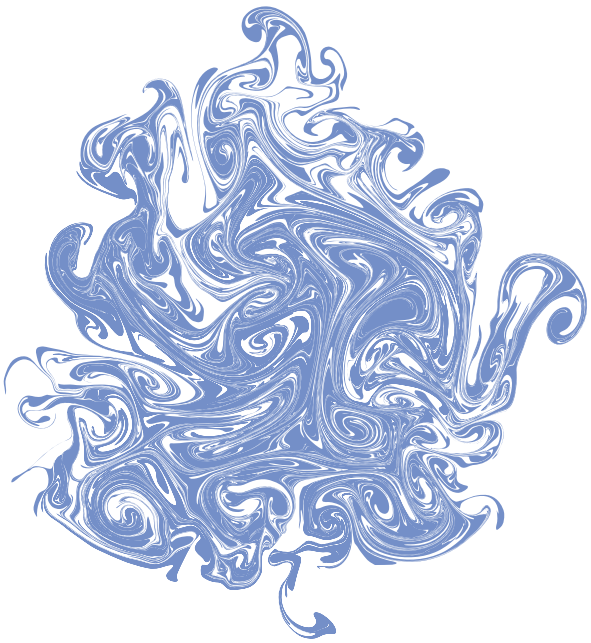


Figure 15: Complex marbling-like silhouette: This vector fluid was created with our GPU prototype. During the simulation, the rendering and interaction were responsive.

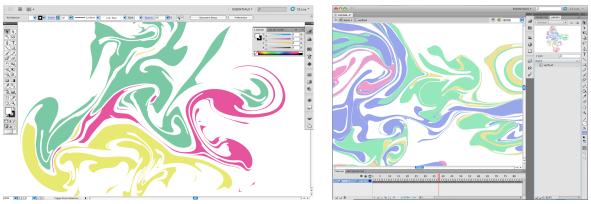


Figure 16: Shape imported into Adobe©Illustrator©and Adobe©Flash©: Our vector fluid can be directly exported into a vector based program without losing detail.

our vector fluid is wholly described in vector graphics, artists are allowed to print the images with limitless high resolution.

5.2. Shape Designing Tool

With a highly viscous fluid, the velocity rapidly diffuses through the entire space so that the effect of advection is only noticeable just after the user agitates the fluid. This feature provides artists ample time for each interaction, and it offers undo&redo functionality. This effect can be exploited for instant stylish, curly designing. As an example, Figure 17 was created by an artist through a trial and error process with the undo function.

5.3. Target-Driven Design

The underlying fluid can be controlled by external forces. For example, we can combine with a target-

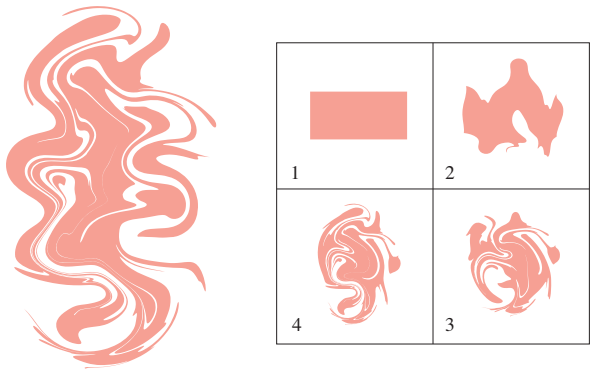


Figure 17: A stylish logo created with viscous fluid. This figure was interactively created by an artist through a trial and error process. The right side of the table shows snapshots of the design process.

Figure 18: Effects of target-driven vector graphics: When combined, our vector fluid can be used to design user-specified stencil art with curly, fluid textures.

driven fluid field [9] to guide regions into a specified target shape. In our implementation, a solid region was rasterized and blurred at each time step. We skipped the smoke gathering procedure because our vector field was not dissipative. Figure 18 shows an example of target-driven smoke animation effects. To generate this figure, we first placed small ovals randomly on the canvas; then we gathered them according to the guiding flow.

5.4. Vector Graphics Editing

Using our method, artists are also allowed to edit existing vector graphics to depict marble-like clip art, as shown in Figure 19. Despite of the complex visual features of beautifully designed vector graphics, the vector graphics are only a collection of concave polygons. Hence, we can trivially achieve this effect by simply extracting each concave polygon embedded in the vector graphics and advecting them under shared fluid motion. In this example, we retrieved path information through Adobe©Illustrator©directly and displaced or subdivided the path with the software API. Hence, the way in which additional information, such as gradient colors in a house or dots on a fish, propagates depends on the implementation of the host program. However,

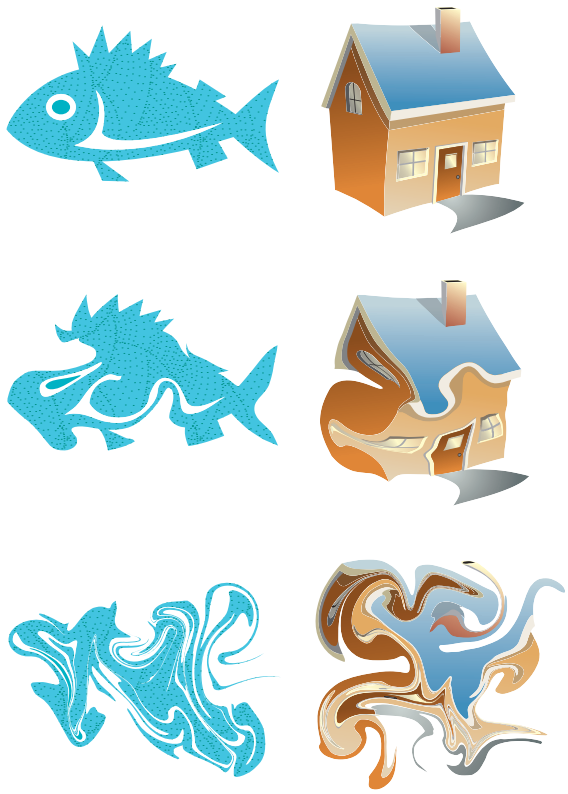


Figure 19: **Swirled vector graphics clips:** Because vector graphics files are a collection of closed paths, we can individually advect those paths under shared motion to deform any kind of vector graphics.

we believe similar visual effects can be easily replicated by filling with patterns or naive gradient fills using stencil masks.

5.5. Flash Animation

Our vector fluid can be efficiently animated on a web browser using Adobe©Flash©(but not interactively). To export it into a Flash movie, each frame was computed and exported as a vector graphic. Those frames were then simplified using a built-in function provided with the commercial software. More specifically, we used the "Simplify Path" function on Adobe©Illustrator©. Finally, the sequential frames were imported into Adobe©Flash©.

The animation was rendered with the built-in Flash engine so that it could be rotated or zoomed dynamically. As can be seen from Figure 20, our vector fluid offers a new medium for web page design. The frame rates in Flash are quite high. We couldn't measure a precise performance, but, in this example, the animation performed stably at 30 FPS on a laptop PC.

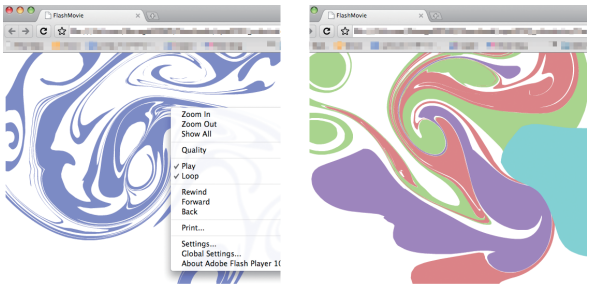


Figure 20: Vector fluid animated in a web browser using the built-in Flash renderer. To animate in Flash, a sequence of vector graphics frames was precomputed and stitched.

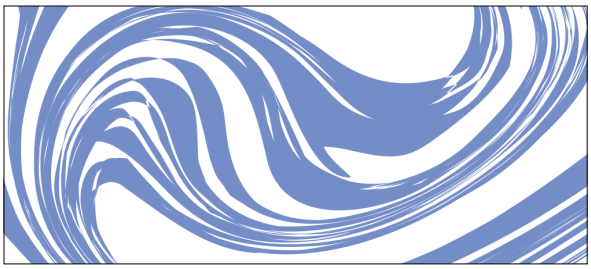


Figure 21: **Result of intersection of contour:** Because this is a numerical method, collisions are inevitable; however, unless the user looks closely at the artifact, the collisions may be tolerated. These collisions can be reduced by embedding equation 7 into equation 3 at the expense of the insertion of extra vertices.

6. Limitations

Despite our "Adaptive Refinement" technique, GPU acceleration and the simplification steps, our method has inherent limitations with respect to the number of vertices and time steps that can be interactively simulated. The underlying fluid flow should be somewhat viscous to keep the contours tractable because small vortices rapidly increase subdivision. Each time step should also be small, otherwise the advection will produce many collisions. Note that these kinds of collisions do not cause the simulation to hang, although they do cause some remarkable artifacts that are apparent when the figures are examined closely (Figure 21).

Moreover, all vector graphics programs have a maximum number of vertices that one vector graphics path object can hold. In our current prototype, we try split regions into two graphics objects that are separated at the thinnest point if we reach this maximum number, and we halt the simulation if it fails. But in practice, we found this kind of limitation can be easily avoided by starting simulation with three or four isolated regions because those regions usually develop evenly complex over time. In such cases, the number of vertices merely reaches the maximum that one graphic object can handle before the simulation becomes slow.

As described in the latter of section 3.6, our simplification test can sometimes fail in specific situations. Most of the failures can be detected by checking whether the connection is circular around the split vertices. In our prototype, we undo such failed simplifications as a remedy.

7. Discussion

We believe that the real-time interaction and the aesthetics of the rendered silhouettes are the most impressive aspect of our method. Traditional approaches to fluid dynamics in computer graphics based on Eulerian grids or Lagrangian particles suffer from numerical diffusion or blobby artifacts when they are applied to the generation of clear surface flow silhouettes. It may be possible to achieve the same goal by employing state-of-the-art front tracking methods, but our method is greatly simplified and specifically tuned to produce more reasonable results. The GPU acceleration and the fast rendering technique presented in this paper are only feasible within two dimensions. The overall idea may be conceptually extendable to three dimensions; however, it is not practically feasible.

8. Conclusion

In this paper, we presented a new method for creating marbling textures using an explicit surface tracking method. The principle idea behind our method is that the contours of the marbling shape rarely collide. To achieve real-time, high-quality marbling rendering, we sampled contours adaptively according to curvature and turbulence. We also ported the base algorithm on a GPU and thus increased the interactive performance to several times faster than that of a CPU. To correct volume errors, which accumulate slowly over time, we presented a fast and accurate volume correction method that maintains the exact volume at any time step. Finally, we removed extremely thin lines according to user preference to massively reduce vertices involving topological changes. As shown in the results and the applications, our method can produce unique marbling textures that have detailed swirly and striped patterns. We believe that our method opens up a new opportunity for vector artists. However, although our prototype is interactive, the running time of our method increases sharply as the contour stretches. In future work, we would like

to modify our method to increase its ability to deal with more complex scenes.

References

[1] Acar, R. (2007). Level set driven flows. *ACM Trans. Graph.*, *26*, 15.

[2] Acar, R., & Boulanger, P. (2006). Digital marbling: A multiscale fluid model. *IEEE Transactions on Visualization and Computer Graphics*, *12*, 600–614.

[3] Ando, R., & Tsuruno, R. (2010). Vector fluid: A vector graphics depiction of surface flow. In *NPAR '10: Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering* (pp. 129–135). New York, NY, USA: ACM.

[4] Bridson, R. (2008). *Fluid Simulation*. A K Peters.

[5] Brochu, T., & Bridson, R. (2009). Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing*, *31*, 2472–2493.

[6] Eden, A. M., Bargteil, A. W., Goktekin, T. G., Eisinger, S. B., & O'Brien, J. F. (2007). A method for cartoon-style rendering of liquid animations. In *GI '07: Proceedings of Graphics Interface 2007* (pp. 51–55). New York, NY, USA: ACM.

[7] Enright, D., Fedkiw, R., Ferziger, J., & Mitchell, I. (2002). A hybrid particle level set method for improved interface capturing. *J. Comput. Phys*, *183*, 83–116.

[8] Enright, D., Losasso, F., & Fedkiw, R. (2005). A fast and accurate semi-lagrangian particle level set method. *Comput. Struct.*, *83*, 479–490.

[9] Fattal, R., & Lischinski, D. (2004). Target-driven smoke animation. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (pp. 441–448). New York, NY, USA: ACM.

[10] Funck, W., Theisel, H., & Seidel, H.-P. (2008). Volume-preserving mesh skinning. In O. Deussen, D. A. Keim, & D. Saupe (Eds.), *VMV* (pp. 409–414). Aka GmbH.

[11] Glimm, J., Grove, J. W., Li, X. L., Shyue, K.-m., Zeng, Y., & Zhang, Q. (1998). Three-dimensional front tracking. *SIAM J. Sci. Comput.*, *19*, 703–727.

[12] Grand, S. L. (2007). *GPU Gems 3, Broad-Phase Collision Detection with CUDA*. Addison Wesley.

[13] Grunebaum, G. (1984). *How to Marbleize Paper: Step-by-Step Instructions for 12 Traditional Patterns (Other Paper Crafts)*. (Reprint ed.). Dover Publications.

[14] Harada, T., Koshizuka, S., & Kawaguchi, Y. (2007). Smoothed particle hydrodynamics on gpus. In *Proc. of Computer Graphics International* (pp. 63–70).

[15] Harlow, F. H., & Welch, E. J. (1965). Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, *8*, 2182–2189.

[16] Hirt, C. W., & Nichols, B. D. (1981). Volume of fluid vof method for the dynamics of free boundaries. *Journal of Computational Physics*, *39*, 201–225.

[17] Kass, M., Witkin, A., & Terzopoulos, D. (1988). Snakes: Active contour models. *International journal of computer vision*, *1*, 321–331.

[18] Maurer-Mathison, D. (1999). *The Ultimate Marbling Handbook: A Guide to Basic and Advanced Techniques for Marbling Paper and Fabric (Watson-Guptill Crafts)*. Watson-Guptill.

[19] McGuire, M., & Fein, A. (2006). Real-time rendering of cartoon smoke and clouds. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering* (pp. 21–26). New York, NY, USA: ACM.

[20] McInerney, T., & Terzopoulos, D. (2000). T-snakes: Topology adaptive snakes. *Medical Image Analysis*, *4*, 73–91.

[21] Mokberi, E., & Faloutsos, P. (1999). A particle level set library. http://www.magix.ucla.edu/software/levelSetLibrary/.

[22] Müller, M. (2009). Fast and robust tracking of fluid surfaces. In *SCA '09: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (pp. 237–245). New York, NY, USA: ACM.

[23] Müller, M. (2009). Fast and robust tracking of fluid surfaces. *SCA '09: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, .

[24] Müller, M., Charypar, D., & Gross, M. (2003). Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (pp. 154–159). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association.

[25] Osher, S., & Sethian, J. A. (1988). Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations. *J. Comput. Phys.*, *79*, 12–49.

[26] Pedersen, H., & Singh, K. (2006). Organic labyrinths and mazes. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering* (pp. 79–86). New York, NY, USA: ACM.

[27] Rohmer, D., Hahmann, S., & Cani, M.-P. (2009). Exact volume preserving skinning with shape control. In *SCA '09: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (pp. 83–92). New York, NY, USA: ACM.

[28] Selle, A., Mohr, A., & Chenney, S. (2004). Cartoon rendering of smoke animations. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering* (pp. 57–60). New York, NY, USA: ACM.

[29] Tryggvason, G., Bunner, B., Esmaeeli, A., Juric, D., Al-Rawahi, N., Tauber, W., Han, J., Nas, S., & Jan, Y. (2001). A front-tracking method for the computations of multiphase flow. *Journal of Computational Physics*, *169*, 708–759.

[30] Witting, P. (1999). Computational fluid dynamics in a traditional animation environment. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (pp. 129–136). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

[31] Wojtan, C., Thürey, N., Gross, M., & Turk, G. (2009). Deforming meshes that split and merge. In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers* (pp. 1–10). New York, NY, USA: ACM.

[32] Woo, M., Neider, J., & Davis, T. (1997). *OpenGL programming guide (2nd ed.): the official guide to learning OpenGL version 1.1.*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[33] Zhao, H., Jin, X., Lu, S., Mao, X., & Shen, J. (2009). Atelierm++: a fast and accurate marbling system. *Multimedia Tools Appl.*, *44*, 187–203.