DIGITAL MARBLING
A GPU APPROACH WITH PRECOMPUTED VELOCITY FIELD

Si Wen *
David R. Cheriton School of Computer Science
University of Waterloo

ABSTRACT

Paper marbling is the art of creating intricate designs on an aqueous surface. We present an interactive digital marbling system by simulating fluid dynamics on the GPU using precomputed solutions to the Navier-Stokes equations. Experimental results have shown that our approach is not only significantly faster, but also sufficiently accurate compared to existing approaches.

Keywords: Fluid Simulation, Navier-Stokes Equations, Semi-Lagrangian Techniques, Graphics Processing Unit

CR Categories:
I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling;

1 INTRODUCTION

Paper marbling is a traditional form of art and printing technique developed in East Asia around the 10th Century. It was initially used to produce patterns similar to smooth marble or other stones, which are applied onto different surfaces, such as book covers and calligraphy prints.

The marbling process involves dropping oil-based paint on an aqueous surface. Artists would then use a rake or stick to manipulate the flow of the paint to create figures. After the design is completed, a sheet of *washi* paper is carefully laid onto the aqueous surface to capture the floating design. Figure-1 shows an example of a finished marbled artwork.

Figure 1: *Example of marbled art (image courtesy of Jane Carr).*

There are a few challenges to traditional marbling. Firstly, it involves a cumbersome setup process. The pigments are difficult to mix and require many ingredients at exact ratios to achieve the desired effects. Secondly, due to the dynamics of water, precise timing and accurate applications are needed to produce desired figures and patterns. Once a mistake is made, the artist has to start over from scratch. Finally, the beauty of marbling lies not only in the final product, but also in the evolution of the art. However, the design process cannot be easily saved.

In recent years, several digital marbling systems have been developed. Most of these systems model the marbling process as a two-dimensional Computational Fluid Dynamics (CFD) problem, and use numerical methods to solve the Navier-Stokes equations. This tends to be computationally expensive and is usually performed on the Graphics Processing Unit (GPU) for real-time interactivity. Nonetheless, even on powerful machines, this cannot be done at a high resolution in a desirable frame rate.

We made an interesting observation that under certain conditions such as high viscosity and low external force, the velocity field of a given fluid exhibits a certain degree of linearity. In applications such as paper marbling, where visual subjectivity is more important than numerical accuracy, we can exploit this property to improve the performance of our marbling system, yet maintain a high degree of similarity to pure physical approaches. In this paper, we present a technique where velocity field is generated using transformations of precomputed results. We will also describe the GPU implementation of our system using vertex and fragment shaders.

2 PREVIOUS WORK

Although extensive research has been done on both fluid dynamics and artistic simulations, the history of marbling simulation is relatively short. The idea of modeling the marbling system as a two-dimentional CFD problem was first suggested by Suzuki et al [13]. Later, Mao et al. [10] extended the system into an interactive application named *AtelierM*. Akgun [2] first described a digital tool suite for creating traditional Turkish art forms, based on numerical solutions to the Navier-Stokes equations. Acar and Boulanger [1] introduced a multiscale fluid model to simulate the turbulent flows in traditional marbling techniques.

*swen@uwaterloo.ca

Although these systems can model increasingly complex flows, they do not provide real-time feedback as solving the physical equations can be extremely time-consuming on the CPU. Jin et al. [6] presented a different approach which solves the Navier-Stokes equation on the GPU using the multigrid method [4]. However, like other finite grid-based fluid solvers, it suffers from blurry paint outlines due to numerical diffusion and dissipation. Xu et al. [14] managed to alleviate the numerical problems with higher-order interpolation schemes such as the Back-and-Forth Error Compensation and Correction (BFECC) algorithm and the MacCormack algorithm. In *AtelierM++*, Zhao et al. [15] employed the fast and accurate third-order Unsplit Semi-Lagragian Constrained Interpolation Profile (USCIP) method to further reduce numerical dissipation, implemented on NVIDIA's CUDA GPUs.

Apart from the grid-based CFD approaches, there are also a few different ways to model the marbling system. Ando and Tsuruno [3] presented an efficient framework that generates vector graphics quality marbled textures based on an explicit surface tracking method. This solves the problem of the blurry paint interface. However, because the underlying fluid is still modeled using physical equations, it is time-consuming and cannot achieve a satisfying frame rate.

Lu et al. [9] on the other hand discovered an entirely different approach that models fluid flows using closed-form mathematical expressions. It produces high-quality marbled art with clearly defined contours. Because it is based on closed-form equations, it is significantly faster than the physics-based approach, thus allowing high-quality artwork as well as vector graphics output. However, only a limited set of functions are defined, such as the paint drop function, the tine-line pattern function, and the wavy pattern function. Although the system runs in real-time with user interactions, it does not allow natural free-hand operations that marbling artists are most accustomed to.

3 METHOD

This paper follows the two-dimensional grid-based CFD approach. We present a significantly faster way to generate real-time velocity field. We were able to implement our system on Apple's iPad (4th generation) in full resolution at a high frame rate, thus allowing marbling artists to interact our system using natural touch gestures.

Our approach is based on two observations on viscous fluids. First, under certain conditions such as high viscosity, any external force dissipates quickly. Its area of diffusion is bounded by a relatively small region of the whole grid. Thus, it is unnecessary to perform calculations on the entire grid, as at any time most of the resulting velocities will be either a zero vector or so negligible that we can treat it as such. Second, given high viscosity and low external forces, the velocity field exhibits a certain degree of linearity. In other words, the velocity of a series of external forces is similar to the combination of the velocity of each individual force.

In real-life marbling, both the paint and the medium on which the paint floats are both viscous. In addition, the stick that the artist uses to manipulate the paint has a thin pointy tip, which allows precise control of the paint and its surrounding fluid. This means any external force exerted by the stick is relatively small. Since both conditions are satisfied, we can exploit the aforementioned property to devise a fast and stable way to generate velocity fields in real-time.

Our approach can be broken down into an initialization step and three repeating steps.

- Step 0: precompute the velocity field of a "unit force".
- Step 1: compose a new velocity field based on user inputs.
- Step 2: advect density using the composed velocity field.
- Step 3: render onto the screen and an off-screen buffer.

Our algorithm is based on this heuristic without any formal proof. Nonetheless, we will later compare results of traditional physics-based approach and our approach to demonstrate its correctness in both simple and complex scenarios.

3.1 Precomputed Navier-Stokes Equations

We first precompute the velocity field of a "unit force". In this case, we set the "unit force" to be a single unit of external force applied onto the center of an empty velocity field in a pre-specified direction (in our case, in the positive-y direction). To precompute the velocity field, we use numerical methods to solve the Navier-Stokes equations (1-2) following Stam's semi-Lagrangian techniques [11].

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{F} \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

For equation (1), $\frac{\partial \mathbf{u}}{\partial t}$ represents the velocity of a fluid at any point in time and is determined by the four acceleration factors given by terms on the right. These terms respectively represent the advection, pressure, diffusion, and external force of the fluid, with ρ being the fluid density and ν being the kinematic viscosity. Equation (2) enforces the conservation of volume, as we assume our fluid to be incompressible.

Since we are to precompute the velocity field, performance is not a concern. We instead focus on accuracy. Thus, we use small timesteps and employ the fourth-order Runge-Kutta scheme. To solve the Laplacian operator ∇^2, we setup a consistent system of linear equations and solve for its exact solution. This minimizes errors that can accumulate in the final velocity field.

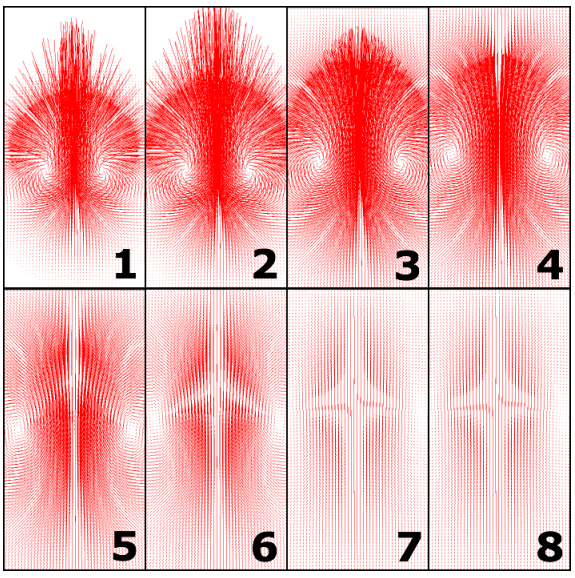


Figure 2: *The precomputed velocity fields of a "unit force". Shown in a 512×512 texture of 8 frames of velocity fields, each being 128×256, with ν = 0.03, Δt = 0.1.*

Because $\nu > 0$, the velocity of a "unit force" diffuses quickly and dissipates completely after a given amount of time. Thus, the effect of an external force can be described completely by a finite number of frames. When ν is big, the number of frames needed will be small. Figure-2 depicts the frame-by-frame "evolution" of a unit force, whose velocity dissipates completely after 8-10 frames.

3.2 Composition of Velocity Field

We allow users to interact with the fluid by "drawing" on a touch screen. We treat these inputs as injection of external forces onto the existing velocity field (which is initially empty). As a user interacts with our touch application, we track his/her finger movement across the screen. We first interpolate the touch inputs and discretize the resulting spline into uniform segments. This avoids problems when the user swipes his/her finger too fast or too slow, generating too many or too few touch points, resulting in unrealistic simulations. This interpolation/discretization process can be seen in Figure-3.

Figure 3: *A sample of touch inputs, represented by red dots. The red curve is an interpolated spline that connects these inputs. The black dots are the re-sampled inputs, which is spaced more uniformly. The black segments are the discretized velocity vectors.*

We then apply the precomputed velocity field to these segments. Since the segments are not identical to the "unit force" used to precompute the velocity field, we scale and rotate the precompute velocity field accordingly. In addition, since there is a chronological ordering to the input segments, we apply the precomputed velocity fields accordingly. For example, suppose the input in Figure-3 is drawn from left to right. The right-most segment is the newest input, thus its corresponding precomputed velocity field is the first frame (see in Figure-2). On the other hand, the left-most segment is drawn 4 frames before the right-most segment, thus its velocity field has already evolved for 4 frames (or Δt = 0.4). Therefore, we need to apply the 5th frame of the precomputed velocity field. Figure-4 shows how the precomputed velocity fields are being applied onto each segment.[1] The resulting composition is a newly generated velocity field. It represents the total impact of external forces on a given fluid.

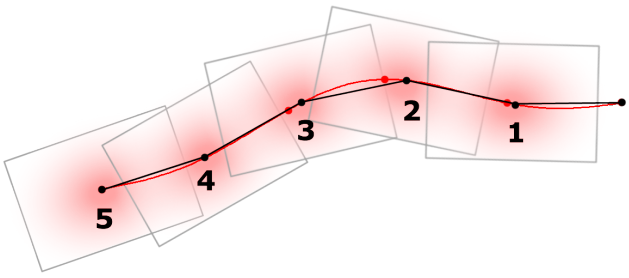


Figure 4: *The transformation and composition of velocity fields.*

Unlike the traditional physics-based approach, in which each successive velocity field depends on the previous one (or the previous few when using higher-order schemes), our approach generates the velocity field from scratch, only depending on current inputs. This means we need to store inputs (external forces) from previous frames, which is cheap in terms of memory. It also means the effect of an external force can last for as many frames as we precompute. This does not affect the quality of our simulation because in the traditional approach, an external force will dissipate over time and its effect deteriorate. Since we precompute and store the entire "evolution" of the "unit force", we do not lose any information.

[1] In reality, the precomputed velocity field is usually much bigger than depicted in Figure-4, and successive frames overlap much more.

3.3 Density Advection

After generating the velocity field, we use it to advect the paint. Since we are using the grid-based approach, each cell in the grid represents an amount of paint, or its density. Thus, density advection involves "transporting" the density from one cell to another based on the velocity of that cell.

We use the semi-Lagrangian approach for density advection [12]. Instead of advecting the density of a cell forward in time using its velocity, we trace backward following its velocity and advect the density of the destination cell. In case the destination is not at the center of a cell, we interpolate the density of nearby cells to give a better estimate. This is usually done using bilinear interpolation, which is prone to numerical dissipation and results in the edge of marbled shapes being blurry.

To reduce numerical dissipation and thus sharpen the contour of marbled shapes, we employ the Unsplit Semi-Lagragian Constrained Interpolation Profile (USCIP) method, proposed by Kim et al. [8] and used in *AtelierM++* [15]. USCIP is a third-order interpolation scheme with its polynomial defined as follows:

$$\Phi(x,y) = \sum_{0 \le i,j \le 3} C_{ij} x^i y^j + C_{31} x^3 y + C_{13} x y^3 \qquad (3)$$

The coefficients of this polynomial are uniquely given by

$$\begin{aligned}
C_{00} &= \phi_{00} \\
C_{10} &= \phi_{x00} \\
C_{01} &= \phi_{y00} \\
C_{20} &= 3(\phi_{10} - \phi_{00}) - \phi_{x10} - 2\phi_{x00} \\
C_{02} &= 3(\phi_{01} - \phi_{00}) - \phi_{y01} - 2\phi_{y00} \\
C_{30} &= -2(\phi_{10} - \phi_{00}) + \phi_{x10} + \phi_{x00} \\
C_{03} &= -2(\phi_{01} - \phi_{00}) + \phi_{y01} + \phi_{y00} \\
C_{21} &= 3\phi_{11} - 2\phi_{x01} - \phi_{x11} - 3(C_{00} + C_{01} + C_{02} + C_{03}) - C_{20} \\
C_{31} &= -2\phi_{11} + \phi_{x01} + \phi_{x11} + 2(C_{00} + C_{01} + C_{02} + C_{03}) - C_{30} \\
C_{12} &= 3\phi_{11} - 2\phi_{y10} - \phi_{y11} - 3(C_{00} + C_{10} + C_{20} + C_{30}) - C_{02} \\
C_{13} &= -2\phi_{11} + \phi_{y10} + \phi_{y11} + 2(C_{00} + C_{10} + C_{20} + C_{30}) - C_{03} \\
C_{11} &= \phi_{x01} - C_{13} - C_{12} - C_{10}
\end{aligned} \qquad (4)$$

Here, $\phi_{00}, \phi_{01}, \phi_{10}, \phi_{11}$ are the densities at each cell corner; $\phi_x = \frac{\partial \phi}{\partial x}$, $\phi_y = \frac{\partial \phi}{\partial y}$; and $\phi_{x00}, \phi_{x10}, \phi_{x01}, \phi_{x11}, \phi_{y00}, \phi_{y10}, \phi_{y01}, \phi_{y11}$ are the derivatives at each cell corner.

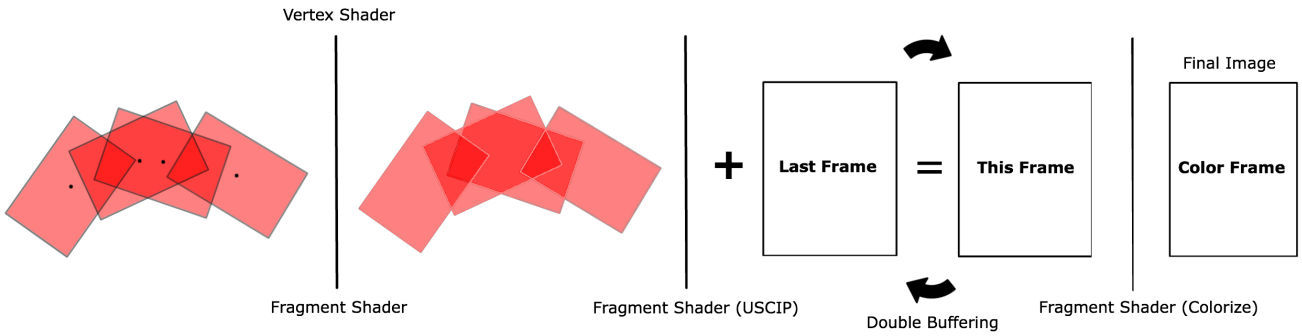


Figure 5: *An illustration of the graphics pipeline.*

3.4 GPU Implementation

Our target platform (Apple's iPad 4) has a quad-core GPU which supports OpenGL ES 2.0 with GLSL 1.20. It is ideal to implement our method on the GPU using GLSL.

The composition of precomputed velocity field is essentially multi-texturing. The precomputed velocity field can be uploaded onto the GPU as textures. Then, for each frame, we perform Steps 1-3 as outlined at the beginning this section.

Step 1. We compose the precomputed velocity field according to the input forces. For each input segment, we render a GL_QUAD with the corresponding rotation and scale. We then map its corresponding frame of the precomputed velocity field onto the GL_QUAD. In the fragment shader, we rotate the velocity vectors (since rotating the GL_QUAD does not rotate the velocity vectors in the texture) and sum them up.

Step 2. We sample both the newly generated velocity field and the density field of the previous frame. We redraw the density field with a GL_QUAD the size of the screen. In the fragment shader, we advect the densities using the forementioned USCIP scheme.

Step 3. Finally, we colorize advected densities as a texture and render it onto the screen.

We use different shaders and Frame Buffer Objects (FBOs) in each step. Most of the parameters (such as the input segment's angle of rotation and its corresponding frame) can be uploaded as uniform variable, thus avoids unnecessary computations. This pipeline can best be visualized in Figure-5. On other platforms that support Multi-Target Rendering, Steps 2 & 3 can be combined into a single step.

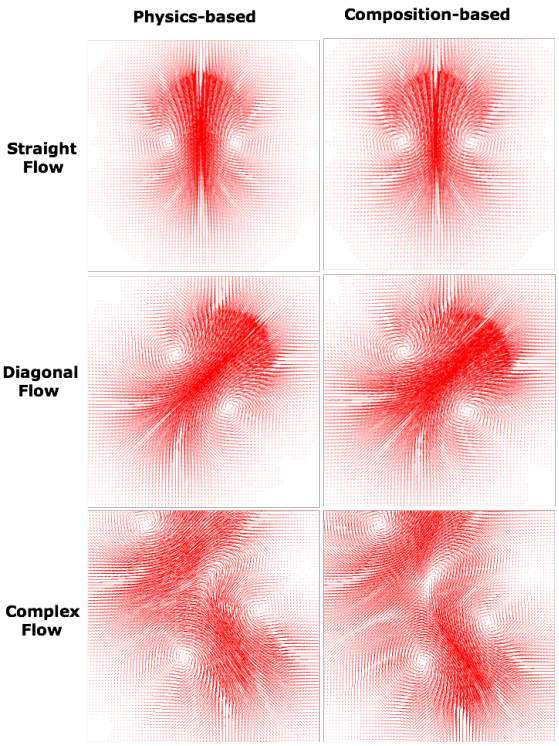


Figure 6: *Comparison of velocity fields generated using different methods based on the same settings and same inputs.*

4 Results

Grid Size	Stam's	Ours (CPU)	Ours (GPU)
256×192	37ms	18ms	3ms
512×384	186ms	70ms	8ms
1024×768	981ms	270ms	20ms

Table 1: *Time took to render a single frame on the iPad 4.*

On Apple's iPad 4 [2], we compared our approach with an implementation of Stam's approach. As depicted Figure-6, the velocity field generated using our approach has a high degree of resemblance to that of Stam's approach, even for complex turbulent flows. This indicates that our result is a good approximation to the physics-based approach, and validates our hypothesis about fluid's behaviour under prescribed conditions.

On the other hand, our approach is much more efficient than Stam's approach, which can be shown in Table-1.

5 Discussion

5.1 Limitations

An obvious limitation is that our approach depends on the assumption that the viscosity of the fluid is high and the external force is relatively low. Therefore, our approach is not applicable for other

[2] CPU: A6X 1.4 GHz (dual-core); GPU: PowerVR SGX554MP4 (quad-core) @ 280MHz; RAM: Quad-channel 533 MHz LPDDR2-1066

simulations such as smoke and fire, where viscosity is close to zero and external force can be large. However, we have used our system to model other forms of simulation such as the interaction of coffee and milk, mimicking the result of *latte art* (Figure-7).

Another limitation is the quality of the marbled art. Although USCIP reduces the effect of numerical dissipation, it does not eliminate it completely. Thus, after extensive manipulation on the paint, the density will eventually get dampened and the marbled art will appear to be blurry, especially at the edges.

Also, so far our system works for a single color only, which is used to colorize the density in the last step of rendering. To support multiple colors, one approach is to use multiple layers of density field, as proposed by Jin et al. [6]. However, this essentially slows down the advection step by a factor of k, where k is the number of layers supported.

In the following section, we propose a way to solve the above two problems.

Figure 7: *A piece of artwork produced using our application. With a colored filter and some random noise, it resembles latte art.*

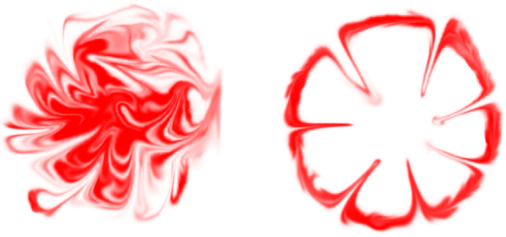
5.2 Future Improvements

Ando and Tsuruno [3] introduced an explicit surface tracking method used to generate vector graphics output with sharp contours and vibrant colors. Instead of representing the marbling paint using a density grid, this method uses points to track the contour of the paint. This frees the marbled art from a fixed resolution determined by the grid size. Nonetheless, in Ando's method, the underlying fluid is modelled using the traditional grid-based physics approach. We can combine our velocity composition mechanism with the front-tracking method in order to achieve vector graphics quality output, which solves the problem of blurry edges and immiscible paint.

We will also develop a tool suite which would allow users to have access to tools in that of real marbling. This includes the ability to inject new paint, disperse paint to create ring-shaped figures, save and replay their works from scratch. In addition, since velocity composition is independent of results of other frames, it can be easily inverted. This gives us the ability to "rewind", allowing users to go back in time and fix their mistakes. This is impossible to perform on physics-based approaches.

6 Conclusion

Based on the observation that under certain conditions such as high viscosity and low external force, the velocity field of a fluid exhibits a certain degree of linearity, we devised an algorithm for generating real-time velocity field by combining precomputed solutions to the Navier-Stokes equations. Due to its parallel nature, it is done on the GPU and is implemented as a stage in our marbling system's pipeline. Experiments have shown that it is significantly faster than existing approaches and is capable of producing marbled artworks that resembles the ones in real life.



(a) *A pattern* **(b)** *A figure*

Figure 8: *Marbled artworks produced by our system.*

References

[1] R. Acar and P. Boulanger. "Digital Marbling: A Multiscale Fluid Model," in *IEEE Transactions on Visualization and Computer Graphics*. Piscataway, NJ: IEEE Educational Activities Department, vol. 12, no. 4, pp. 600-614, Jul 2006.

[2] T. Akgun, "The Digital Art of Marbled Paper," in *Leonardo*. Cambridge, MA: The MIT Press, vol. 37, no. 1, pp. 49-51, 2004.

[3] R. Ando and R. Tsuruno, "Vector Graphics Depicting Marbling Flow," in *Computers & Graphics*. Elsevier, vol. 35, no. 1, pp. 148-159, Nov. 2011.

[4] W. Briggs and S. McCormick. *A Multigrid Tutorial*. Philadelphia, PA: SIAM, vol. 72, 2000.

[5] M. J. Harris, "Fast Fluid Dynamics Simulation on the GPU," in *GPU Gems Volume One*. Indianapolis, IN: Pearson Education, 2004 ch. 38, pp. 637-665.

[6] X. Jin *et al.*, "Computer-Generated Marbling Textures: A GPU-Based Design System," in *Computer Graphics and Applications*. New York, NY: IEEE, vol. 27, no. 2, pp. 78-84, 2007.

[7] D. Khalid and R. Egli. "Precomputed 3D Velocity Field for Simulating Fluid Dynamics," in *Game Engine Gems, Volume One*. Sudbury, MA: Jones and Bartlett Publishers, ch. 12, pp. 177-184. 2010.

[8] D. Kim *et al.*, "A Semi-Lagrangian CIP Fluid Solver without Dimensional Splitting," in *Computer Graphics Forum*. Blackwell Publishing Ltd, vol. 27, no. 2, pp. 467-475, Apr. 2008.

[9] S. Lu *et al.*, "Mathematical Marbling," in *Computer Graphics and Applications*. New York, NY: IEEE, vol. 32, no. 6, pp. 26-35, Nov. 2012.

[10] X. Mao *et al.*, "AtelierM: a physically based interactive system for creating traditional marbling textures," in *Proc. 1st international conference on computer graphics and interactive*

techniques in Australasia and South East Asia. New York, NY: ACM Press, pp. 79-86, Mar. 2003.

[11] J. Stam, "Stable Fluids," in *Proc. SIGGRAPH, 26th annual conference on Computer Graphics and Interactive Techniques*. New York, NY: ACM Press, pp. 121-128, 1999.

[12] J. Stam, "Real-Time Fluid Dynamics for Games," in *Proc. 2003 Game Developer Conference*. Game Developer Conference 2003, vol. 18, Mar. 2003.

[13] T. Suzuki *et al.*, "Simulating Marbling with Computer Graphics, ," in *Proc. IASTED International Conference on Visualization, Imaging, and Imaging Processing*. Calgary, AB, Canada: IASTED, Sep. 2001.

[14] J. Xu, *et al.*, "Nondissipative Marbling," in *Computer Graphics and Applications*. New York, NY: IEEE, vol. 28, no. 2, pp. 35-43, 2008.

[15] H. Zhao *et al.*, "AtelierM++: a fast and accurate marbling system," in *Multimedia Tools and Applications*. New York, NY: Springer, vol. 44, no. 2, pp. 187-203, Sep. 2009.