Project Report: Implementation of a Procedural Watercolor Engine

Khan Mostafa
Graduate Student, Computer Science, Stony Brook University, NY 11794, USA
Email: khan.mostafa@stonybrook.edu
Student ID# 109365509

ABSTRACT

This is the report of *Computer Graphics course project*. The goal of this project is to re-implement a procedural watercolor engine, developed recently by DiVerdi et al [1]. This approach uses vector based empirical simulation for watercolor, rather than traditional raster based approaches. By exploiting sparse nature of watercolor, this procedural engine can recreate watercolor effects in real time as it opts to calculate only that fraction of picture where paintings are live. In this report, a brief review of prior work is given along with short description of the reference work. The implementation and outcomes are also noted.

1 INTRODUCTION

With the advancement of computing devices, digital paint media has emerged with suites of tools that artists have never experience before. Tools include palettes of multitude of colors, ability to undo, erase, mechanism for drawing precise geometric shapes, curves and so on. However, these digital paint features lack a lot of effects and desirable artistic nature of real paint mediums like watercolor, oil paint, acrylic, pencil sketch etc. All of these real media has their own characteristic natures derived from physical natures of the components used in them. Artists have long been exploited these features and they crave for such features even in digital media. Attempts are made from the dawn of digital paint applications to recreate realistic painting experience and yet researches are going on to perfect them. Earliest approaches could hardly work in real-time. Nevertheless, computation power has increased since the first attempts. As well as, researches have unveiled less computationally intensive approaches for approximating realist phenomena.

Section 2 briefly discusses some approaches that recreates watercolor. Later a work by DiVerdi et al is presented with details of core parts of it. It describes a procedural watercolor engine which models watercolor paints as live polygons (splats) and computes pigment advection as random walk. This project re-implements the core part of the presented work. Original paper discussed a commercial implementation of their algorithm, however very sophisticated application with nifty user interfaces is beyond scope of this project. Implementation is documented in section 5 and section 6 shows some results.

2 PROPERTIES OF WATERCOLOR AND OTHER REAL MEDIA PAINTING

To mimic watercolor in computer graphics, we need to understand physical and characteristic nature of watercolor and artistic affects created by it. Curtis *et al* [2] and some other authors discussed them in detail.

2.1 Watercolor materials

Water color paintings are created by applying watercolor paints, often simply referred to as watercolors, on special type of absorbent papers.

Watercolor papers are typically made from cotton or linen, pounded into small fibers. This produces microscopic web of tangled fibers, trapping air pockets. This makes such papers extremely absorbent. This causes immediate soaking and diffusion of water and paint pigments. To reduce this and create more desirable artistic experience, sizing (generally made from cellulose) is applied. Sizing create barriers and reduce rate of soaking. Sizing is placed sparingly, so there remains pores too. Watercolor paper surfaces are desirably rough.

Watercolor paints consists pigments of different color. Pigments are grounded into grains of about 0.05 to .5 microns with variant weights. They vary in density. Pigments are mixed with binder (glue) and surfactants (solvents). Binder allows pigments to be absorbed by the paper and surfactants let them flow. So, watercolor paints have a viscoelastic nature. Elasticity vary to meet artists' choice. Watercolors are generally mixed with water before soaking brush into it.

Watercolor pigments vary in density. Heavier ones tend to get soaked in paper while lighter ones remain suspended in water for a longer while. Suspended pigment flow with water and create several artistic effects. Pigments, once adhered to paper fibers tend to stay.

Artists generally put several glazes of paints on watercolor papers. A final appearance of watercolor painting incorporate several effects as described in following section.

2.2 Watercolor Effects

Following are list of watercolor effects, commonly observed and mentioned by several authors [2] [3] [1] [4]:

- *Dry brush effect:* when almost dry brush is applied on a paper, it stains only raised areas of paper. [2]
- *Edger Darkening:* Pigments flow after being applied to paper. In wet on dry situation, pigment cannot propagate into dry regions of paper. When pigment flow within wet areas of applied region edge between wet and dry portions of paper accumulate more pigments and seem darker in color. [2] [1]
- *Backruns:* When some water is applied on already stained but damp area, pigments accumulate motion from applied watercolor and thus flow. This create darkened edges with lightened stains – a specific effect called backrun. [2] [1]
- *Granulation:* Pigments have different size and weight. Heavier pigments settle much earlier than lighter ones. Roughness of paper cause uneven settling of pigments in peaks and valleys of the surface. [2] [1]
- *Flow pattern:* In wet on wet painting, wet paper let pigment flow freely and create soft feathery shape. [2]
- *Glazing:* Artists put several glazes of water color. Once some color is settled and dried, artists put another layer of brush strokes. This strokes do not perturb already dried stains. So, transparent layers of washes or glazes become visible. This overlaid glazes display dynamic, luminous, bright colors. [2]
- *Rewetting:* Sometimes, when strokes are not still damp, application of strokes over it rewets some part of already stained area. This creates some features. [1]
- *Color blending:* Several glazes, overlaid one upon another create transparent color blending. [1]
- *Feathering:* Free flow of pigments create feathered look. This feature identified by DiVerdi *et al* is similar to Curtis *et al*'s flow pattern. [1]

Lei and Chang [4] identifies edge darkening as most important effect. They also simulate glazing, backrun, granulation in their approach.

Van Laerhoven and Liesenborgs [3] identify similar effects and recreate dark edges, overlapping strokes etc.

MoXi, which is a simulation of Chinese ink consider similar effects as of watercolor. Chinese ink, is different from watercolor. However, they display many common effects of watercolor. Chu and Tai [5] discussed on feathery pattern, light fringes (like glazing), branching patterns, boundary roughening, and boundary darkening (like edge darkening) as major features.

Figure 1 show these effects as described by [1].

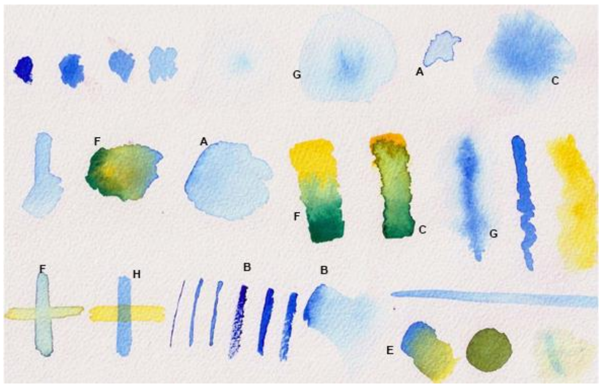

Figure 1. Features identified by DiVerdi et al: edge darkening (A),nonuniform pigment density (B), granulation (C), rewetting (D), back runs (E), color blending (F), feathering (G), and glazing (H).

3 Prior Work

Digital paint systems lack many features of real media like watercolor. Many effects of watercolor are much appreciated by both artists and viewers and simulating such effects in computer paint systems are very desirable.

There have been ample undertakings to recreate watercolor like effects in images. Some tried to convert real photo images to give watercolor like effects [6]. Yet some approaches are to render 2-D and 3-D models non-photorealistically to give essences of watercolor [6] [4].

Early approaches to simulate watercolor painting took physics inspired approach [7]. Simulating all physics based effects are very computation heavy and cannot recreate all desirable effects. Rather some empirical approaches are studied, to recreate effects without necessarily computing all physical characteristics [2] [3] [1] [5] [8]. Most of these approaches are paper based models, simulating fluid propagation and pigment propagation, diffusion etc. Until recent, approaches are taken to simulate water color using cellular automaton [7] [2] [3] computes per pixel. More recent approaches take vector based approaches [1] by computing stamps and splats rather than per pixel, allowing rendering in high resolution. Model based approaches often incorporate cellular automaton too. For fluid simulating several approaches are studied including: Foster model [2], Navier-Stokes equation [3] and Lattice Boltzmann Equation [5].

In a more recent work, DiVerdi *et al* [1] (2012-2013) proposed a vector based model. They have also deviated from using physics based simulation further by using rather more empirical approach which is inspired by actual water color characteristics. They proposed **Painting with Polygons** for a **Procedural Watercolor Engine**. Which I have re-implemented in this project.

4 Description of Base Work

A major drawback of earlier works were that, they were computationally very heavy. This came from two reasons – (1) raster based simulation of whole canvas (2) attempts to exactly replicate watercolor. DiVerdi *et al* took several intuitions,

- Watercolor strokes effects generally sparse
- Specific path taken by pigment can be much realistically replicated by random walk which creates similar result as achieved by more complex method

4.1 Algorithm

The algorithm adopts a sparse representation to model watercolor strokes and uses random walk to update position in each time step. Paint pigments are represented as "splat" particles – each being a complex polygon of n vertices. The algorithm has four major operations: - Paint Initialization, Pigment Advection, Sampling Management, and Lifetime Management.

4.1.1 Paint Initialization

At first, stroke input is recorded by placing stamps in uniform length increment along the stroke path. Stamps are set of splats which store age, flow and several other attributes. At each stamp, water are also placed on canvas and stored separately as rasterized 2D grid of cells called water-map.

4.1.2 Pigment Advection

Splats are advected at each time step embodying biased random walk behavior. Vertex's own velocity, splat's bias velocity, water velocity, paint viscosity, roughness of canvas media and gravity are modeled at this point. These helps to achieve branching and roughening aspects of watercolor.

Following equations used,

$$\mathbf{d} = (1-\alpha)\mathbf{b} + \alpha \frac{1}{\mathbf{U}(1, 1+r)} \mathbf{v}$$

$$\mathbf{x}^* = \mathbf{x_t} + f\mathbf{d} + \mathbf{g} + \mathbf{U}(-r, r)$$

$$\mathbf{x_{t+1}} = \begin{cases} \mathbf{x}^* & \text{if } w(\mathbf{x}^*) > 0 \\ \mathbf{x_t} & \text{otherwise,} \end{cases}$$

Here, d is displacement calculated. A tuning parameter α is used with a value =0.33. This tunes the effect of global force fields and local force fields. Forces acting on a splat is shown in Figure 2.

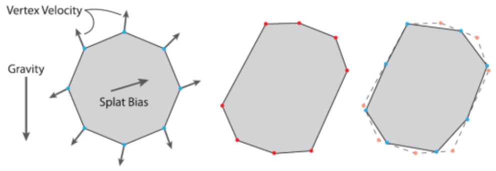

Figure 2. Splat as in DiVerdi et al. (left) forces acting on a splat, (center) advected splat (right) after sampling management

4.1.3 Sampling Management

Over time, different advection directions may create unrealistically straight hard edges due to local undersampling. This artifact is dealt by the third operation, by resampling the splat. Two approaches can be taken,

- Constrained vertex motion
    - Vertex cannot move too far from its neighbor vertices of same splat
- Periodic resampling of splat boundary
    - Compute total perimeter
    - Arc length per vertex is computed
    - Vertices moved to maintain uniform arc length

Boundary resampling is shown in Figure 2

4.1.4 Lifetime management.

Here, a splat has three stages of life: flowing, fixed and dried. Initially a splat is flowing; after certain steps (age) they are fixed – but can potentially be rewetted to resume advection. After a certain period the splat is dried and then rasterized into dry pigment buffer and removed from the simulation. While rasterizing, a paper texture (also mimicking granulation) is applied. Rewetting helps to achieve effects like back runs and feathered edges. Lifetime management also mimics water flow, granulation and other watercolor affects.

4.1.5 Brush Types

The paper represents a way to implement brush types to reproduce a variety of watercolor characteristics. It demonstrates five brush types.

To summarize, all four operations of the algorithm and brush types achieves common watercolor effects, such as, color blending, feathering, edge darkening, non-uniform pigment density, granulation, back runs, rewetting, glazing etc.

Brush types can be achieved by varying several parameters as described below.

- Different arrangement of splat per stamp
- Different brush parameter settings
  - Target width, w
  - Initial wet at wet map (=255)
  - Splat life (l= initial a)
  - Roughness, r
  - Flow, f

Brush types examples:

Simple
- Single splat, d=w, b = <0,0>

Wet on dry
- 7 splats, central splat bias = <0,0>, d=w/2
- Perimeter splat bias = $\langle \frac{d}{2}\cos\theta, \frac{d}{2}\sin\theta \rangle$

Wet on wet
- Small splat (d=w/2) inside large splat (3w/2)
- r=5, l=15, b=<0,0>

Blobby
- Randomly sized 4 splats, l=15, b=<0,0>

Crunchy
- 1 splat, r=5, f=25, l=15

Figure 3 shows these brush types.

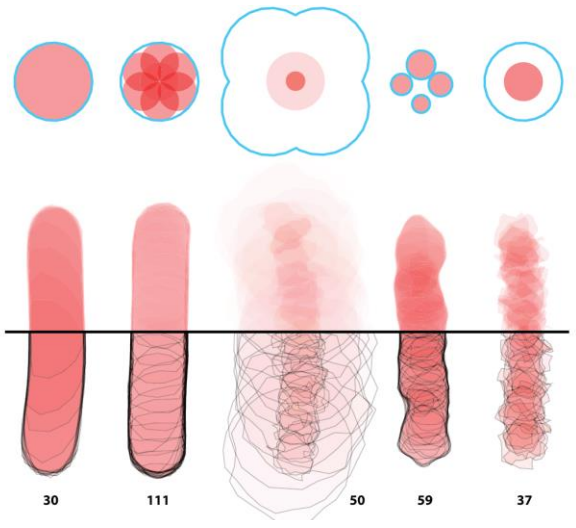

Figure 3. Brush types exemplified in DiVerdi et al (left to right: simple, wet on dry, wet on wet, blobby, crunchy).

4.2 Author’s Implementation

The authors have also implemented a commercial application which performs well in real time in low end devices. This method can also recreate most water color effects without requiring very high computation power.

For rendering purposes, they suggested using OpenGL facilities. They suggested using two pass stencil buffer per splat. For anti-aliasing, they suggested using post processing after full rendering to reduce computation cost.

This approach, also have some limitations. Splats are used for live strokes. This vector representation might use up a lot of memory and computation time, when drawing start to use a lot of live strokes. Although, to limit this, they have rasterized dried splats to avoid re-computing stable splats each time step.

5 PROJECT IMPLEMENTATION

To implement the project, I have used OpenGL tools, including the original GL library for graphics functionalities, Nate Robin’s GLUT (GL Utility Toolkit) for basic user interface and SOIL (Simple OpenGL Image Library) for storing created image. This section describes the implementation.

Appendix A shows main classes used in the implementations.

5.1 Paint Initialization

User may want to paint using mouse or touchscreen. I have used GLUT's mouse motion capture functionality, which periodically sample mouse position (x, y). This required to register a listener using `glutMotionFunc` which plugs a callback function to mouse motion events. I have registered `regStrokePoint` function that stores listened (x, y) position in a vector or Stroke objects. Stroke object are modeled with a Class `Stroke` which stores the listened co-ordinates (x, y), current brush type and size (in pixels), and current color.

Drawing a stroke completes as the mouse button is released. Another GLUT event listener is registered using `glutMouseFunc` which triggers when a change of mouse button is occurred. I have acted to event: `(button == GLUT_LEFT_BUTTON && state == GLUT_UP)`. Upon this event, the application records a full stroke input and invokes a function to convert strokes input points into stamps. Stamps are uniformly placed in stroke trajectory in a distance of roughly one thirteenth of actual brush distance. Stamps are modeled as a queue of object of `Stamp` class. This class stores, location co-ordinates (x, y), (x, y), current brush type and size (in pixels), and current color which are copied from records of corresponding stroke pint input record.

Upon computing queue of stamps, they are immediately processed and splats are enlisted to simulation vector. Each stamp is converted into one or more splats depending on brush types. Splats are stored in a vector as objects of `Splat` type which stores, co-ordinates of vertices of the polygon (a splat is represented as a polygon of N vertices, for this implementation N=8). Additionally, age, flow, roughness, an opacity value (based on brush type) and color (from stamp) is stored for each splat. Vertexes of polygon is calculated from brush size (pixels) and the stamp location, using the stamp location as center. Per vertex velocity is similar for all splats. This velocity is assigned outward according to initial placement of the vertex in the splat.

During computation of strokes, wet-map (a Width*Height matrix) is also assigned with wet values, depending on brush types.

5.2 Pigment Advection

Pigment advection is modeled in the Splat class using a function `advect()` which takes the wet-map as input. This function models the set of equations presented in 4.1.2. As the gravity is actually omitted considering that, paper is placed parallel to earth surface, effectively having x or y component of g nullified, this function simply puts zero in place of g.

In my model, the scene model is mapped in floating point values rather than integer pixel values, with a ratio using parameter (`RATIO`). For this case, `RATIO = 100;`

Hence, while computing the advection, parameter that are in pixels are properly mapped to scene model. To get the wet map value, indices are also needed to be computed.

At each time step advection is done, age is decremented.

Advection is done after call of `drawscene` function of OpenGL by iterating over the splat vector and invoking advect function for each iterated splat. In fast machine, calling advection after each damascene call is unnecessary computing overload and too fast to be realistic. That is why, the frequency of advection is capped to roughly at 30Hz.

When advection for vector of splats is invoked, `UpdateWetMap` is also invoked to reduce wetness of each pixel by one.

5.3 Sampling Management

I have constrained motion of vertices to limit within a realistic threshold to preserve splat shapes as described in 4.1.3. More computation heavy boundary re-computation approach is not used as the tradeoff of achieving higher quality simulation to computation cost is not impressive.

5.4 Lifetime Management

DiVerdi et al suggested using three life types of a splat: flowing, fixed and dried. My implementation does this in two part: - flowing and fixed is handled in `advect ()` function and drying is done using a `dryOut` function.

`dryOut` is invoked infrequently to go over the splat vector and check for age. If the splat is fixed for over 2400 time steps, it is removed from the simulation. To preserve the simulated effect of the splat itself, `dryOut` function store the simulated stains to canvas. Canvas is drawn in the scene and invoked in `drawscene` function.

In advect function, advection is omitted if the age negative. Negative ages are representative of fixed stage. In this stage they can be potentially rewetted. A splat is checked whether it is about to be rewetted or not by simply comparing wet map within it. If the splat area is highly wet, age is above dry-out limit then it is checked against a random value. If all three cases yield true, the splat is rewetted.

5.5 Water dispersion through capillary layer of paper

Although, DiVerdi et al omitted water flow in the paper capillary layer, it is a realistic phenomenon which can be easily modeled. This is done by simply, redistributing water to dry pixels which are surrounded by many wet pixels at each time step in `UpdateWetMap` function.

5.6 Brush Types

All five brush types described in DiVerdi et al are implemented, as outlined in 4.1.5. Besides, another variant of Blobby brush type, blobby2 is also implemented.

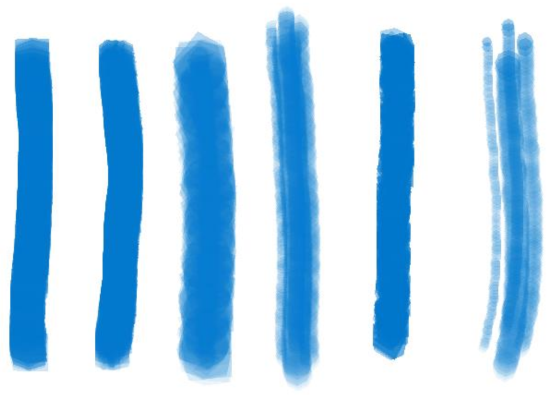

Figure 4. Brush types: from left to right: simple, wet-on-dry, and wet-on-wet, blobby, crunchy, blobby2, with brush size: 37px

5.7 Rendering and Color Blending

Rendering is simply done with OpenGL facility. Following functions are called,

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,
      GL_ONE_MINUS_SRC_ALPHA);
```

This allows additive color blending as desired in watercolor paintings.

5.8 User interface

A simple user interface is built, yet with major functionalities. A nifty UI can be built by choosing any platform dependent UI system, for Windows or Linux, using QT, .Net, Win8 etc. However, this project concentrated on core of the algorithm rather than UI details and submits the core algorithm implementation with all user interfacing mapped to some key strokes.

5.8.1 Choice of colors

To mimic color palette several letter strokes are taken for colors, as in Table 1

Table 1. Keystrokes and colors

Keystroke	Color Name	Color value (hex)
w	White	FFFFFF
k	black	000000
a	Ash	808080
A	Ash (darker)	B0B0B0
e	Eggshell	F0F0F0
E	Eggshell dark	F0F0C0
F	Fawn	E5AA70
r	Red	FF0000
R	Dark red	8B0000
p	Pink	FF8080
P	Pink, light	FFC0C0
g	Green	00FF00
G	Green, dark	008000
b	Blue	0000FF
B	Blue, dark	000080
s	Sky Blue	80C0F0
S	Sky blue, light	B0E0F0
v	Violet	EE82EE
m	Magenta	FF00FF
z	Violet-purple	8000FF
y	Yellow	FFFF00
o	Orange	FF8C00
O	Orange, dark	FF4500
l	Lime green	80FF00
c	Cyan	00FFFF
F	Fluorescent green	00FF80
t	Teal	0080F0
d	Dusk red	561010
D	Dusk dark	210808

5.8.2 Saving the image

This project utilized SOIL open source library function to save drawn paintings. To do so, current buffer is loaded in an array for RGBA values and then called with SOIL image save function. User can use the `Ctrl+S` keystroke to save current painting.

5.8.3 Wet map hints

For ease of users, wet maps is hinted by putting near-white grey in wet pixels of wet map. This is helpful for users to intuitively know which part of paper is wet. See

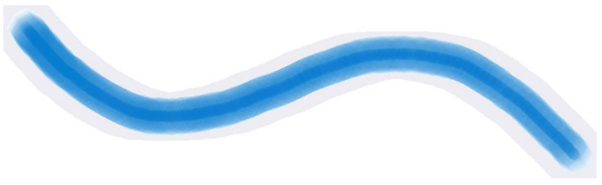

Figure 5. Wet map hint around a stroke

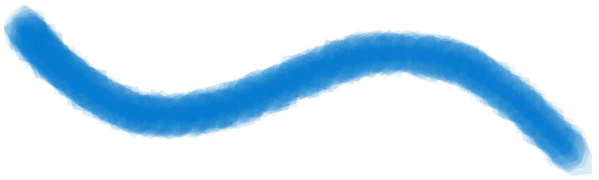

Figure 6. Same stroke, rendered

5.8.4 Selecting brushes

Use can chose from six implemented brushes, as in Table 2. Table 3 shows how brush sizes can be changes in pixels.

Table 2. Keystrokes to select brush type

Keystroke	Action
1	Chose brush type: Simple
2	Chose brush type: Wet-on-dry
3	Chose brush type: Wet-on-wet
4	Chose brush type: Blobby
5	Chose brush type: Crunchy
6	Chose brush type: Blobby2

Table 3. Keystrokes to select brush type

Keystroke	Action
+	Increment brush size (pixels)
-	Decrement brush size (pixels)

6 Results

Following images are created from the app to display results. Figure 7 shows the application recreating many watercolor features including edge darkening, Non uniform pigment density, Color Blending, Glazing, Rewetting, Backruns, and Feathering. Figure 8 shows some painting made with the application.

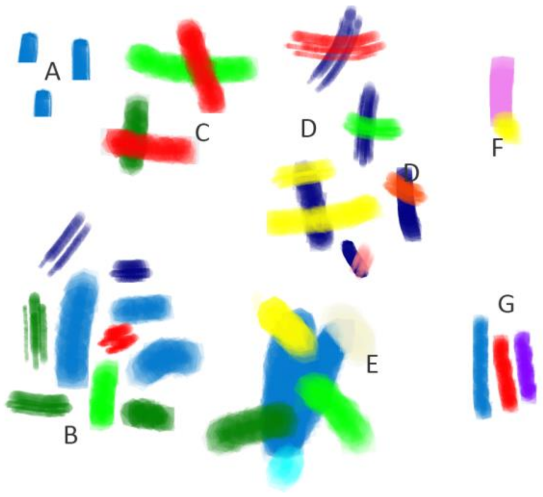

Figure 7. Watercolor effects recreated in the app: Edge darkening (A), Non uniform pigment density (B), Color Blending (C), Glazing (D), Rewetting (E), Backruns (F), Feathering (G)

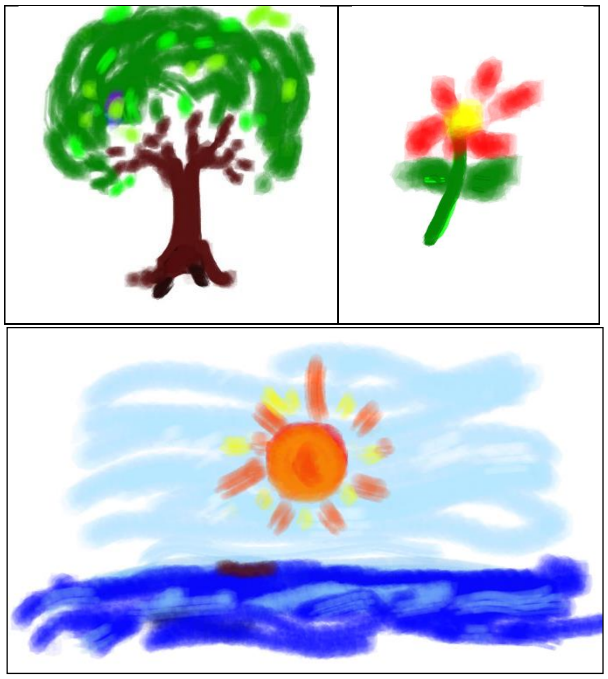

Figure 8. Some paintings created by author in the application built

7 Conclusion

This project has successfully re-implemented referred work. It runs in real time as suggested in original work. It can also salient watercolor painting features.

8 Bibliography

[1] S. DiVerdi, A. Krishnaswamy, R. Mech and D. Ito, "Painting with Polygons: A Procedural Watercolor Engine," *IEEE Transactions on Visualization and Computer Graphics,* Vols. 19, no. 5, pp. 723-735, 2013.

[2] C. J. Curtis, S. E. Anderson, J. E. Seims, K. W. Fleischer and D. H. Salesin, "Computer-generated watercolor," in *Proc. ACM SIGGRAPH*, 1997.

[3] T. Van Laerhoven, J. Liesenborgs and F. V. Reeth, "Real-time watercolor painting on a distributed paper model," in *Computer Graphics International, 2004. Proceedings*, 2004.

[4] S. I. E. Lei and C.-F. Chang, "Real-time rendering of watercolor effects for virtual environments," in *Advances in Multimedia Information Processing-PCM*, 2004.

[5] N. S.-H. Chu and C.-L. Tai, "MoXi: Real-Time Ink Dispersion in Absorbent Paper," *ACM Transactions on Graphics (TOG),* vol. 24, no. 3, pp. 504-511, 2005.

[6] A. Bousseau, M. Kaplan, J. Thollot and F. X. Sillion, "Interactive watercolor rendering with temporal coherence and abstraction," in *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, 2006.

[7] D. Small, "Simulating watercolor by modeling diffusion, pigment, and paper fibers," in *Electronic Imaging'91*, San Jose, CA, 1991.

[8] M. You, T. Jang, S. Cha, J. Kim and J. Noh, "Realistic paint simulation based on fluidity, diffusion, and absorption," *Computer Animation and Virtual Worlds,* vol. 24, no. 3-4, pp. 297-306, 2013.

[9] D. N. M. Nick Foster, "Realistic Animation of Liquids," 1996.

[10] S. Worley, "A cellular texture basis function," in *SIGGRAPH*, 1996.

[11] A. Lake, C. Marshall, M. Harris and M. Blackstein, "Stylized rendering techniques for scalable real-time 3D animation," in *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, 2000.

[12] J. M. Squyres and A. Lumsdaine, "A componenet architecture for LAM/MPI," in *Preoceedings of 10th European PVM/MPI Users' Group Meeting*, Venice, Italy, 2003.

APPENDIX A: CLASSES USED IN THE PROJECT

```
class Stroke
{
public:
	GLfloat x;
	GLfloat y;
	GLint color;
	BrushType brushType;
	GLushort strokePx;
	GLshort bx; //motion bias x
	GLshort by; //motion bias y

Stroke(GLfloat x=0, GLfloat y=0, GLint color=0x0077CC,

GLuint strokePx=5, BrushType brush=BrushType::Simple);

void init(GLfloat x, GLfloat y, GLint color,
	GLuint strokePx=5, BrushType brush=BrushType::Simple);
};

class  Stamp : public Stroke
{
public:
	void copyStroke(Stroke stroke);
private:
};

class Splat{
protected:
	GLfloat x[N];
	GLfloat y[N];
	GLint color;
	GLfloat splatSize;
	//params
	GLshort bx; //motion bias x
	GLshort by; //motion bias y
	GLshort a;//age
	GLubyte  r;//roughness [1~255px]
	GLubyte  f;//flow percentage [0-100]
	GLubyte  o;//opacity

private:
	GLfloat area();
	void zeroout();
	void rewet(const GLushort wetmap[][HEIGHT]);
public:
void init(GLint splatPx, GLfloat pos_x, GLfloat pos_y, GLint color, GLshort bias_x,
	GLshort bias_y, GLushort age, GLubyte roughness, GLubyte flow_pct, GLubyte opacity);
void advect(const GLushort wetmap[][HEIGHT]);
void draw(int i);
};
```