Multiscale Moment-Based Painterly Rendering

DIEGO NEHAB[1] LUIZ VELHO[2]

[1]PUC-Rio – Pontifícia Universidade Católica, Rio de Janeiro
diego@tecgraf.puc-rio.br
[2]IMPA – Instituto de Matemática Pura e Aplicada
lvelho@impa.br

Abstract. In this paper we present a new method for painterly rendering of images. Our method extends the image-moment stroke placement algorithm in two ways: we employ a multiscale scheme for computing strokes and we provide a parametrized mechanism for controlling stroke distribution. In addition, we present a specialized image abstraction for the algorithm.

1 Introduction

The evolution of computer graphics led naturally to the development of different types of visualization techniques. Initially, the focus was on photorealistic rendering, where the goal is to generate synthetic images that are indistinguishable from real photographs [17]. More recently, there has been a growing interest in non-photorealistic rendering techniques that emphasize the expressive depiction of visual information [3].

Non-photorealistic rendering is, by definition, a very broad area of research that encompasses many rendering styles in various application contexts. Two useful criteria for the classification of non-photorealistic rendering techniques are: the type of source data; and the nature of the simulated process.

Techniques are classified according to source data into *object-space* methods that use the 3D model of a scene to create the rendered image [10, 14], and *image-space* methods that work directly on the 2D image [4, 5]. Hybrid methods take advantage of both 3D and 2D data to produce the final result [8].

Most non-photorealistic techniques are inspired in traditional visual art forms, such as painting (oil [5], watercolor [1, 18]), drawing (pen-and-ink [7, 2, 22], pencil [19, 21], charcoal [23]), and printing (etching, engraving [12, 13]). Here, both the physical process and the medium provide a paradigm for computation and interaction.

In this paper we present a new method for painterly rendering of images that improves upon previous work in the area.

1.1 Related work

Painterly rendering simulates the appearance of painted images. The basic primitive in this technique is a brush stroke. Images are generated by applying a sequence of brush strokes to a 2D canvas. A brush stroke has various attributes, such as position, shape and color [20].

In object-space methods, brush strokes are first associated with the 3D geometry of objects in a scene, and then projected to the image plane defined by a virtual camera [10]. In image-space methods, brush strokes are placed on the output image, based on 2D information derived from input images [4].

Interactive methods allow the user to guide the rendering process in a manual or semi-automatic manner, by indicating where strokes should be placed [4]. Non-interactive methods render the image automatically based on input parameters and data analysis [5]. Some methods process a sequence of images exploiting temporal coherence [9, 6].

The technique described in this paper is a non-interactive, image-space method. It is based on the image moment-based stroke placement algorithm [15, 16]. The main original contributions in our work are: a multiscale scheme for computing the strokes, a parametrized mechanism for controlling stroke distribution, and an image abstraction specially optimized for the algorithm.

1.2 Overview

This paper is organized as follows. Section 2 reviews the moment-based painterly rendering method. Section 3 describes the additions proposed in the new method. Section 4 concludes with final remarks and a discussion of ongoing work.

2 Review of the image-moment painterly rendering

Given a source image and a stroke template image, the painterly rendering algorithm outputs a painted version of the source image. The method proceeds as an artist who progressively strokes a canvas trying to reproduce the source image on it. The algorithm outlined in the this section, which is the result of previous work [16], generates images similar to that shown in Figure 1(b).

The process can be divided into two phases: analysis and synthesis. In the analysis phase, a list of strokes is calculated from the source image. In the synthesis phase, the strokes are painted over a blank canvas. We proceed with the description of the synthesis process, which will make clear to the reader the requirements to be fulfilled by the analysis process, explained subsequently.

2.1 The synthesis phase

The synthesis process receives as its input a list of strokes to be painted. The algorithm works with strokes that are described by the following set of attributes: color, location, orientation and size. According to these parameters, painting a stroke on a canvas corresponds to the process of scaling, rotating, and using the stroke template image as a transparency map to be blended on the output image, in the correct position and with the appropriate color.

Figure 2: Example of stroke list painting.

The partial result for the stroke list used to paint Figure 1(b) can be seen be seen in Figure 2. The computation of such a list of strokes is the task of the analysis phase of the algorithm.

2.2 The analysis phase

The first step in the creation of a stroke list is the definition of the stroke distribution. In a second step, each stroke in the distribution has its parameters computed.

The stroke distribution is based on the observation that high-frequency details in the source image should be represented by many small strokes, whereas low-frequency regions should be represented by fewer larger strokes.

The stroke parameters result from the analysis, with the help of the image-moments theory, of the neighborhood in the source image where the stroke is to be placed. Since the same tools provide the information needed for the computation of the stroke distribution, their use is introduced first.

2.2.1 Computing stroke parameters

The goal of each stroke is to approximate a neighborhood in the source image. From each neighborhood, the image-moment based approach determines the corresponding stroke parameters in two steps. The first step computes a color difference image between the region and the color at the center of that region. The second step determines the remaining stroke parameters based on the image moments of the color difference image created in the first step.

The color difference image attempts to measure the distance between the color of the stroke and the color of each point in the source image neighborhood being considered. Ideally, the resulting image shows a picture of the shape that a stroke of that color should have if was to approximate the region. In other words, the operation isolates the segments of the region that can be better represented with the chosen stroke color.

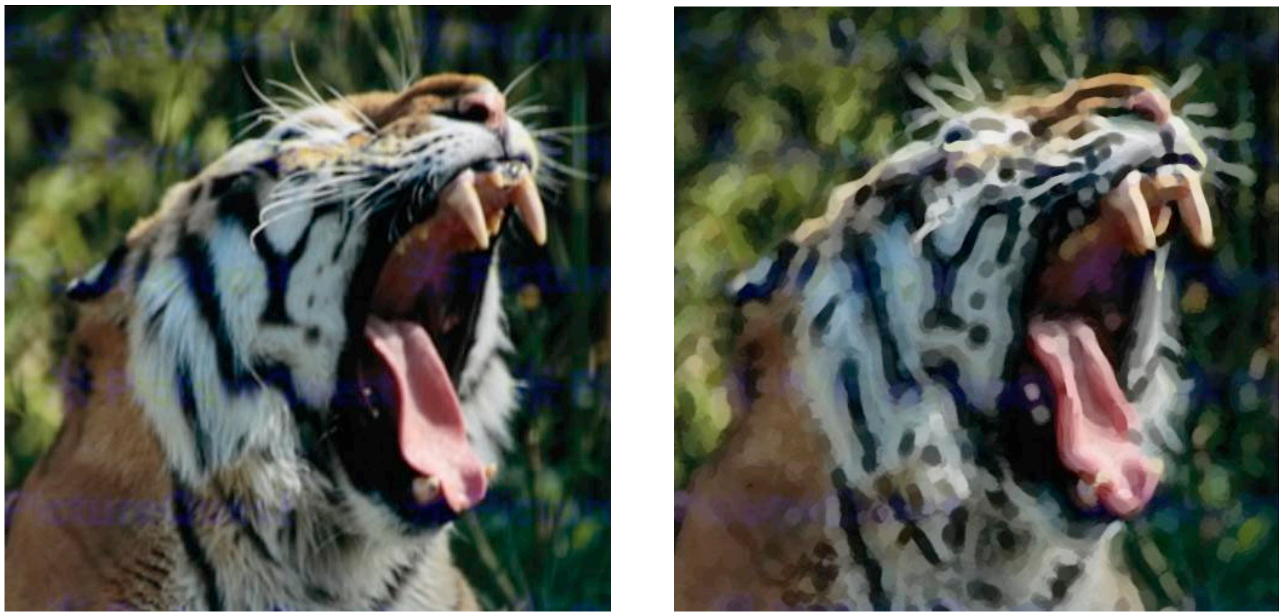
The quality of the computed parameters depends on the quality of the segmentation produced by the color difference image. In particular, low contrast images may consistently give rise to similar stroke parameters. To increase the contrast, a function is used to map color difference values into the intensity values actually stored in the resulting image.

The stroke parameters corresponding to a rectangle that closely matches the shape of the color difference image of the region being approximated is then computed with the help of image moments. Image moments are summations over all pixels of the image, which capture the notions of area, position and orientation. See [16] for formulas and a detailed explanation of the theory.

2.2.2 Determining the stroke distribution

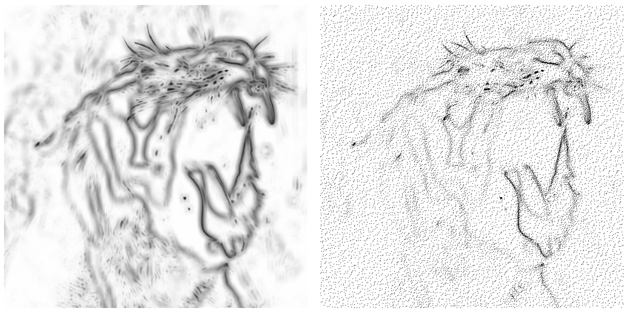
The frequency information needed for the definition of a stroke distribution is obtained with the computation of a stroke area image — an image in which the value of each pixel corresponds to the area of a stroke approximating its neighborhood. The area of a stroke associated to a position in the source image is computed from the color difference image between the color at the position and its neighboring image, also with the use of image moments.

The stroke distribution is given by a stroke positions image, in which each position is marked by a dot. This image is generated from the stroke area image by a special monochrome dithering algorithm. The algorithm used to create the stroke distribution must be designed to concentrate strokes around the dark regions of the stroke area image, and to avoid large regions without strokes. To this end, the previous study used a modified version of a space-filling curve dithering algorithm, in which the accumulated intensity values were inversely proportional to the area of the stroke.



(a) (b)

Figure 1: Painterly rendering process



(a) (b)

Figure 3: Stroke area and positions images.

Figure 3(b) shows an example of how the stroke positions image should look. This image was created from the stroke area image of Figure 3(a) by our own parametrized dithering algorithm. This new algorithm is detailed in Section 3.

3 Original contributions to the algorithm

The ideas presented so far describe a complex process to create images that resembles human hand painting, as seen in Figure 1. Some aspects of the process can be improved, other parts can be implemented in a way that deserves documentation. This session describes what was added by our research.

3.1 Multi-resolution analysis

As expected, the stroke parameters, computed with the aid of the color difference images and the image moments theory, correctly approximate local source image neighborhoods. Unfortunately, although small details can be captured within a neighborhood, features that occupy more than the size of a single neighborhood cannot, and must therefore be represented by a group of smaller strokes. Unless the used stroke template image has a low opacity overall, this composition becomes evident. Furthermore, a needlessly large amount of small strokes must be used to represent what could be approximated by fewer larger strokes.

In order to capture strokes over a wider range of sizes, previous work [16] suggests, as a possible improvement, that an adaptive method should be developed to vary the size of the neighborhoods being analyzed throughout the source image. As an alternative solution, we use a multi-resolution approach during the analysis phase of the algorithm. Stroke lists are collected for each resolution in a pyramid built from the source image. The painted result at each level is obtained by blending its strokes over the result of the lower resolution level. The implementation[1] of these ideas can be seen in Program 1.

```
function MultiResolutionPainterlyRender(Source, Stroke, S, L)
  local Pyramid = { Source }
  local w, h
  local l = 2
  while l <= L do
    w = GetWidth(Pyramid[l-1])/2
    h = GetHeight(Pyramid[l-1])/2
    Pyramid[l] = Copy(Scale(Pyramid[l-1], w, h, New()), New())
    l = l + 1
  end
  local Canvas = BlankCanvas(w, h)
  l = L
  while l >= 1 do
    w, h = GetWidth(Pyramid[l]), GetHeight(Pyramid[l])
    Canvas = Copy(Scale(Canvas, w, h, New()), New())
    local Sp, E = Spread(S, L-l), Enhance(S, L-l)
    local List = StrokeList(Pyramid[l], S, Sp, E)
    Canvas = PaintStrokeList(Stroke, List, Canvas)
    l = l - 1
  end
  return Canvas
end
```

Program 1: The multi-resolution painter algorithm.

Figure 5 shows an example of painterly rendered image created by the multi-resolution method. Although finer details than those seen in Figure 1(b) are clearly visible, Figure 5 required only 14475 strokes, whereas Figure 1(b) required 18917.

Figures 4(a) to (d) depict the strokes at each resolution level that, when composed together, create the final image in Figure 5. The images are shown scaled to the same resolution to simplify comparison, and to illustrate the steps followed by the algorithm. Notice the different stroke position distributions at different levels, computed from blurred images coming from the multiresolution pyramid.

3.2 Parametrized stroke positions image

The stroke list for higher resolution levels should not only concentrate strokes on high frequency areas of the source image, but also avoid placing strokes over lower frequency areas. Otherwise, strokes coming from lower resolution levels would be consistently obscured by the strokes coming from higher resolution levels, producing a result no better then the single-resolution approach.

To avoid this problem, the new procedure used to create stroke positions images accepts two parameters, referred to as the spreading and the enhancing factors. The spreading factor places an upper bound on the maximum distance between strokes, effectively controlling the overall stroke density. The enhancing factor controls the degree to which the density of strokes increases when close to the edges found in the stroke area image.

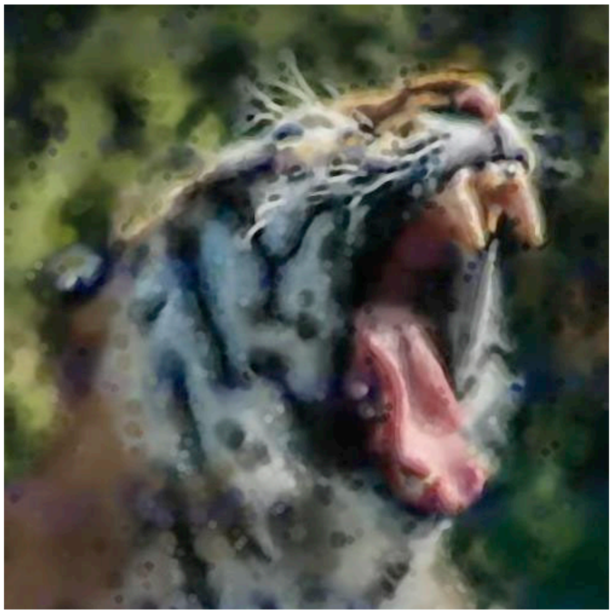


Figure 5: Multi-resolution painted image.

Before being considered, the value of each pixel is passed through the function defined by Equation (1), along with spreading and enhancing parameters.

$$\mathrm{se}(v, s, e) = \frac{1}{(s^2 - 1)v^e + 1} \qquad (1)$$

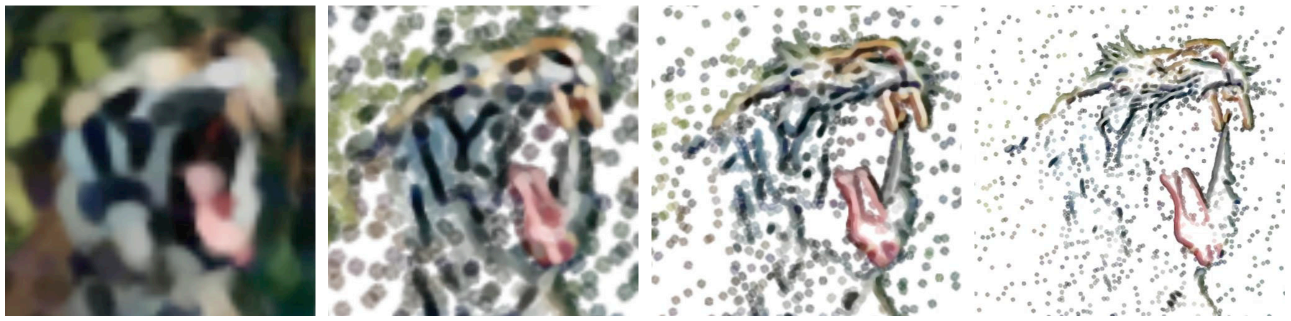
This function was designed to map the value 0 into 1 (small area values generate more strokes) and the value 1 into $\frac{1}{s^2}$ (even large areas contribute to the generations of strokes). The value $\frac{1}{s^2}$ was chosen so that a stroke position is issued after at most s^2 pixels. Furthermore, the effect of the enhancing factor is to accentuate small input values, which are exactly those close to the black edges of the stroke area image.

At each level, the spreading and enhancing factors are computed as functions of the level and the size of the neighborhoods. Experimental functions that presented good results and are used in our implementation are:

$$\mathrm{spread}(s, l) = (s + 0.3) \cdot l \qquad (2)$$
$$\mathrm{enhance}(s, l) = 3 \cdot (l + 1) \qquad (3)$$

In our implementation, the dithering proceeds by traditional error diffusion. Accordingly, foreach pixel, the error accumulated due to truncation is spread to three of its adjacent pixels. However, in order to avoid undesirable periodic artifacts, at each pixel we randomly shuffle the coefficients

[1]The source code is presented in the Lua programming language, embedded in our implementation. A full description of the software can be found in [11].



(a) Level 4 (b) Level 3 (c) Level 2 (d) Level 1

Figure 4: Strokes at different resolution levels.

used to diffuse the error. Although the randomization of the dithering process may not be useful to produce high quality images, we are not interested in photo-realism. Therefore, these simple ideas are enough to produce results with the desired properties, as seen in Figure 6.

3.3 Optimized Image abstraction

During the two phases of the algorithm, the basic computations performed over images are: stroke area image, stroke positions image, color image difference, image moments, scaling, rotation and blending. The choice of an appropriate image abstraction can simplify the task of efficiently implementing these operations. In particular, the abstraction should simplify computations involving image neighborhoods and avoid unnecessary memory allocations.

```
/* image data type */
typedef struct Tmono {
  float *buffer;
  int width, height, row;
} Tmono;
typedef Tmono *Pmono;
```

Program 2: C structure representing an image.

The data structure described by Program 2 can be used to store information about a newly allocated image (such as the `source` image of Figure 7) and can also store information representing part of a previously allocated image (such as the `shared` image on the same figure). The `row` field always relate to the image that *owns* the buffer, i.e. the image that was actually allocated, and allows routines to correctly determine the position of each pixel in shared regions.

Our experience shows that adapting a image processing algorithm to deal transparently with the above image representation, either as input or output, is an effortless task. Furthermore, the adapted version usually suffers no measurable performance degradation. Therefore, every image processing function in our implementation makes no assumptions whether the buffers are shared or owned by the images structures that point to them. A simple example in C of such function is presented by Program 3.

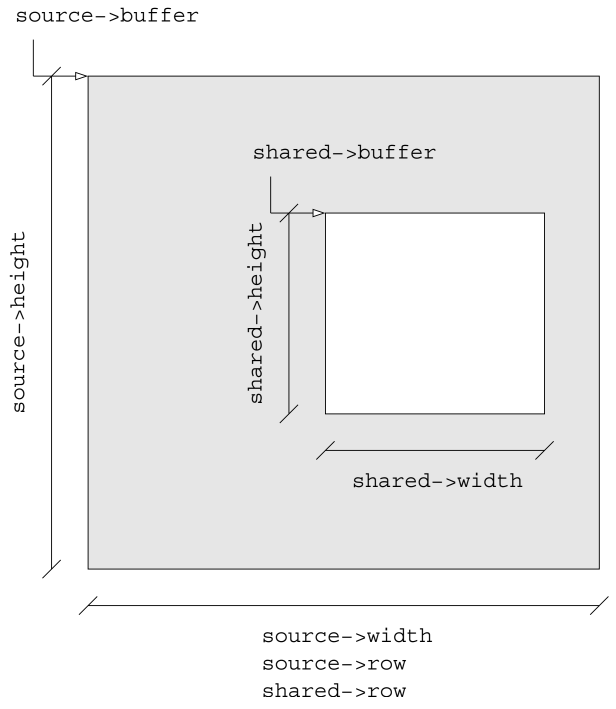
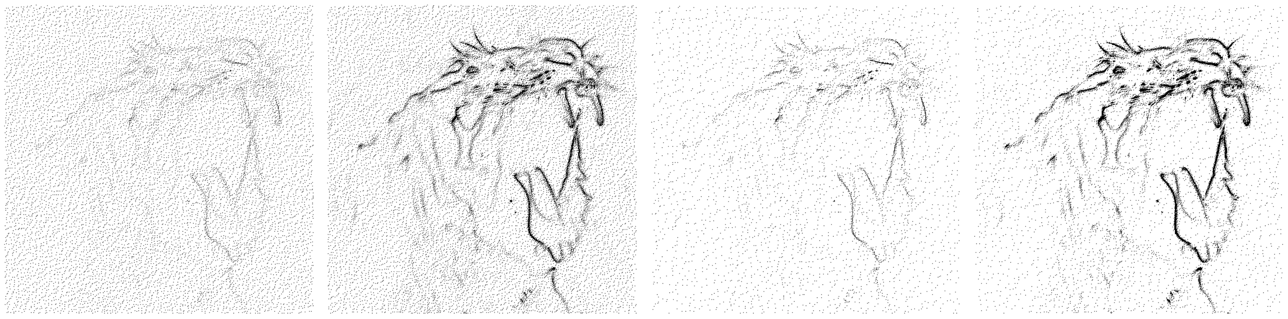


Figure 7: Meaning of image structure fields.

This agreement allows the extensive use of shared regions throughout all parts of the algorithm. Instead of supplying neighborhood limits to each function in the API, an image structure representing the neighborhood can be created and the functions can operate directly over them. The



(a) $s = 5, e = 2$ (b) $s = 5, e = 5$ (c) $s = 10, e = 4$ (d) $s = 10, e = 8$

Figure 6: Parametrized stroke positions images.

```
int mono_clear(Pmono in, float c)
{
  int x, y, skip = in->row - in->width;
  float *p = in->buffer;
  for (y = 0; y < in->height; y++) {
    /* process row */
    for (x = 0; x < in->width; x++)
      *p++ = c;
    /* skip part of row not belonging to region */
    /* if image owns the buffer, skip is 0 */
    p += skip;
  }
  return 1;
}
```

Program 3: Clearing an image.

code is greatly simplified by concentrating all clipping logic in a single function and, since image parts are shared, there is no performance loss due to memory allocation.

4 Conclusions

In this work we introduced a multi-resolution approach to painterly rendering method by the local source image approximation. The development of this research gave rise to a parametrized stroke distribution algorithm, easily implemented. Finally, a specially designed image abstraction simplified and optimized the implementation.

Figures 8 and 9 illustrate some results of using our method. Figure 8 shows two versions of the canoe image painted with different parameters for stroke distribution and size. Figure 9 shows the train image. Note how the smoke is rendered with large round strokes, while the railroad is rendered with long thin strokes.

4.1 Future work

The image moment theory provides a powerful tool in the creation of local source image approximations. The quality of these approximations, however, is strongly dependent on the quality of the local color difference images. In a future work, we intend to investigate alternatives to the color difference images, attempting to produce better controllable results.

In the present work, strokes computed at each level have no influence on strokes computed at higher resolution levels. One way this information could be taken into account would be to subtract the computed strokes from the source image at lower resolution levels before computing new strokes at higher resolutions.

The encoding and storage of stroke lists will also be studied. By compacting the information provided by the lists, it is possible to represent painterly rendered images in a space efficient way. If the new segmentation techniques lead to substantial improvements, it may even be possible to encode photo-realistic images, transforming the scheme into an image compression algorithm.

Acknowledgments

This work was developed at the VISGRAF Laboratory of IMPA and TECGRAF of PUC-Rio.

We would like to thank Danilo Tuler de Oliveira and Diogo Vieira Andrade who took part in the early stages of this research.

The authors are partially supported by research grants from the Brazilian Council for Scientific and Technological Development (CNPq) and Rio de Janeiro Research Foundation (FAPERJ).

References

[1] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-generated watercolor. *Proceedings of SIGGRAPH 97*, pages pages 421–430, August 1997.

[2] Gershon Elber. Line Art Illustrations of Parametric and Implicit Forms. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), January – March 1998. ISSN 1077-2626.

[3] Stuart Green, David Salesin, Simon Schofield, Aaron Hertzmann, Peter Litwinowicz, Amy Gooch, Cassidy Curtis, and Bruce Gooch. Non-Photorealistic Rendering. *SIGGRAPH '99 Non-Photorealistic Rendering Course Notes*, 1999.

[4] Paul E. Haeberli. Paint by numbers: Abstract image representations. *Proceedings of SIGGRAPH 90*, 24(4):207–214, August 1990.

[5] Aaron Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. *Proceedings of SIGGRAPH 98*, pages 453–460, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[6] Aaron Hertzmann and Ken Perlin. Painterly rendering for video and interaction. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 7–12, June 2000.

[7] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, July 2000. Held in New Orleans, Louisianna.

[8] Allison W. Klein, Wilmot W. Li, Michael M. Kazhdan, Wagner T. Correa, Adam Finkelstein, and Thomas A. Funkhouser. Non-photorealistic virtual environments. *Proceedings of SIGGRAPH 2000*, pages 527–534, July 2000. ISBN 1-58113-208-5.

[9] Peter Litwinowicz. Processing images and video for an impressionist effect. *Proceedings of SIGGRAPH 97*, pages 407–414, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[10] Barbara J. Meier. Painterly rendering for animation. *Proceedings of SIGGRAPH 96*, pages 477–484, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[11] Diego Nehab and Luiz Velho. A lua implementation of image moment-based painterly rendering. Technical Report 01-11, IMPA, Visgraf Laboratory, 2002.

[12] Victor Ostromoukhov. Digital facial engraving. *Proceedings of SIGGRAPH 99*, pages 417–424, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[13] Yachin Pnueli and Alfred M. Bruckstein. Digdurer - a digital engraving system. In *The Visual Computer*, volume 10, pages 277–292, August 1994.

[14] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. *Proceedings of SIGGRAPH 2001*, pages 579–584, August 2001. ISBN 1-58113-292-1.

[15] Michio Shiraishi and Yasushi Yamaguchi. Image moment-based stroke placement. Technical Report skapps3794, University of Tokyo, Tokyo Japan, May 1999.

[16] Michio Shiraishi and Yasushi Yamaguchi. An algorithm for automatic painterly rendering based on local source image approximation. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*, pages 53–58, June 2000.

[17] François X. Sillion. The state of the art in physically-based rendering and its impact on future applications. *Second Eurographics Workshop on Rendering (Photorealistic Rendering in Computer Graphics)*, pages 1–10, 1994. Held in New York.

[18] David Small. Simulating watercolor by modeling diffusion, pigment, and paper fibers. *Proceedings of SPIE '91*, February 1991.

[19] Mario Costa Sousa and John W. Buchanan. Observational models of graphite pencil materials. *Computer Graphics Forum*, 19(1):27–49, March 2000. ISSN 1067-7055.

[20] Steve Strassmann. Hairy brushes. *Siggraph*, 20(4):225–232, August 1986.

[21] Saeko Takagi, Masayuki Nakajima, and Issei Fujishiro. Volumetric modeling of colored pencil drawing. *Pacific Graphics '99*, October 1999. Held in Seoul, Korea.

[22] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. *Proceedings of SIGGRAPH 94*, pages 91–100, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[23] Eric Wong. Artistic rendering of portrait photographs. Master's thesis, Cornell University, 1999.

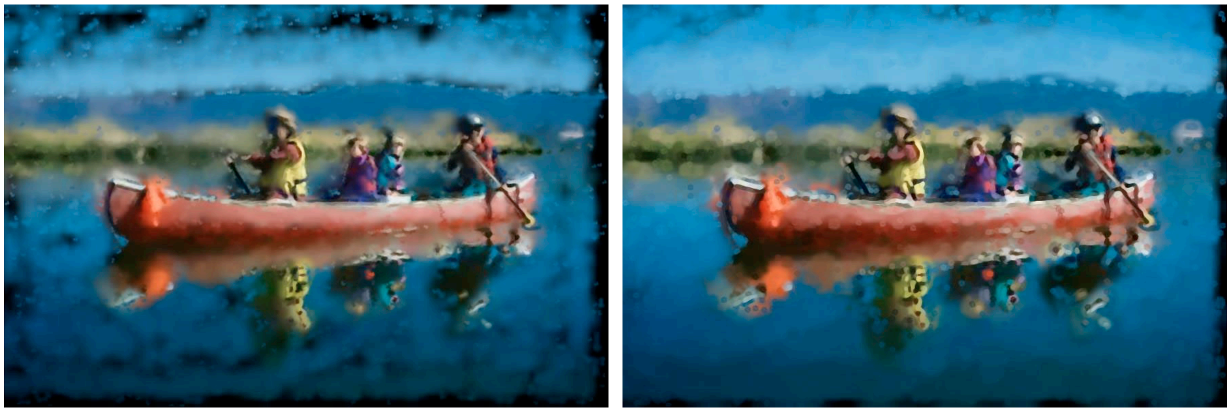


Figure 8: Canoe: painted with different parameters.

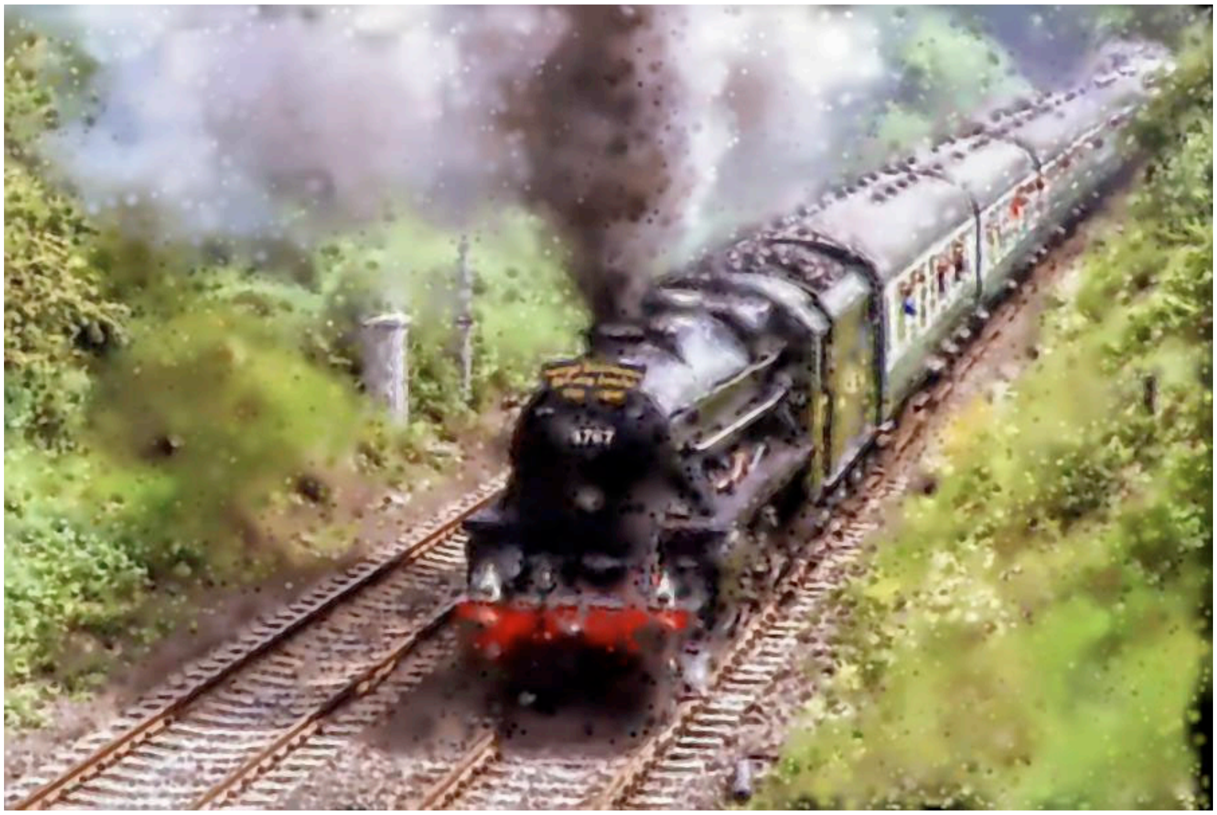


Figure 9: Train.