Using Line Integral Convolution to Render Effects on Images

Ricardo David Castañeda Marín

VISGRAF Lab

Instituto Nacional de Matemática Pura e Aplicada

A thesis submitted for the degree of

Master in Mathematics-Computer Graphics

Feb 2009

Dedicated To My Family...

Acknowledgements

I am very grateful to my parents Mariela and Gildardo, my sisters Eliana and Andrea and my friends Alejandro Mejia, Julian Lopez and Andres Serrano who have supported all my studies and have encouraged me to carry on with my academic and personal life; this monograph is dedicated to all them. Many thanks to my academic advisor Dr. Luiz Henrique de Figueiredo who accepted my request on working in this topic. Thank you for all the suggestions and for helping me in writing this dissertation. Particular thanks go also to Dr. Luiz Velho for his valuable tips and time we spent discussing new ideas. Those sessions were very helpful, thank you. Also I am in debt with Emilio Ashton Vital Brazil who gave generously his time to offer explanations especially with the pencil effect approach of Section 4.3.

Contents

List of Figures

1

Introduction

In this monograph we are going to expose the use of some ideas involved in the Line Integral Convolution (LIC) algorithm for the generation of many Non-Photorealistic Renditions on arbitrary raster images. In other words, our main objective is to create images that could be considered as pieces of visual art generated using ideas from the LIC algorithm. From this point of view, the output of our algorithms does not need to be considered as right or wrong, an aesthetic judgement will be more appropriate. That is what we are expecting from the reader.

It is well known that even since its roots Computer Graphics procedures have been used by artists for both aesthetics and commercial purposes. Our motivation comes from the original paper on LIC (1) which explores another kind of applications considered as realistic effects, more specifically blur-warping. In a similar way, we searched for other uses of these LIC ideas mixing the already known NPR techniques like painterly rendering and pencil sketches. This monograph is then the result of such experiments.

The material presented demands basic knowledge of ordinary differential equations and vector calculus. Chapter 2 defines and explains the LIC algorithm to visualize the structure of planar vector fields using white noise as the input image. Since our approach uses arbitrary vector fields to guide the effects, two methods for designing these vector fields are given in chapter 3. Chapter 4 discusses the actual NPR effects results. There is also an appendix A which exposes the implementation of our procedures using the CImg library for image processing and visualization of the results, and an appendix B which is a gallery with our results.

2

Line Integral Convolution

This chapter is about the main algorithm of this text: Line Integral Convolution (LIC). It was introduced in (1) by B. Cabral and L. Leedom at the SIGGRAPH conference in 1993. LIC was designed primarily to plot 2D vector fields that could not be visualized with traditional arrows and streamlines, i.e., fields with high density. Due to the low of performance of the original algorithm on large images, over the years several alternatives of the same calculations have been published to increase the speed and the details of the visualization. We will discuss a fast procedure briefly. Since every other algorithm here will be an extension or a derivation from LIC, it is important to have a good understanding of how it works.

2.1 DDA Convolution

The LIC algorithm takes as input an image and a vector field defined on the same domain. The output image is computed as a convolution of the intensity values over the integral curves of the vector field (see Figure 2.1). In the case we want to visualize the topology and structure of this field, the input image needs to have pixels uniformly distributed and with mutually independent intensities (6). A simple white noise as input image will be enough for what we want here. On the other hand, the input image can be arbitrarily chosen, and the output image will have an effect depending on the input vector field. We will turn to this subject later.

As stated in the original paper, LIC is a generalization of what is known as DDA convolution. The DDA algorithm performs convolution on a line direction rather than

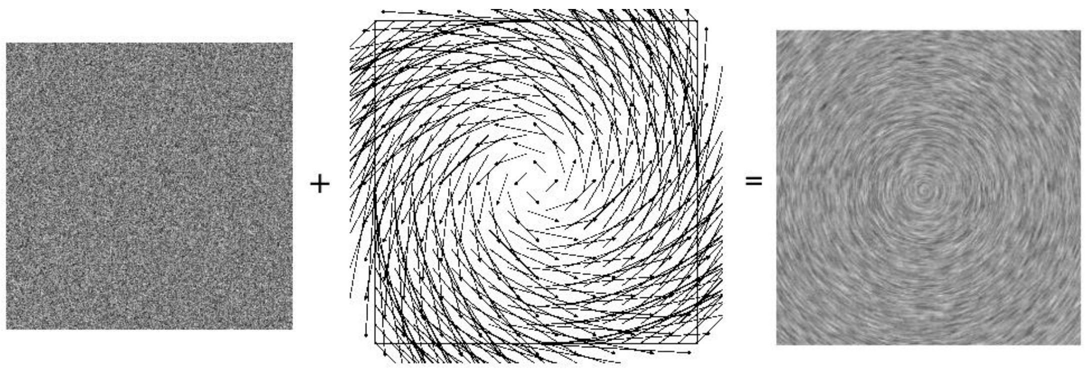

Figure 2.1: LIC algorithm overview - Left to right: White noise, input vector field and output LIC visualization.

on integral curves. For each pixel location (i, j) on the input image I, we want to compute the pixel intensity in the output image $O(i, j)$. For this, DDA takes the normalized vector $V(i, j)$ corresponding to that location and moves in its positive and negative directions some fixed length L. This generates a line of locations

$$l(s) = (i, j) + sV(i, j), \quad s \in \{-L, -L+1, ..., 0, ..., L-1, L\}$$

and a line of pixels intensities $I(l(s))$ of length $2L + 1$. Choosing a filter kernel $K : \Re \to \Re$ with $Supp(K) \subseteq [-L, L]$, the line function $I(l(s))$ is filtered and normalized to generate the intensity output $O(i, j)$:

$$O(i, j) = \frac{1}{2L+1} \sum_{s=-L}^{L} I(l(s))K(l^{-1}(i, j) - s) \equiv \frac{1}{2L+1} I_{l(i,j)} \circledast K$$

The symbol $\circledast$ stands for *discrete convolution* and is responsible for the name of the algorithm. So for each pixel we perform a discrete convolution of the input image with some fixed filter kernel. As a special case, when K is chosen to be a box kernel ($K \equiv 1$ on $[-L, L]$ and $K \equiv 0$ everywhere else), the convolution becomes the average sum of the pixels $I(l(s))$:

$$O(i, j) = \frac{1}{2L+1} \sum_{s=-L}^{L} I(l(s))$$

Fig 2.2 depicts the process. As expected, DDA is very sensitive to the fixed length L, since we are assuming not only that the vector field can be locally approximated by a straight line, but also that this line has fixed length L everywhere, generating a uneven visualization: linear parts are better represented than vortices or paths with high curvature. Line integral convolution remedies this by performing convolution over integral curves.

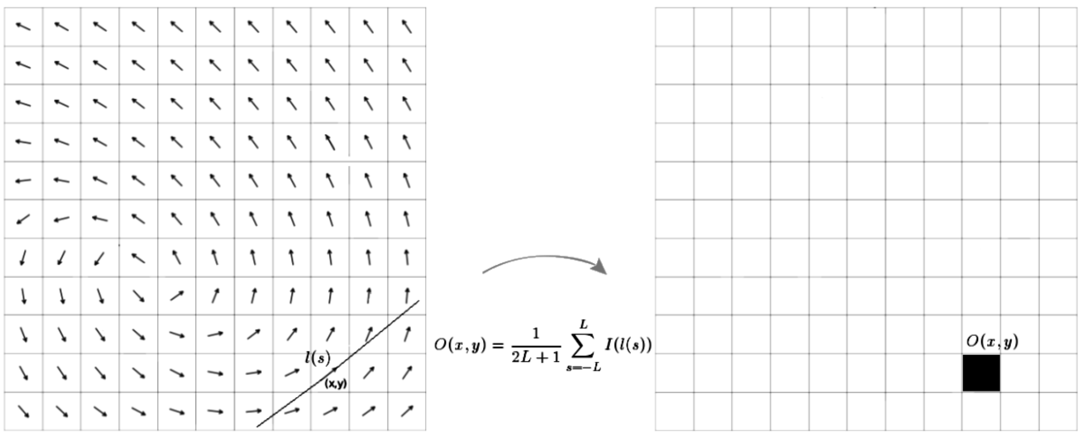

Figure 2.2: DDA convolution - Convolution over a line of pixels. Picture adapted from (1).

2.2 LIC Formulation

LIC can be performed on 2D and 3D spaces. Because we are only concerned with the generation of effects on 2D images, our vector field will have a planar domain. An integral curve of the vector field $v : \Omega \subset \Re^2 \rightarrow \Re^2$, passing over $x_0 \in \Omega$ at time τ_0, is defined as a function $c_{x_0} : [-L, L] \rightarrow \Re^2$ with:

$$\frac{d}{d\tau} c_{x_0}(\tau) = v(c_{x_0}(\tau)), \quad c_{x_0}(0) = x_0$$

that is, a curve solution of the initial value problem

$$\frac{d}{d\tau} c(\tau) = v(c(\tau)), \quad c(0) = x_0$$

Uniqueness of the solution of this ODE is reached when the field locally satisfies a Lipschitz condition. When $\frac{d}{d\tau}c_x(\tau) \neq 0$ for all $x \in \Omega$ and for all $\tau \in [-\tau, \tau]$, every c_x can be reparametrized by arc length s (2). An easy computation of this reparametrization leads to an alternate definition of integral curves:

$$\frac{d}{ds}c_{x_0}(s) = \frac{v(c_{x_0}(s))}{\|v(c_{x_0}(s))\|}, \quad c_{x_0}(s_0) = x_0$$

Basically the generalization from DDA to LIC is done when one changes the line $l(s)$ involved, for the integral curve $c_{l(0)} \equiv c_{(i,j)}$. The new output pixel $O(i,j)$, with s_0 such that $c_{(i,j)}(s_0) = (i,j)$, is computed by LIC as:

$$O(i,j) = \frac{1}{2L+1}\sum_{s=s_0-L}^{s_0+L} I(c_{(i,j)}(s))K(s_0-s) \equiv \frac{1}{2L+1} I_{c_{(i,j)}} \circledast K$$

and simplifying with a box filter:

$$O(i,j) = \frac{1}{2L+1}\sum_{s=s_0-L}^{s_0+L} I(c_{(i,j)}(s))$$

Figure 2.3 shows this process. Notice that we are restricting the integral curve to the interval $[-L, L]$ for some fixed length L like in DDA. In general a good L depends on the vector field and its density. Figure 2.4 show some examples for different values of L and the same vector field.

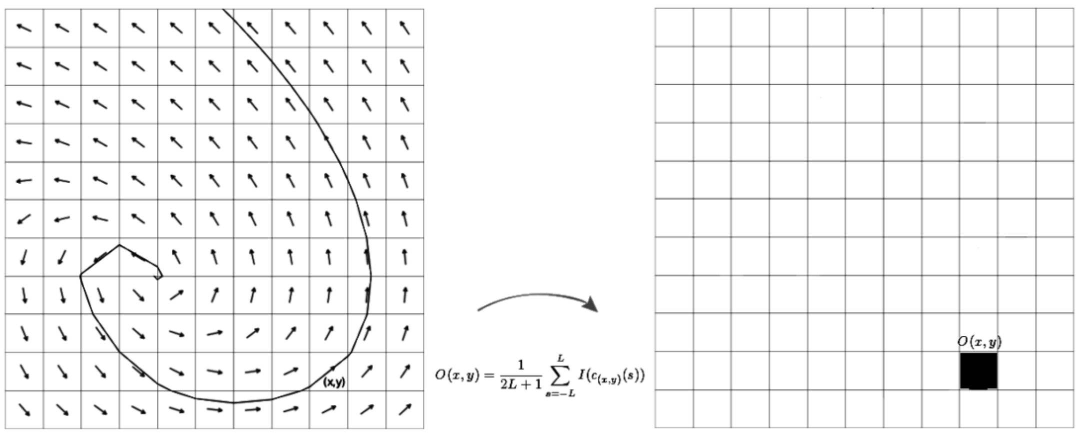

Figure 2.3: LIC - Convolution over a integral curve of pixels. Picture adapted from (1).

To compute the integral curves of the input vector field, the solution of the ODE is obtained by integration:

$$c_{x_0}(s) = x_0 + \int_{s_0}^{s} v(c_{x_0}(s'))ds'$$

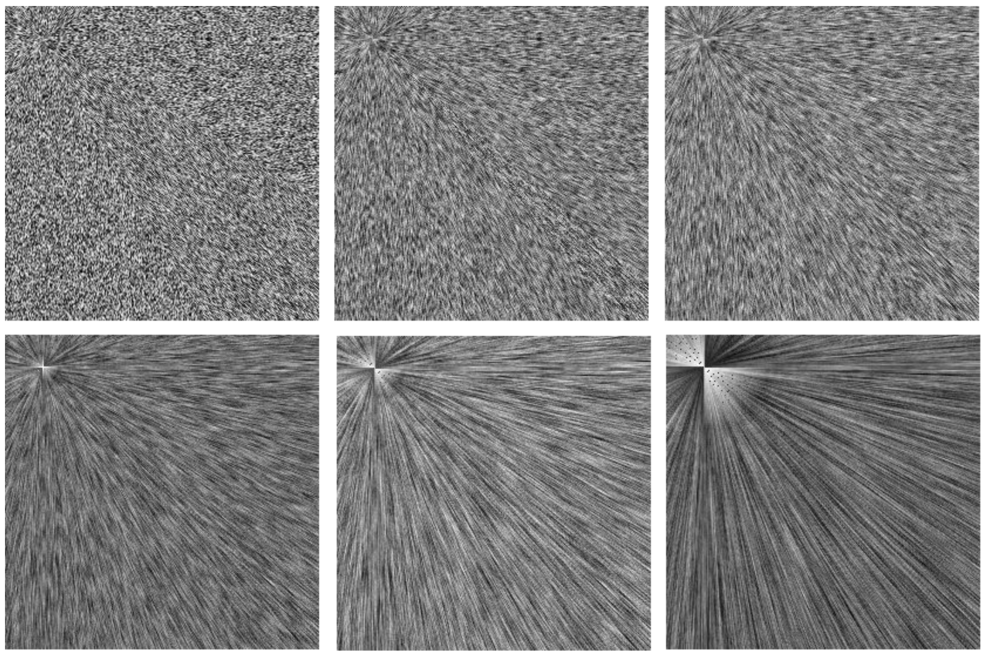

Figure 2.4: Different Values for L - From top to bottom and left to right: Values for L: 1,3,5,10,20, and 50.

The next pseudocode performs a discretization of this equation, storing in an array C the pixel locations of the integral curve. The function `vector_field`(p) returns the vector at point p. The constant ds is the sample rate for the integral curve, see section 2.4 for an explanation.

Computing the integral curve for a pixel $p = (x, y)$:

```
function compute_integral_curve(p){
     V=vector_field(p)
     add p to C
     for (s=0;s<L;s=s+1){  // positive calculations
          x=x+ds*V.x
          y=y+ds*V.y
          add the new (x,y) to C
          compute the new V=vector_field(x,y)
     }
     (x,y)=p  V=vector_field(p) //return to original point and original vector
     for (s=0;s>-L;s=s-1){  // negative calculations
```

```
          x=x-ds*V.x
          y=y-ds*V.y
          add the new (x,y) to C
          compute the new V=vector_field(x,y)
     }
     return C
}
```

To compute the convolution with a box kernel a simple average of the intensities is used:

Computing the convolution along integral curves:

```
function compute_convolution(image,C){
     sum=0
     for each location p in C {
          sum=sum+image(p)
      }
     sum=sum/(2*L+1)  // normalization
     return sum
}
```

Next is the pseudocode of the final LIC algorithm.

LIC Pseudocode with a box kernel:

```
function LIC(image){
     create an empty image O_img
     for each p on image{
          array C=compute_integral_curve(p)
          sum = compute_convolution(image,C)
          set pixel p on O_img to sum
      }
     return O_img
}
```

Note that for a point on a particular integral curve c, its own integral curve is highly related to c. The low performance of the LIC algorithm can be seen there: for each pixel location we have to compute the integral curve passing through that location (without using the already computed integral curves) and perform a convolution with some filter kernel. In the next section we will see how this relations can be explored to increase speed of the LIC algorithm.

2.3 A Fast LIC algorithm

Given that when computed, an integral curve covers lot of pixels, uniqueness of the solution of the ODE implies that the convolution involved in LIC can be reused. Choose a box filter kernel and suppose we have an integral curve of a location (i, j), say $c_{(i,j)}$ and another location along it $c_{(i,j)}(s)$, then their output values are related by

$$O(c_{(i,j)}(s)) = O(i,j) - \sum_{s'=s_0-L}^{s_0-L+s} I(c_{(i,j)}(s')) + \sum_{s'=s_0+L}^{s_0+L+s} I(c_{(i,j)}(s'))$$

Figure 2.5 illustrates this relation. In practice, to reuse an already computed convolution for a set of pixels, a matrix of the same size of the image is created such that each entry stores the number of times that pixel has been visited. The order in which the pixels are analyzed is important for the efficiency of this process. The goal is to hit as many uncovered pixels with each new integral curve to reuse the convolutions as possible, and thus it is not a good choice to make it in a scanline order. Nevertheless, we can adopt another approach in which the image is subdivided in blocks and process the pixels in scanline order on each block. For instance, we take the first pixel of each block and make the calculations, then the second pixel and so on[1].

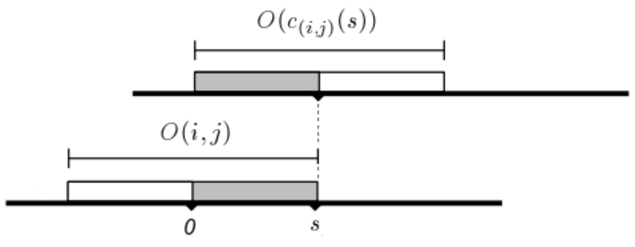

Figure 2.5: FastLIC - Integral curves relation involved in the FastLIC approach. The shaded region of the convolution could be reused.

[1]There are other methods to compute the order to process the pixels, see for example (6; 11).

The following is a pseudocode of the basic FastLIC algorithm (5). This code is used on each block, as stated in the previous paragraph, to ensure reusability of the integral curves.

FastLIC Pseudocode:

```
for each pixel p
    if p hasn't been visited then
        compute the integral curve with center p=c(0)
        compute the LIC of p, and add result to O(p)
        m=1
        while m<L
            update convolutions for c(m) and c(-m)
            set output pixels: O(c(m)) and O(c(-m))
            set pixels c(m) and c(-m) as visited
            m=m+1;
```

2.4 Final Considerations

As you can see from the previous sections, LIC is a simple but powerful tool for visualizing vector fields. In this section I want to make explicit some considerations regarding the implementation of this algorithm. This section is optional for the readers whose interest is the implementation rather than the applications. The already given material is enough for what we want to develop with LIC.

The first consideration is about the space of the variable s. In section 2.1 we define the DDA convolution for discrete values: $s \in \{-L, -L+1, ..., 0, ..., L-1, L\}$. However, in general s is a real variable on the interval $[-L, L]$. The line of locations $l(s)$ is in general generated by sampling this interval, with $l(0) = (i, j)$. It is clear that for some sampling rates this line is not injective, given that our image is a raster image with integer locations (i, j). The DDA line is computed as $l(k\Delta s) = (i, j) + k\Delta s V(i, j)$ with integer $k \in [\frac{-L}{\Delta s}, \frac{L}{\Delta s}]$. If $\Delta s \equiv 1$ we will be back in our original definition. Practical experience (6) shows that using a Δs of about a third or half an image pixel width is enough for good visualizations. The same sampling consideration is applied for integral curves in the general LIC algorithm.

Another thing one should consider when implementing LIC is that the domain of the input I and output image O can be taken as continuous domains rather than a grid of pixels (i, j). Basically we take a continuous rectangular domain and define a set of cells with center a pixel location (i, j). Then to compute the output pixel at location (i, j) one choose a number of samples locations within its correspondent cell, perform the computations and compute an average intensity value. Because increasing the number of samples on each cell increases the run time of the algorithm, a small number is recommended.

Finally, when performing the convolution on pixels near the boundary of the image domain, sometimes the algorithm will try to retrieve an intensity value of a invalid pixel location, because generally the integral curve will leave the image domain. Figure 2.6 illustrates this. One solution is to pad the image with zeros on the boundary. This however will cause sometimes black regions at the image boundaries. In the case we have a vector field defined only on the image grid, we simply extend it arbitrarily and smoothly on the domain (e.g., by repeating values).

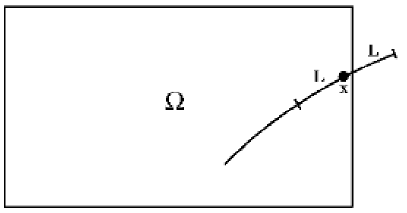

Figure 2.6: Image Domain Overflowing - A padding with zeros is used to avoid overflowing.

3

Vector Field Design

In the previous chapter we saw how planar vector fields with high density can be visualized using the LIC algorithm. We now turn on the subject to design the input vector field. This is motivated by many graphics applications including texture synthesis, fluid simulation and, as we will see in the next chapter, NPR effects on images.

3.1 Basic Design

In section 2.2 we saw that a vector field $v : \Omega \subset \Re^2 \to \Re^2$ defines the differential equation

$$\frac{d}{d\tau}c(\tau) = v(c(\tau))$$

such that for each point $x_0 \in \Omega$, the solution, with intial condition $c_{x_0}(s_0) = x_0$, is the integral curve $c_{x_0}(\tau)$. A singularity of the vector field v is a point $x \in \Omega$ such that $v(x) = 0$.

A very basic vector field design consists of local linearizations and a classification of the singularities. Explicitly, if v is given by the scalar functions F and G, i.e $v(x) = (F(x), G(x))$, then the local linearization of v at a point x_0 is

$$V^*(x) = v(x_0) + Jv(x_0)(x - x_0)$$

where $Jv(x_0) = \begin{pmatrix} \frac{\partial F}{\partial x}(x_0) & \frac{\partial F}{\partial y}(x_0) \\ \frac{\partial G}{\partial x}(x_0) & \frac{\partial G}{\partial y}(x_0) \end{pmatrix}$ is the Jacobian matrix evaluated at the point x_0. When x_0 is a singularity we have

$$V^*(x) = Jv(x_0)(x - x_0)$$

We will assume that for singularities x_0 its corresponding Jacobian matrix has full rank and thus that it has two non zero eigenvalues. This implies also that the only element on the null space of the Jacobian is zero, and so the only singularity of the vector field V^* is x_0. We know from linear algebra that in this case the eigenvalues of the Jacobian are both real or both complex. When they are real, we have three cases:

1. Both are positive. In this case the singularity is called a *source*.
2. Both are negative. In this case the singularity is called a *sink*.
3. One is positive and the other is negative. In this case the singularity is called a *saddle*.

On the other hand, when the eigenvalues are complex, we have a *center* when the real part of both are zero. Figure 3.1 shows this classification for the singularities of our vector field.

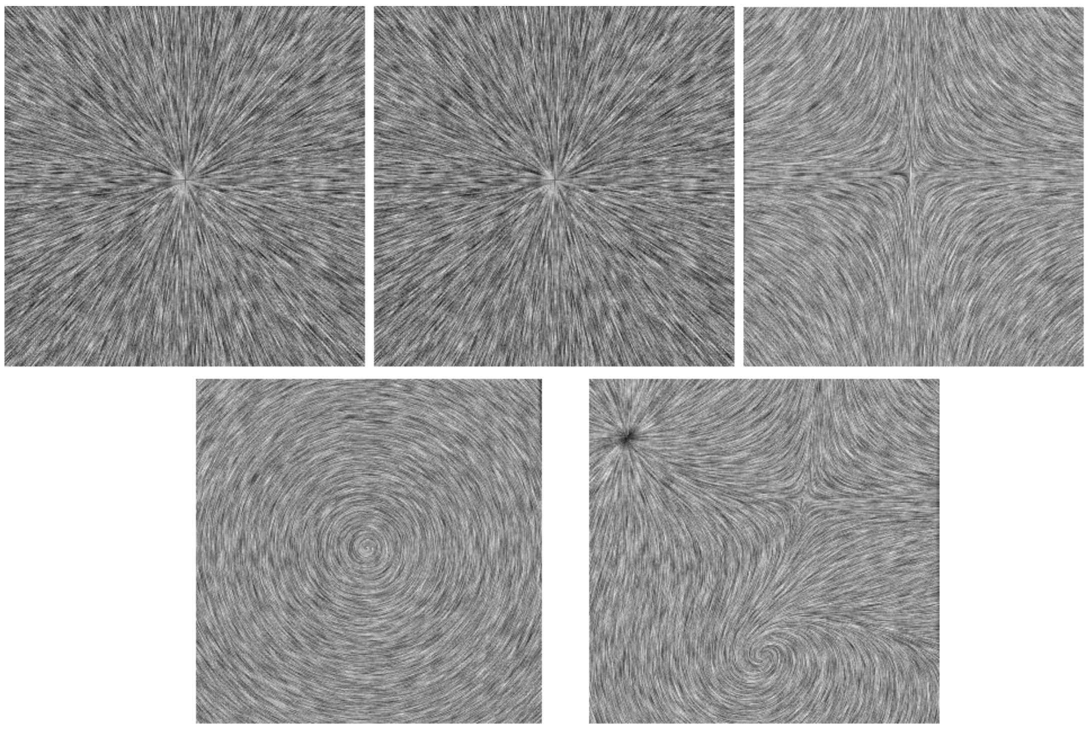

Figure 3.1: Singularities Classification - Top and left to right: A sink, a source and a saddle. Bottom and left to right: A center, and a mix of a saddle a sink and a center.

The basic design consists of providing locations and types of singularities on the image domain. For instance, to create the center at $(0,0)$ shown in figure 3.1, we defined the vector field as

$$V(x,y) = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

In general the type of singularity can be stored as the Jacobian matrix JV, that we can define as:

- $JV = \begin{pmatrix} -k & 0 \\ 0 & -k \end{pmatrix}$ for a sink.
- $JV = \begin{pmatrix} -k & 0 \\ 0 & k \end{pmatrix}$ for a saddle.
- $JV = \begin{pmatrix} k & 0 \\ 0 & k \end{pmatrix}$ for a source.
- $JV = \begin{pmatrix} 0 & -k \\ k & 0 \end{pmatrix}$ for a counter-clockwise center.
- $JV = \begin{pmatrix} 0 & k \\ -k & 0 \end{pmatrix}$ for a clockwise center.

where $k > 0$ is a constant representing the strength of the singularity. In practice, if we want a vector field with a singularity of any type at position $p_0 = (x_0, y_0)$, then one defines the vector field as:

$$V(p) = e^{-d\|p-p_0\|^2} JV \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix}$$

choosing the desired JV and where d is a decay constant that controls the influence of the vector field on points near and far from the singularity. This is essential when one wants to design a vector field which is a composition of many basic fields with singularities. To construct such vector field, we define a simple vector field separately for each singularity, and then we define the final vector field as their sum. For example a vector field with a sink at $q_1 = (10, 10)$ and a center at $q_2 = (-5, 4)$ can be modeled as:

$$V(p) = e^{-d_1\|p-q_1\|^2} \begin{pmatrix} -k_1 & 0 \\ 0 & -k_1 \end{pmatrix} \begin{pmatrix} x - 10 \\ y - 10 \end{pmatrix} + e^{-d_2\|p-q_2\|^2} \begin{pmatrix} 0 & -k_2 \\ k_2 & 0 \end{pmatrix} \begin{pmatrix} x + 5 \\ y - 4 \end{pmatrix}$$

or more compactly:

$$V(p) = V_{q1}(p) + V_{q2}(p)$$

Note that each V_{q_i} has just one singularity, namely q_i. This is not the case for our final vector field, in which new singularities are present when $V_{q_1}(p) = -V_{q_2}(p)$. In particular, choosing each d_i properly, each q_i is a singularity of the final vector field, but it could happen that $V(p) = 0$ for points p where $V_{q_1}(p) \neq 0$ and $V_{q_2}(p) \neq 0$. There is a method to control this undesired new singularities using Conley indices (4) but it falls out of the scope of this monograph.

3.2 Another Approach - Distance Vector Fields

In the previous section we saw how to design planar vector fields classifying its singularities: the user chooses a location and type and a linear vector field its created by choosing some other parameters like strength and influence. In this section we are interested in constructing vector fields using gestures. The idea is to create a vector field that resembles the direction of a given planar curve. We use a distance-based vector field which is explained next.

Given a parametrized curve $C : [a, b] \subset \Re \to \Re^2$ the distance from a point $p \in \Re^2$ to the curve is given by

$$d(p, C) \equiv \min_{t\in[a,b]} \{d(p, C(t))\}$$

The parameter t for which the equality holds in the equation above may be not unique. Indeed, when C is a circumference and p is taken as the center of the circumference, this equation will hold for every value of t. Nevertheless we obtained pleasant results choosing an aleatory value from all the candidates. We will note this value by τ_p. Also we took the Euclidean distance function $d(x, y) = \sqrt{x^2 + y^2}$ for simplicity. To a curve C and a distance function d, we associated two vector fields $V : \Re^2 \to \Re^2$ and $W : \Re^2 \to \Re^2$, each perpendicular to the other by definition:

$$V(p) \equiv C(\tau_p) - p, \quad \langle W(p), V(p)\rangle \equiv 0$$

Assuming that C is differentiable with $C'(t) \neq 0$ for all $t \in [a, b]$, from the definition it is clear that for a point $p \in \Re^2$ with $p \neq C(t)$ for all $t \in [a, b]$, the vector $W(p)$ has the direction (up to sign) of the tangent vector of the curve at τ_p. In fact the function $f : \Re \rightarrow \Re$ defined by

$$f(t) = \langle C(t) - p, C(t) - p \rangle$$

which measures the square of the distance from p to $C(t)$ for each $t \in [a, b]$, has a minimmun at τ_p. On the other hand, we have

$$f'(t) = 2\langle C'(t), C(t) - p \rangle$$

and thus $f'(\tau_p) = 0$ implies that

$$\langle C'(\tau_p), V(p) \rangle = 0$$

leading to $W(p) = \lambda C'(\tau_p)$ for $\lambda \in \Re$ as claimed. From this we see that the vector field W could be a good option to accomplish the design of a vector field that resembles a given curve. However, given that in practice the curve C could be not differentiable (creating it interactively for example), numerous artifacts in the final vector field appear. Figure 3.2 show some examples of V and W for different polylines created with a mouse. Note also how all the points in the curve C became singularities of V and W, which is obvious from their definition.

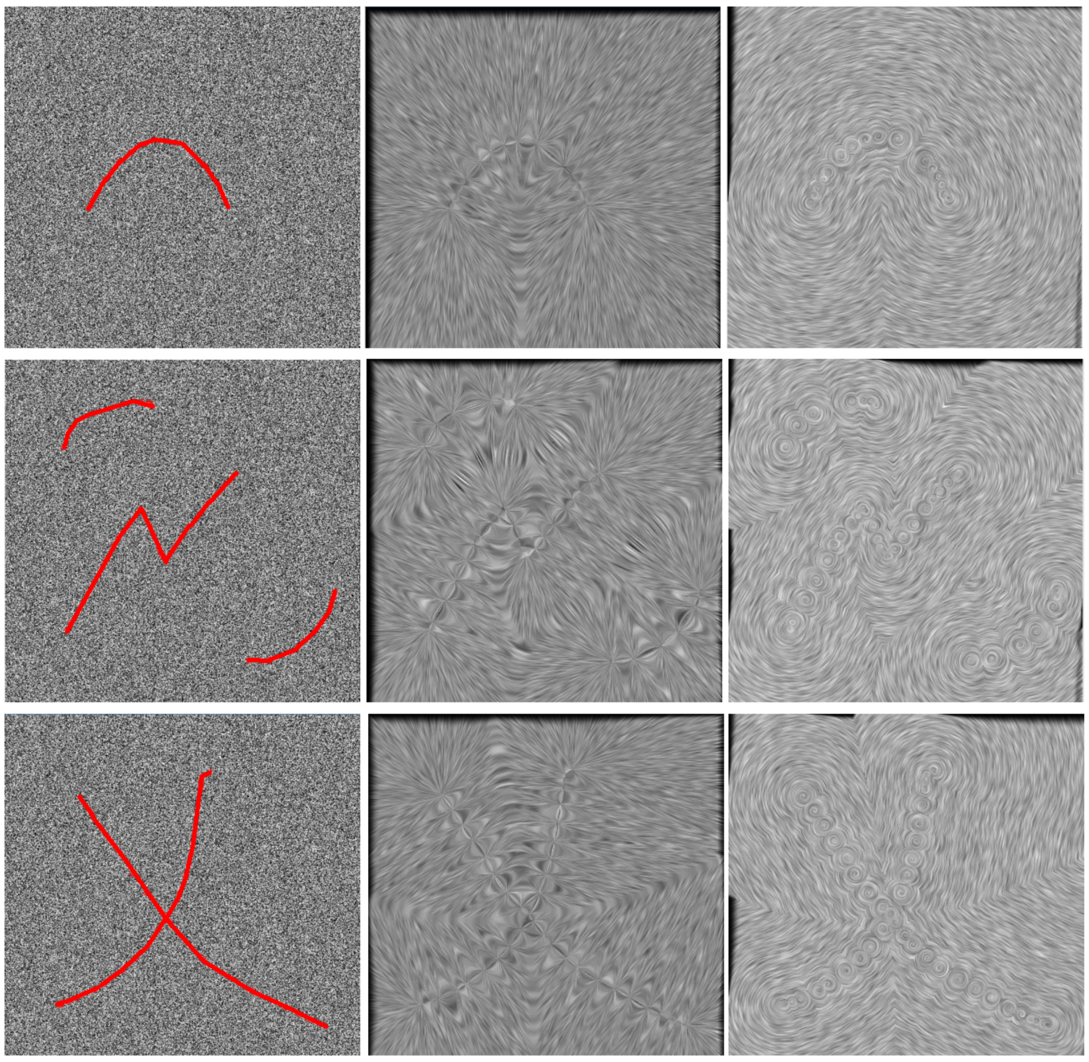

Figure 3.2: Distance Vector Fields - Left to right: White noise with the curve C in red and distance vector fields V and W.

4

Effects Using LIC Ideas

Computational art can be thought as studies concerned in creating and producing pieces of art by means of a computer[1]. In this monograph we are not going to discuss the creative process that leads to a piece of art from the initial white canvas. These subjects, I think, fall into the context of artificial intelligence and cognition and are out of the scope of this text. The creativity involved here will be then of another kind; we will be given an input digital image and we will create and use modifications of the LIC algorithm to process it and generate a new digital image. The result image needs not be compared to any other, since we will be creating rather than imitating styles. Since the results are in some sense non-real, they are called NPR or non-photorealistic rendering effects on images. Nevertheless we will be also including the original real effects blur-warping for completeness.

4.1 Blur

A warping or blur effect can be achieved when using LIC on an arbitrary image rather than on white noise. In this case the vector field will drive the warping effect in its directions. To guarantee visual coherence on the result image, each RGB channel is processed separately. A code in C++ using CImg can be found in the appendix section A.2. Figure 4.1 shows some results.

As you may notice, the deformation on the final image depends strongly on the input image and the vector field used. Also notice that in chapter 2 we were able to

[1]We will be referring just to graphic arts like painting, drawing and photography. This point of view is independent from the definition of art, which we are not going to discuss here.

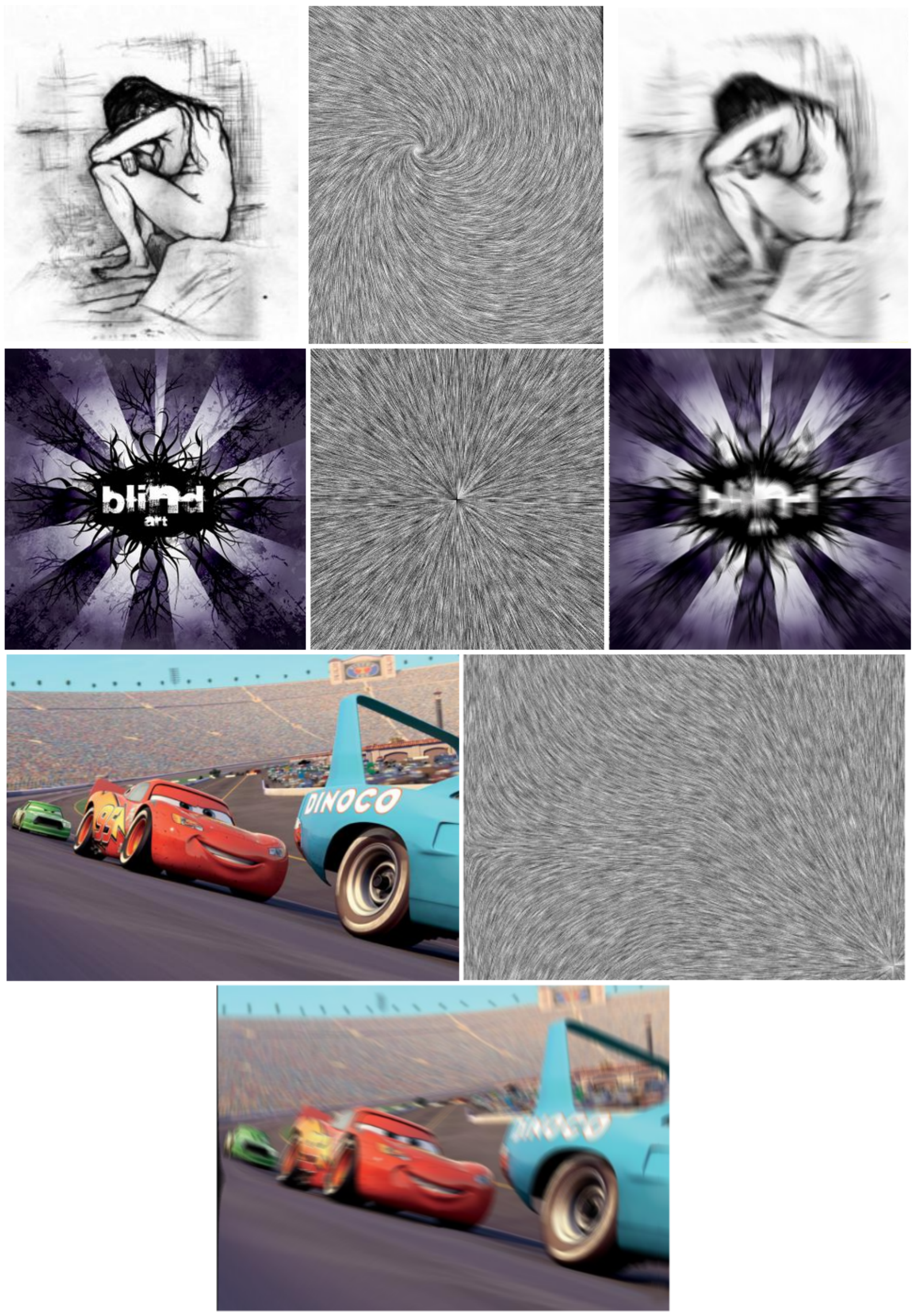

Figure 4.1: Blur Effect - Left-Original Image, Middle-LIC on white noise of the vector field used. Right-LIC of the original image.

visualize the structure of the vector field because of the uniformly distribution of the white noise input image. In general, we compute as a preprocessing step a dithered version of the input image to ensure this charactheristic and then perform the LIC to generate a warping effect. Figure 4.2 shows an example.

Another application of blur-warping is to generate an animation that advects the colors on the image in the direction of the input vector field. This is achieved by iterating LIC several times with the same vector field. This technique could be used to flow visualization not only for steady vector fields but unsteady aswell. However some considerations need to be made to control the color advection at the image boundaries (section 2.4). This black regions can be avoided with a technique called Image Based Flow Visualization (10).

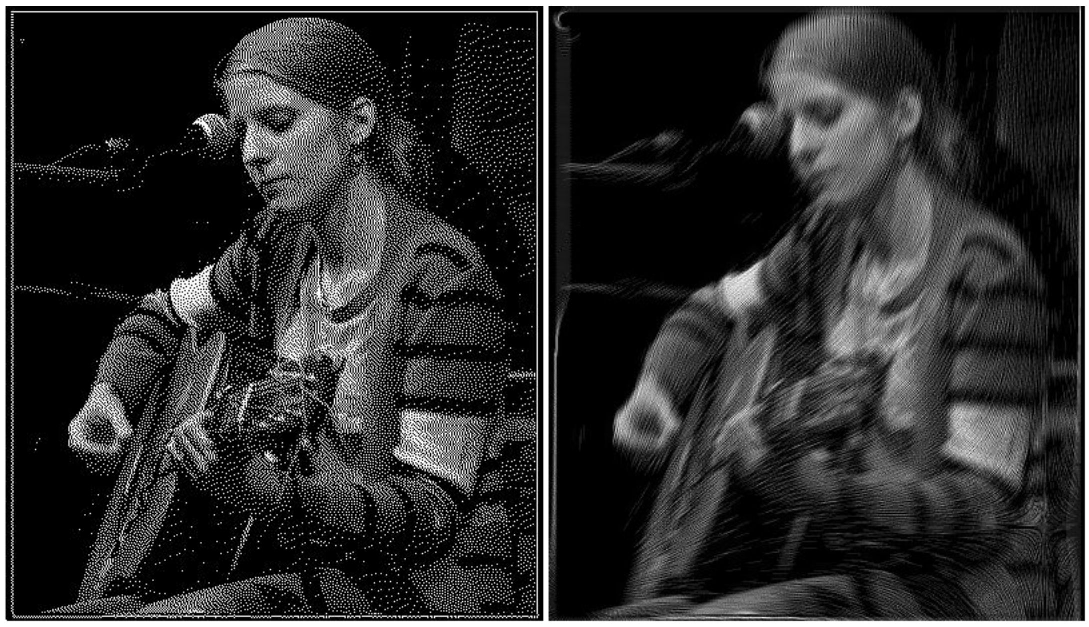

Figure 4.2: Warping a dithered image - From left to right images: dithered, and warping the dithered image

4.2 LIC Silhouette

We can generate a silhouette image automatically with LIC. For this, a threshold is defined to control the value of the convolution of a given pixel. The first step is to convert the image to gray to avoid color incoherences. Then for each pixel we proceed as in LIC, but the output pixel is set depending on the value of the convolution and the thresholds predefined. This process can be used also to generate a dithered version of the visualization of a vector field. Below is a pseudocode of this process. Figure 4.3 show examples of this method.

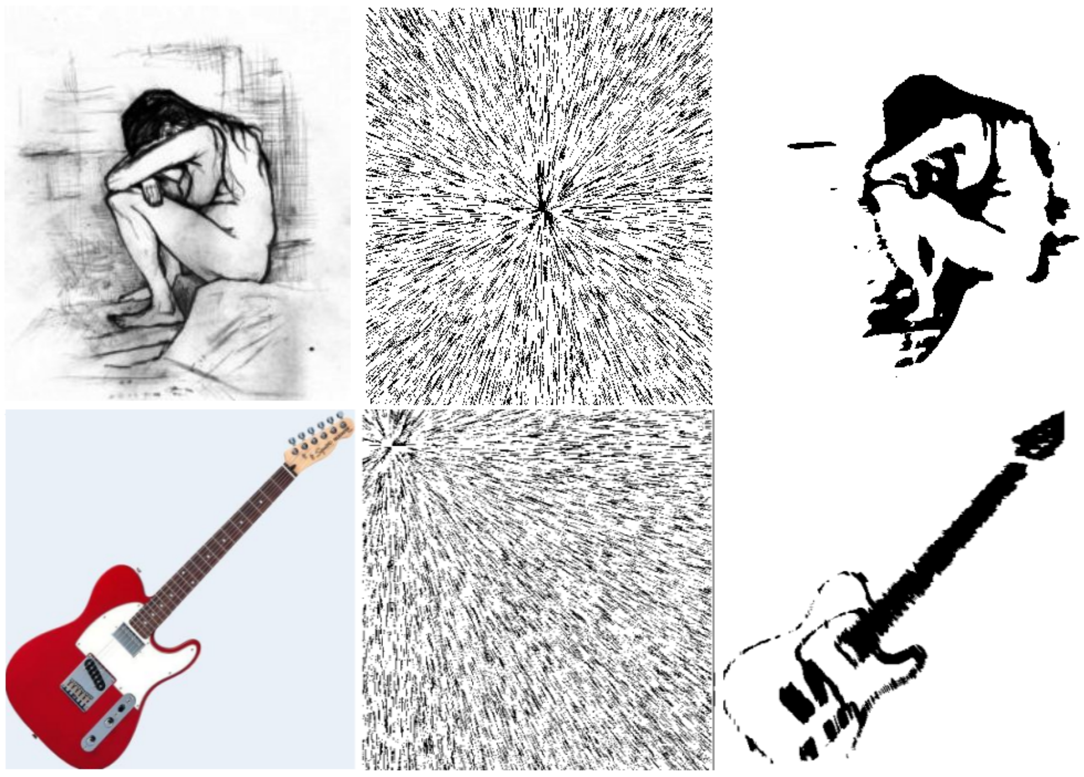

Figure 4.3: Automated Silhouette - Left-Original image, Middle-Dithered LIC, Right-Automated Silhouette using LIC

```
function Silhouette(image){
      convert image to gray(image)
      for each pixel p in image
               C=compute_integral_curve(p)
               I=compute_convolution(image,C)
               if (val 1<I< val 2) { set OutputImage(p)=color 1}
               if (I<val 1){ set OutputImage(p)=color 2}
               else{ set OutputImage(p)=color 3}
 }
```

This approach can be thought as a quantification of the image guided by the values of the line integral convolution. We subdivide the interval $[0, 1]$ in three parts, and we choose an arbitrary color to each part to achieve different effects including the silhouette. We obtained good results setting color i as the color of the first pixel `p` in the original image that belongs to the i part of the subdivision.The thresholds $val1$ and $val2$ can be set arbitrarily. However we found interesting results computing the mean intensity value M of the LIC blurred image and then set $val1 = M - M/3$, $val2 = M + M/3$. Sometimes when the image is too dark we invert colors as a preprocessing step to ensure a good silhouette visualization. Figure 4.4 show some results with this settings.

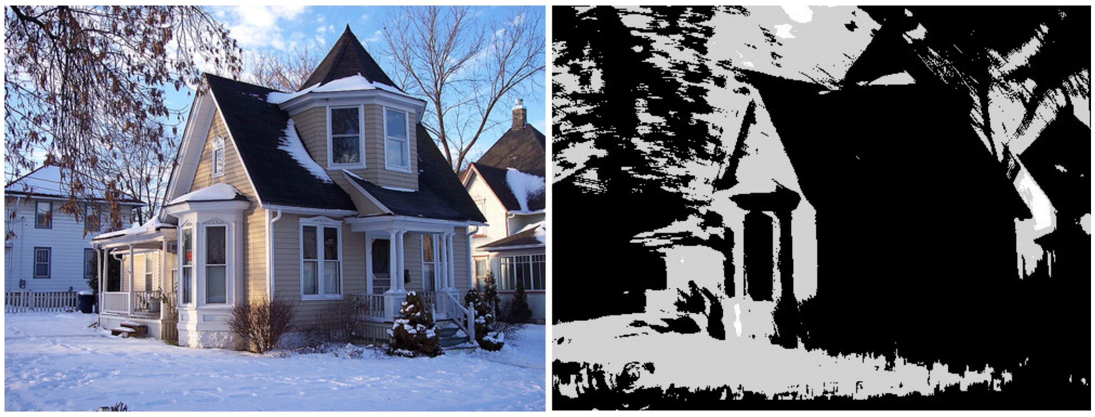

Figure 4.4: LIC Silhouette - Setting color i as the color of the first pixel in the original image belonging to the i part of the $[0, 1]$ subdivision.

It is important to note also the role of the vector field in this approach. Observe that high (low) values of I correspond to convolutions over integral curves on regions with high (low) intensities. Thus for vector fields with integral curves passing from high intensity to low intensity regions we will have an I close to the mean value. That is the reason for edges to appear in the final result for vector fields in directions of discontinuities in the original image.

4.3 Pencil Rendering

We adapted the algorithm described on (8; 9) to create a pencil effect. Below are some results.

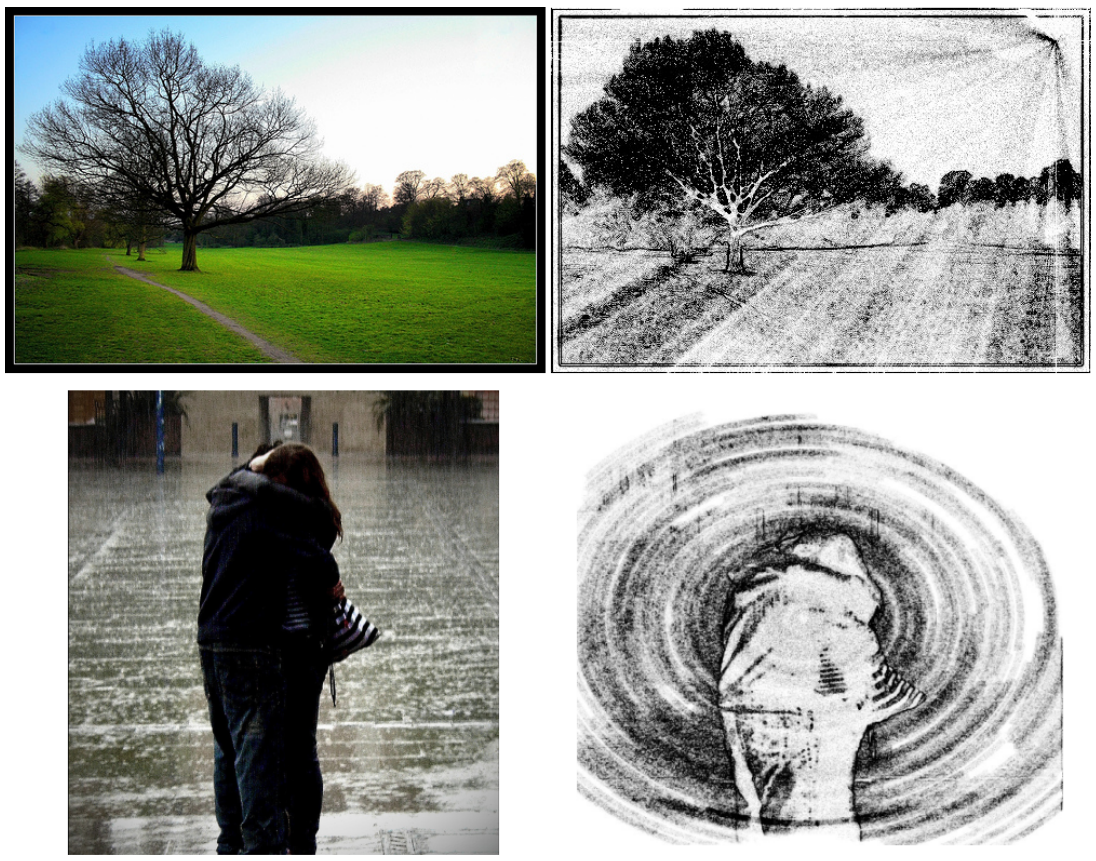

Figure 4.5: LIC Pencil Effect Examples - Some results of the interactive pencil rendering with LIC

Our procedure is set to interactively paint pencil strokes in the direction of an input vector field. Strokes are then integral curves with a fixed length L predefined. As a preprocessing step we compute the gradient image of a grayscale version of the original image as $E(x,y) = |\frac{\partial I(x,y)}{\partial x}| + |\frac{\partial I(x,y)}{\partial y}|$ to perform edge detection. The output energy image is used as the height image and paper is modeled in the same way described in (8). After this when the user clicks on the image, we process a predefined quantity of pixels in the perpendicular direction, rather than this pixel alone. This is done to create strokes with width greater than one pixel, ideal for this kind of effect. Figure 4.7 shows the process step by step. The pseudocode follows:

```
function Interactive_Pencil_rendering(image){
        convert image to gray(image)
        compute the energy image E(gray image)
        create paper with E
        array C=compute_integral_curve(mouseX, mouseY);
        array PP=compute_perpendicular_path(mouseX,mouseY);
        for each location p in C and PP {
            paper_draw(p, presure);
         }
}
```

As in (8), the parameter presure is a value in the interval $[0, 1]$ to model the presure of the pencil in the paper. The `paper_draw` function is guided by the height input image (in this case our energy image) and a sampling function that perturbs locally and uniformly the presure to imitate a hand-made effect.

The method above can be trivially generalized for color images to create an spray-like effect. Here we can choose to process all the colors by pixel or process each RGB channel separately. The latter will create an image with aleatory colors uniformly distributed from the original image. See Figure 4.6.

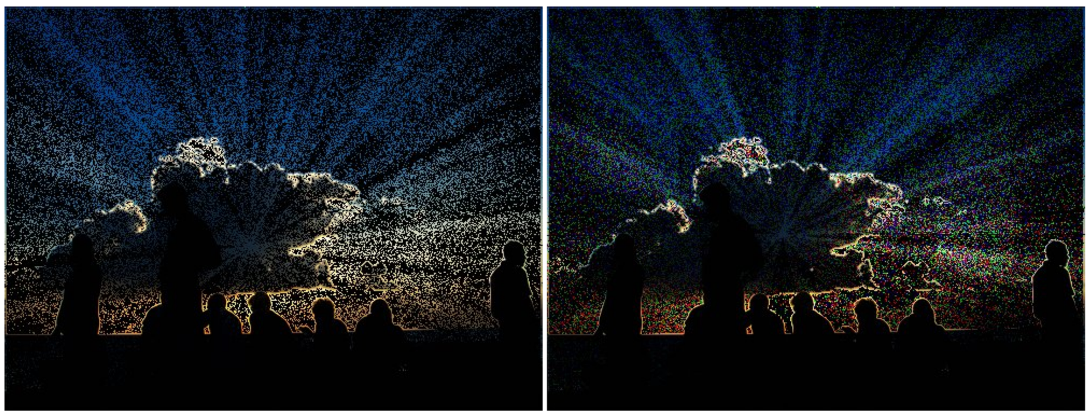

Figure 4.6: LIC Spray-Like Effect - Images from left to right: Processing all colors at once, and processing each RGB channel separately.

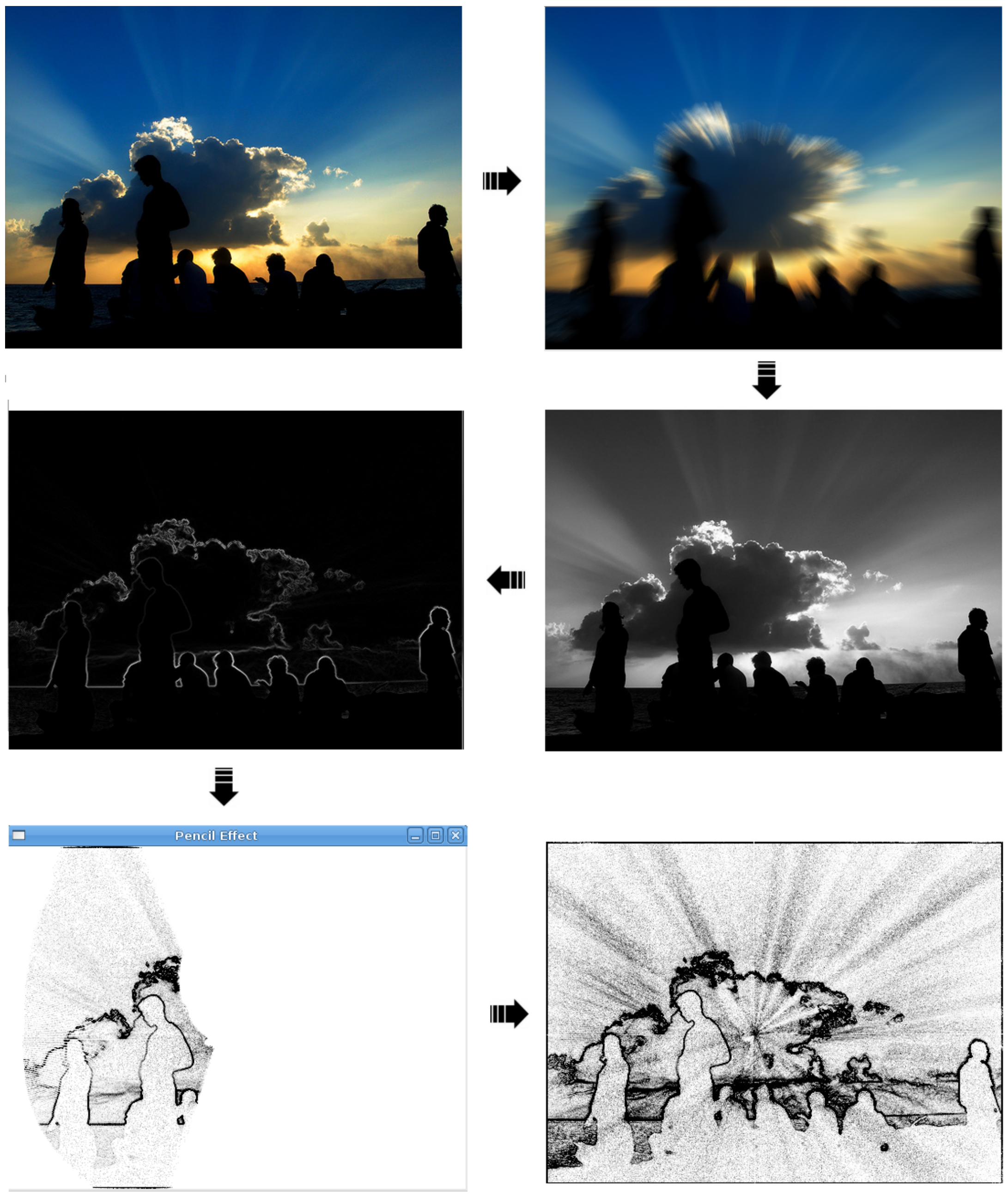

Figure 4.7: LIC Pencil Effect Process - Images: Original, LIC blur, grayscale, energy, interactive pencil rendering and final result.

4.4 Painterly Rendering

To create an image with a hand-painted appearance from an input photograph we used the algorithm described in (7). The method uses curved brush strokes of multiples sizes guided automatically by the contours of the gradient image. We adapted the mentioned algorithm to create strokes in the directions of an input vector field. These strokes are computed as integral curves like in LIC. The stroke length is controlled by the style stroke maximum length like in the original algorithm. A pseudocode the stroke computations procedure follow. Figure 4.8 shows an example.

LIC Strokes Procedure:

```
function makeLICStroke(pixel p, R,reference_img){
        array C=compute_integral_curve(p) with L=style max length
        strokeColor= reference_img.color(p)
        K= new stroke with radius R,
           with locations C,
           and color strokeColor
 }
```

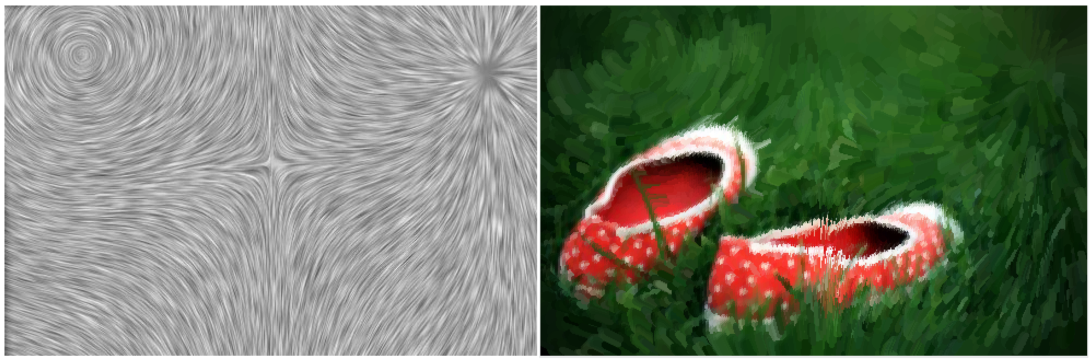

Figure 4.8: Painterly Rendering - Left-Vector Field Image, Right- Painterly rendering.

4.5 Conclusions and Future Work

As we can see from this whole monograph, the Line Integral Convolution algorithm ideas can be used not only to visualize high density planar vector fields, but also to render non-realistic effects on arbitrary images by mixing already know NPR methods like painterly rendering and pencil drawing. The vector field design stage is fundamental in this approach, allowing the user to create a vector field to use as a guide for a determined effect.

There are many ways to continue the work done here by either improving our results or by creating totally new algoritms and experiments. For instance, given that our interactive design is still slow, a GPU implementation for a real-time design will enhance the experimentation process when creating new effects. Given that our painterly rendering algorithm is not interactive, it could be a good experiment to create a system to interactively paint strokes similarly to the pencil and spray effects of section 4.3. Here some considerations with the layers of the original method need to be considered: When the user clicks, is that pixel going to be approximated by which level of detail?. Other future work could be a generalization to 3D spaces. For this, a tensor field design system will be more appropriate as suggested by the literature (3) increasing the flexibility of the whole system and extending the range of visual effects. An interesting next step of our system could be also the use of a Tangible User Interface for the design and visualization of the results.

Out of the topic of this monograph, but still my main interest on flow and vector visualization could be considered as future work. Scientific visualization is a growing area that creates visual representations of complex scientific concepts to improve or discover new understandings from a set of data information. However at this time it is not clear a direct application into the field of computer music. The challenge is to create a new visualization of a piece of music that could gives us an alternative way to understand the basic sound components, or then an artistic visualization that could be used in computer music composition: Given our new visualization of a particular song, can we create another image that has the same characteristics in order to create music from it?. The suggestion is to make a connection between two art components: Graphics and music. Thus, can we use the LIC ideas to create this new visualization?, what information could we retrieve from a song which advects an 1D white noise? and can we visualize this advection?. These are examples of questions that could guide a future work on scientific visualization and computer music.

Appendix A

Implementation in C++

This appendix is included to expose the main algorithms presented all along the chapters making use of the CImg Library, which is an open source C++ image processing toolkit created by David Tschumperlé at INRIA [1]. For a better understanding, we will introduce briefly some of its characteristics before going into our procedures.

A.1 Getting Started with CImg

The CImg Library consist of a single header file *CImg.h* that contains all the C++ classes and methods. This implies among other things, that a simple line of code is needed to use it, namely

```
...
#include "lib_path/CImg.h"
...
using namespace cimg_library;
...
```

given that we had already downloaded the standard package from the website and had placed it into *lib_path*. All the classes and functions are encapsulated in the *cimg_library* namespace, so it is a good idea to use the second line of code too [2]. The main classes of the CImg Library are: `CImg<T>` for images, `CImgList<T>` for a list of

[1]http://cimg.sourceforge.net/

[2]This is different from the *cimg* namespace which implements functions with the same name as standard C/C++ functions! Never use by default the *cimg* namespace.

images, and `CImgDisplay` which is like a canvas to display any image. The template parameter `T` specify the type of the pixels, for example a raster image with entries of type double is defined as `CImg<double>`. Possible values of `T` are `float`, `double` and `unsigned char`. As you may expect displaying an image with *CImg* is as simple as with *MATLAB*. Here is the code to load and display an image called *myimage.jpg* which is in the same directory as our code:

```
#include "lib_path/CImg.h"
using namespace cimg_library;

int main(){
    CImg<unsigned char>("myimage.jpg").display();
    return 0; // needed by the compiler
}
```

To load an image and put it into our code as a variable `I`, we use the code: `CImg<unsigned char> I("myimage.jpg")`. To display an already loaded image `I`, we use its display method: `I.display()`. The code above is a compact version of these two steps: loading and displaying. We also could load an image and display it on a `CImgDisplay`. This is useful when doing applications with interactivity, given that the `CImgDisplay` class allows control to define the user callbacks like mouse clicks and keyboard inputs. The correspondent code using a `CImgDisplay` is next:

```
#include "lib_path/CImg.h"
using namespace cimg_library;

CImgDisplay main_disp;

int main(){
    main_disp.assign( CImg<unsigned char>("myimage.jpg"),"My very first display!!");
    while (!main_disp.is_closed){
       main_disp.wait();
    }
    return 0;
}
```

The `assign` method takes as first argument the image we want to display. We could re-assign any other loaded image at any moment. The second argument will be the title of the window. The `while` loop is necessary to tell the program to wait for user events. Here is where we should put the code to control user events. If this is ommited the program will run normally but will close after displaying the image which ocurs in a small fraction of time, so you will barely see the image!. Each `CImgDisplay` has its own parameters to control the user events which can be retrieved like any other field of the class with a point, some of them are: `.mouse_x` or `.mouse_y` to retrieve the integer coordinates (x, y) of a user click on the display, `.key` to retrieve keyboard inputs and `.button` which is of boolean type to indicate whether or not there was a click on the display. To control the different buttons of the mouse separately we use `.button&1` for the left button and `.button&2` for the right button. For more on this check the CImg documentation.

Once we load an image on a variable, say `I`, we can retrieve the values of its pixel `(x,y,z)` like we were handling a matrix, that is: `I(x,y,z,v)`. The parameter `v` refers to the type of image: `v=1` for gray scale images, and `v=3` for color images. CImg can handle 3D images, in our case for 2D images we will have always `z=1` when declaring a 2D image and `z=0` when retrieving pixel values. Remember that indices on C++ begins at 0, thus to retrieve the RGB components of a 2D image at pixel `(10,10)` we will have: `I(10,10,0,0)` for red, `I(10,10,0,1)` for green and `I(10,10,0,2)` for blue. Finally to retrieve any of the dimensions of I we can call `.dimx()` for the x dimension, `.dimv()` for the v dimension and so on.

As an application the next function convert a color image to a grayscale image:

```
CImg<unsigned char> to_gray(CImg<unsigned char> img){
   if (img.dimv()==1) return img; //already gray
   CImg<unsigned char> gray(img.dimx(),img.dimy(),1,1);
     for (int x=1;x<img.dimx()-1;x++){
         for (int y=1; y<img.dimy()-1;y++){
            gray(x,y,0,0)=.2989*img(x,y,0,0)+
                          .5870*img(x,y,0,1)+
                          .1140*img(x,y,0,2);
          }
      }
   return(gray);
}
```

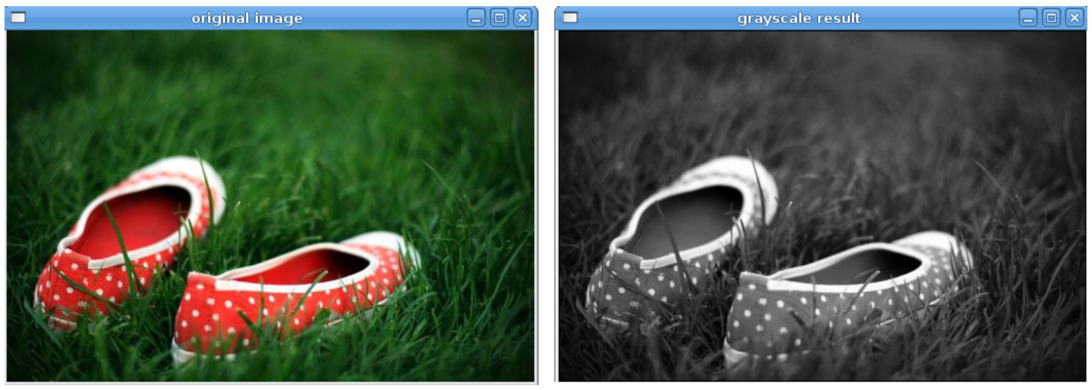

Figure A.1: Color to gray conversion - Converting from color to grayscale using the CImg library.

A.2 LIC in C++ using CImg

Section 2.2 exposed a general pseudocode of LIC with a box kernel. Basically, the algorithm was composed of two functions: computing the integral curve, and computing the convolution. The next code is a mix of these two functions to perform LIC with a box kernel for an arbitrary input image using CImg. Our data type `vector` was defined to store the vector values (vx, vy) of the (x, y) position. The `vector_field` function will be explained in the next section.

LIC-Box kernel with CImg:

```
CImg<double> LIC(CImg<double> img){
 int n_chn=img.dimv(), n=img.dimx(), m=img.dimy();
 CImg<double> OutputImg(n,m,1,n_chn);
 double u, v, Vi, Vj, x, y, sum, ds; int u1,v1,L; ds=1; L=10;
 for (int h=0;h<n_chn; h++){
     for (int i=0;i<n;i++){
         for (int j=0;j<m;j++){
            vector V=vector_field(x,y);
            u=i; v=j; Vi=V.x; Vj=V.y;
            sum=0;
            for (int s=0;s<=L;s++){         //positive calculations
                u=u+ds*V.x; v=v+ds*V.y;
                u1=(int)floor(u); v1=(int)floor(v);
                if (u1<0 || u1>n || v1<0 || v1>m) continue;
                else sum=sum+img(u1,v1,h);
                V=vector_field(u,v);
             }
            u=i; v=j; V.x=Vi; V.y=Vj;
            for (int s=0;s<=-L;s=s-1){         //negative calculations
                u=u-ds*V.x; v=v-ds*V.y;
                u1=(int)floor(u); v1=(int)floor(v);
                if (u1<0 || u1>n || v1<0 || v1>m) continue;
                else sum=sum+img(u1,v1,h);
                V=vector_field(u,v);
             }
            OutputImg(i,j,0,h)=sum/(2*L+1);
         }
      }
  }
  return(OutputImg);
}
```

A.3 A Basic Vector Field Design System using CImg

We can use CImg to create a system that can handle the basic vector field design ideas of section 3.1. For this we use the classes `vector` and `singular_point` to store vector components and singular points with its respectives fields: parameters, type (Jacobian) and position. The code looks like this:

```
typedef struct {double x, y;} vector;

class singular_point {
public:
 vector pos;    //position
 vector W1,W2; //rows of the jacobian matrix
 double d, k;  //parameters
public:
 singular_point(vector poss,vector W11, vector W22, double k1, double d1){
        pos=poss; W1=W11; W2=W22;k=k1;d=d1;
        };
 singular_point(){}; // constructor by default
}; // end of class singular_point
```

We store all the singularities in an simple array `LIST_OF_SING` with a global variable `length` to control its length. The system begins loading an image stored globally as `image` and waiting for user events. A click on the display will add a singularity at this position. The type of the singularity to add is controled by the global variables `V1, V2` which correspond to the rows of the jacobian of the actual singularity. This varibles are initialized for a type *sink* by default, and can be modified with the keyboard: `S` for a sink, `O` for a source, `D` for a saddle, `C` for a clockwise center and `W` for a couter-clockwise center. The main function is next:

```
int main() {
  load_img();
  while (!main_disp.is_closed) {
      main_disp.wait();
      if(main_disp.button && main_disp.mouse_x>=0 && main_disp.mouse_y>=0){
         int u0 = main_disp.mouse_x, v0 = main_disp.mouse_y;
         vector pos; pos.x=u0; pos.y=v0;
         singular_point s(pos,V1,V2,k,d);
         length++; LIST_OF_SING[length-1]=s;
         cout<<"Calculating LIC....\n";
         image=LIC(image);
         cout<<"done.\n";
         image.display(main_disp);
       }
       if (main_disp.key){
         switch (main_disp.key){
           case cimg::keyQ: exit(0); break;
           case cimg::keyS: V1.x=-k; V1.y=0.0; V2.x=0.0; V2.y=-k; break;
           case cimg::keyO: V1.x=k; V1.y=0.0; V2.x=0.0; V2.y=k; break;
           case cimg::keyD: V1.x=k; V1.y=0.0; V2.x=0.0; V2.y=-k; break;
           case cimg::keyC: V1.x=0.0; V1.y=-k; V2.x=k; V2.y=0.0; break;
           case cimg::keyW: V1.x=0.0; V1.y=k; V2.x=-k; V2.y=0.0; break;
         }
       }
   } // end while
    return 0;
 }
```

The additional function `load_img` used to load the image and perform the variables initialization:

```
void load_img(void){
   char name[50];
   cout<<"Please enter the image name (ex: dog.jpg):\n";
   cin>>name;
   CImg<double> im(name);
   image=im;
   main_disp.assign(image,"Basic Vector field Design");
   d=0.0001; k=0.5; // by default
   V1.x=-k; V1.y=0.0; V2.x=0.0; V2.y=-k; // sink by default
   length=0; //no singular points so far
 }
```

Finally, the actual normalized vector field computation for a point (x, y) is done as sums of vector field influences of each singularity, see section 3.1. Here is the code:

```
 vector vector_field(double x, double y) {
   vector OUT; double d,k,x0,y0; vector U1,U2; OUT.x=OUT.y=0.0;
   for (int i=0;i<length;i++){
        double t;
        d=LIST_OF_SING[i].d; k=LIST_OF_SING[i].k;
        U1=LIST_OF_SING[i].W1; U2=LIST_OF_SING[i].W2;
        x0=LIST_OF_SING[i].pos.x; y0=LIST_OF_SING[i].pos.y;
        t=exp(-d*((x-x0)*(x-x0)+(y-y0)*(y-y0)));
        OUT.x=OUT.x+t*(U1.x*k*(x-x0)+U1.y*k*(y-y0));
        OUT.y=OUT.y+t*(U2.x*k*(x-x0)+U2.y*k*(y-y0));
```

```
    }
    double NV=sqrt(OUT.x*OUT.x+OUT.y*OUT.y);
    if (NV!=0){OUT.x=OUT.x/NV; OUT.y=OUT.y/NV;}
    else{OUT.x=OUT.y=0;}
    return OUT;
  }
```

The figure below shows an example of this system.

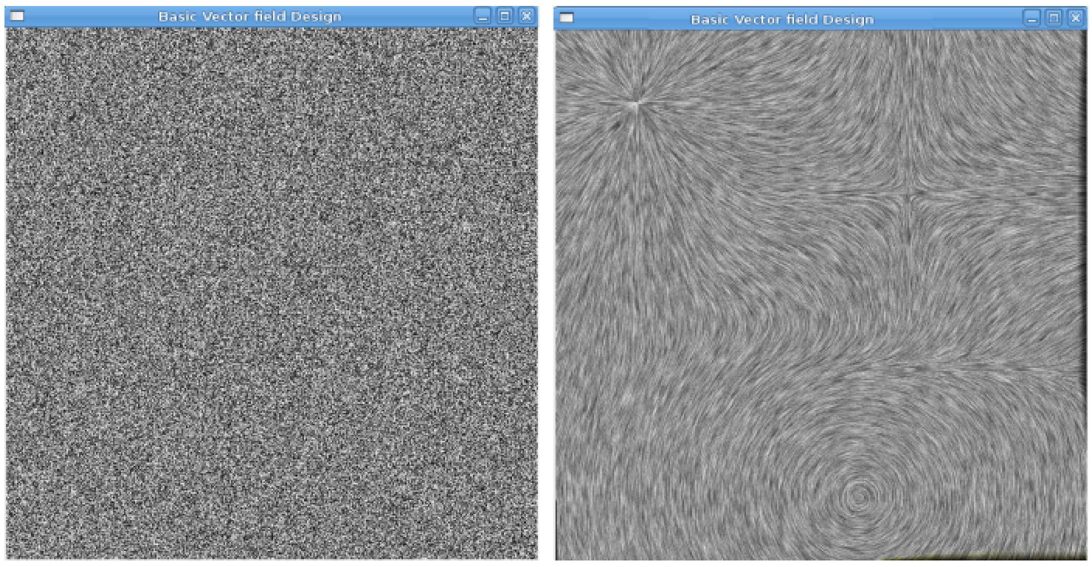

Figure A.2: Basic Vector Field Design example using CImg - A simple combination of a sink a saddle and a center

A.4 LIC effects in C++

This section concludes the implementation in C++ of the LIC effects presented in chapter 4. We already saw the blur-warp effect on section A.2, and the silhouette algorithm is straight forward from that code and the observations of section 4.2. We will proceed then with the pencil effect and painterly rendering.

A.4.1 LIC Pencil Effect

The main class involved in the pencil effect is of course the paper class which stores the height image (in our case the energy image) and the initial white canvas. The drawing function is called interactively with a certain presure perturbed by a sampling function. The final value for a pixel p on the canvas depends linearly on this perturbed presure and the intensity of the pixel p in the height image. For all our results `prs=0.005`:

```
class Paper{
  public:
    CImg<double> height_img;
    CImg<double> canvas;
  public:
    Paper(int resX, int resY, CImg<double> H); //constructor
    void draw(int coordX , int coordY , double prs);
    double sampling(double pres , int res =10);
};

 Paper::Paper(int resX, int resY, CImg<double> H){
  this -> height_img =H;
  CImg<unsigned char> grain(resX,resY,1,1,1);
  this -> canvas=grain;
}

void Paper::draw( int coordX , int coordY , double prs ){
  coordX=(coordX>0)?coordX:0;
  coordY=(coordY>0)?coordY:0;
  double d, h, g;
  h=this->height_img(X,Y,0); h*=0.65;
  g = this->sampling(prs);
  d = h+g;
  canvas(coordX,coordY,0)-=d;
  canvas(coordX,coordY,0)=(canvas(coordX,coordY,0)<0)?
                        0.0:canvas(coordX,coordY,0);
}

double Paper::sampling(double pres, int res){
  int aux = 0 ;
  for (int i = 0 ; i < res ; ++i ){
    double p = (double)std::rand()/(double)(RAND_MAX);
    if ( p < pres ) ++aux;
  }
  return (double)aux/(double)res ;
}
```

Next is the pencil effect main procedure using the above class of paper and the CImg library. For this effect we set the length of the integral curves to $L = 100$ and process $lgd = 10$ pixels in the perpendicular direction (see section 4.3).

```
void pencil_effect(CImg<double> original){
CImg<double> Height=to_gray(original);
Height=invert_colors(Height);
Height=energy(Height);
Height=normalize_0_1(Height);
int n=original.dimx(), m=original.dimy();
Paper paper( n, m, Height,1);
CImgDisplay main_disp(Height,"Pencil Effect");
int const lgd=10;

while (!main_disp.is_closed){
     main_disp.wait();
     if (main_disp.mouse_x>=0 && main_disp.mouse_y>=0){
        int startx = main_disp.mouse_x, starty = main_disp.mouse_y;
        vector V=vector_field(startx,starty,LIST_OF_SING);
        vector Vppd; Vppd.x=-V.y; Vppd.y=V.x;
        for (int i=0;i<lgd;i++){
            double u=startx+i*Vppd.x, v=starty+i*Vppd.y;
            double u0=u,v0=v;
            vector Vstart=V=vector_field(u,v,LIST_OF_SING);
            for (int s=0;s<100;s++){ // positive integral curve
                u=u+V.x; v=v+V.y;
                int u1=(int)floor(u), v1=(int)floor(v);
                u1 = ( u1 < n ) ? u1 : n - 1;
                v1 = ( v1 < m ) ? v1 : m - 1;
                paper.draw( u1 , v1 , 0.005);
                V=vector_field(u,v,LIST_OF_SING);
             }
             u= u0, v=v0; V=Vstart;
```

```
          for (int s=0;s<100;s++){ // negative integral curve
             u=u-V.x; v=v-V.y;
             int u1=(int)floor(u), v1=(int)floor(v);
             u1 = ( u1 < n ) ? u1 : n - 1;
             v1 = ( v1 < m ) ? v1 : m - 1;
             paper.draw( u1 , v1 , 0.005);
             V=vector_field(u,v,LIST_OF_SING);
           }
       }
       paper.height.display(main_disp);
   }
 } // end while
}
```

A.4.2 Painterly Rendering

We implemented the algorithm of section 2.1 in (7) replacing the `makeStroke` procedure with our new `make_LIC_Stroke` to paint strokes in the direction of the input vector field, see section 4.4. Next is the code:

```
stroke make_LIC_Stroke(int x0,int y0,double R){
int n=image.dimx(), m=image.dimy();
int r=floor(ref_img(x0,y0,0,0));
int g=floor(ref_img(x0,y0,0,1));
int b=floor(ref_img(x0,y0,0,2));
int* strokeColor= new int[3];
strokeColor[0]=r; strokeColor[1]=g; strokeColor[2]=b;
stroke K= stroke(R,strokeColor);
point p,q; vector V; double ds=1.0;
p.x=q.x=x0; p.y=q.y=y0;
K.pts[K.lgth]=p; K.lgth++;
vector float_pt, float_qt; float_pt.x=p.x; float_pt.y=p.y;
float_qt.x=p.x; float_qt.y=p.y;
V=vector_field(x0,y0);
  for (int so=0; so<R+sty.maxlgth*0.3;so++){
     float_pt.x+=ds*V.x; float_pt.y+=ds*V.y;
     p.x=(int)float_pt.x; p.y=(int)float_pt.y;
     float_qt.x-=ds*V.x; float_qt.y-=ds*V.y;
     q.x=(int)float_qt.x; q.y=(int)float_qt.y;
     if (!(p.x<0 || p.x>n || p.y<0 || p.y>m)){
         K.pts[K.lgth]=p; K.lgth++;
         V=vector_field(float_pt.x,float_pt.y);
     }
     if (!(q.x<0 || q.x>n || q.y<0 || q.y>m)){
         K.pts[K.lgth]=q; K.lgth++;
         V=vector_field(float_qt.x,float_qt.y);
     }
  } //end for
return K;
}
```

Appendix B

Gallery

Here are some of our results. All the images in full color and the source code can be found in the website *http://w3.impa.br/∼rdcastan/Visualization*.

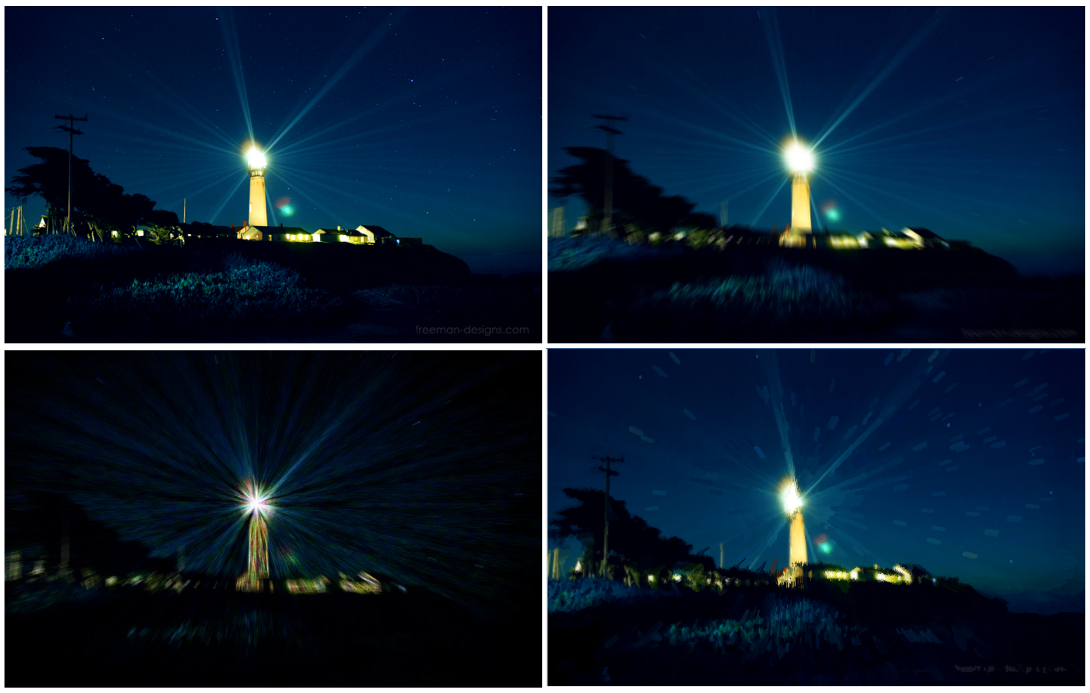

Figure B.1: Pigeon Point Lighthouse, California - Images from top to bottom and left to right: Original, warp, LIC on a spray image with each RGB processed separately and painterly

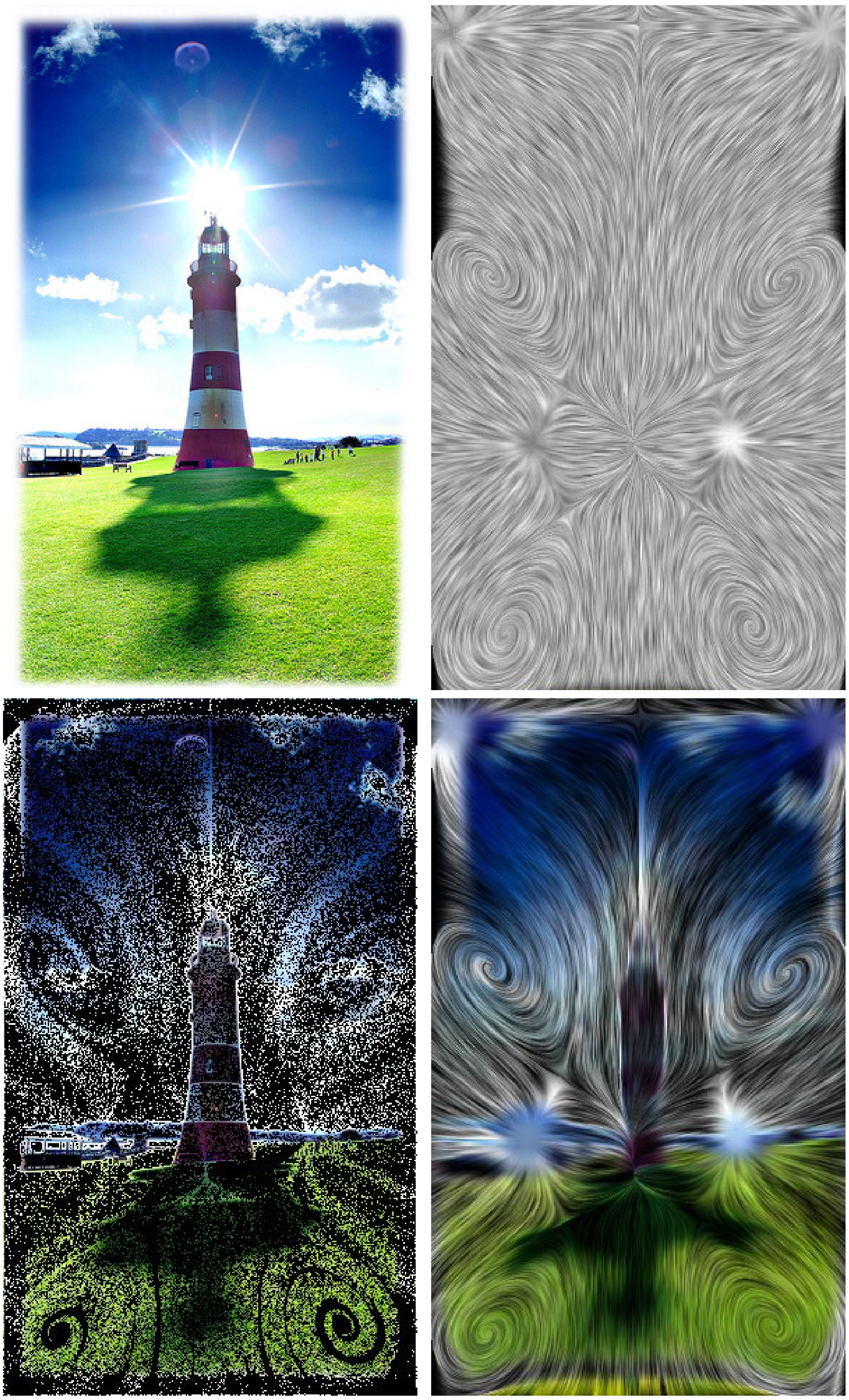

Figure B.2: Plymouth Hoe, England - From top and left to right: Original, vector field visualization, spray and LIC of the spray image.

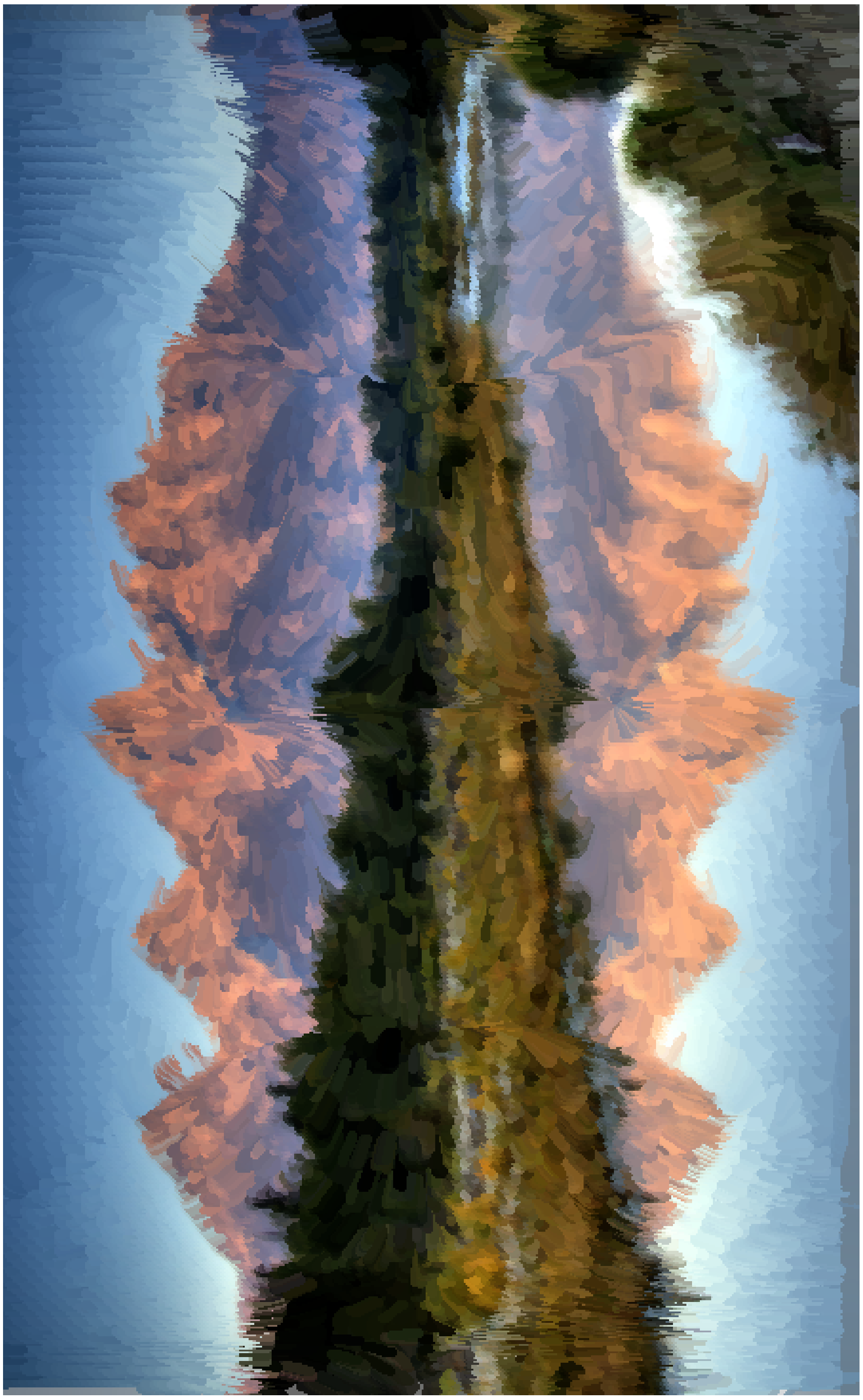

Figure B.3: Landscape - Painterly Rendering.

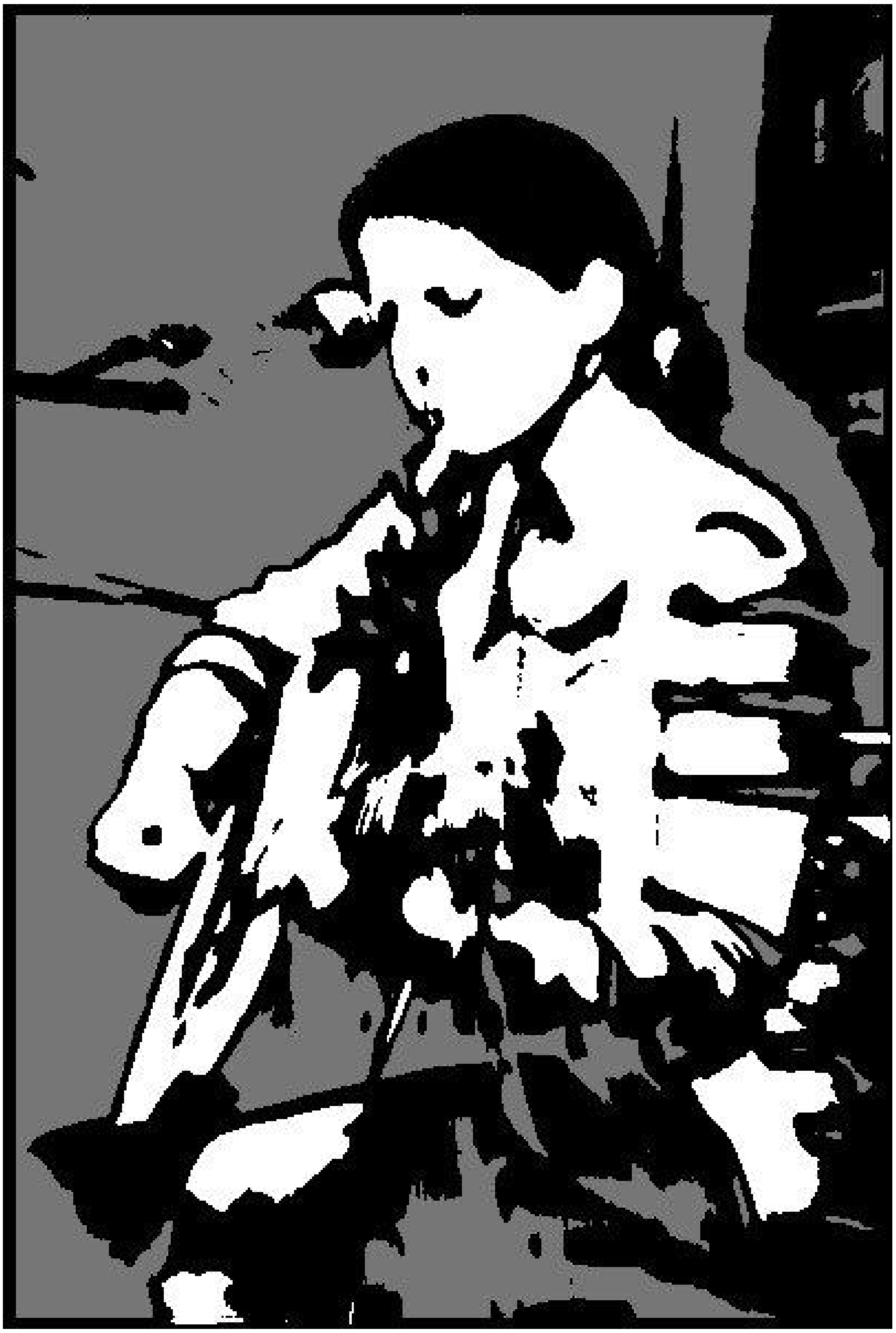

Figure B.4: Girl Playing Guitar - LIC Silhouette effect.

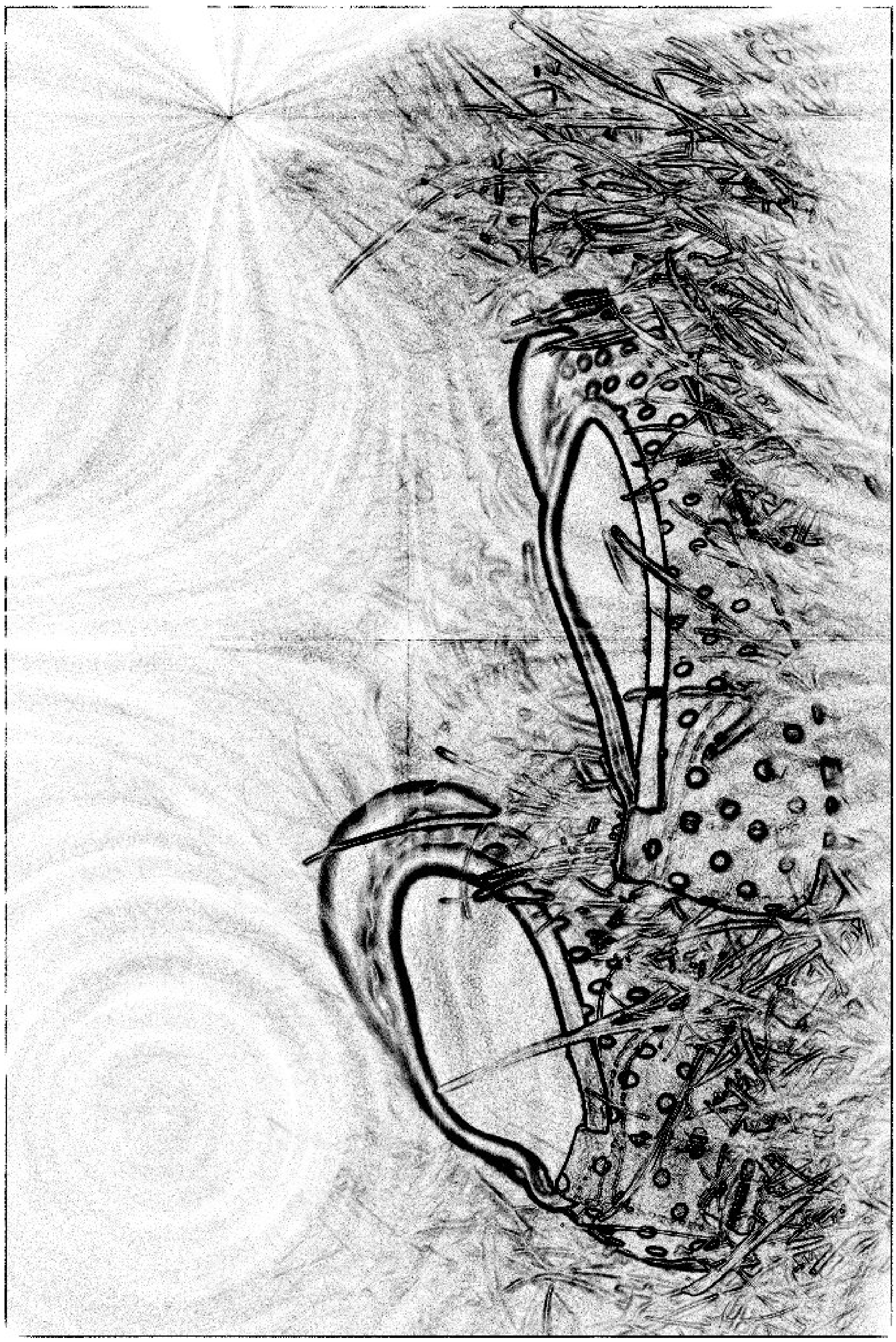

Figure B.5: Shoes in the Grass - LIC pencil effect.

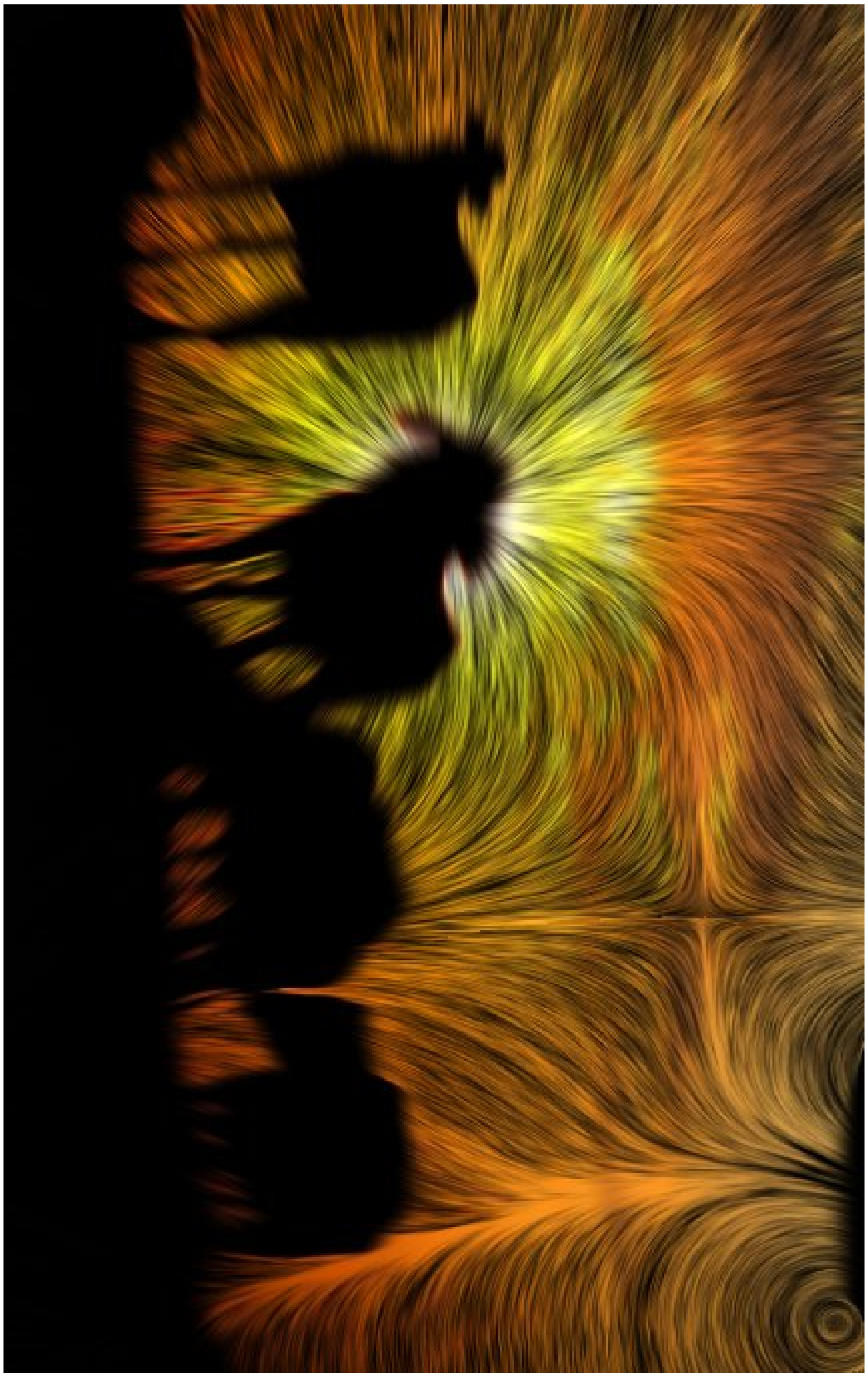

Figure B.6: Cows - LIC after spray effect.

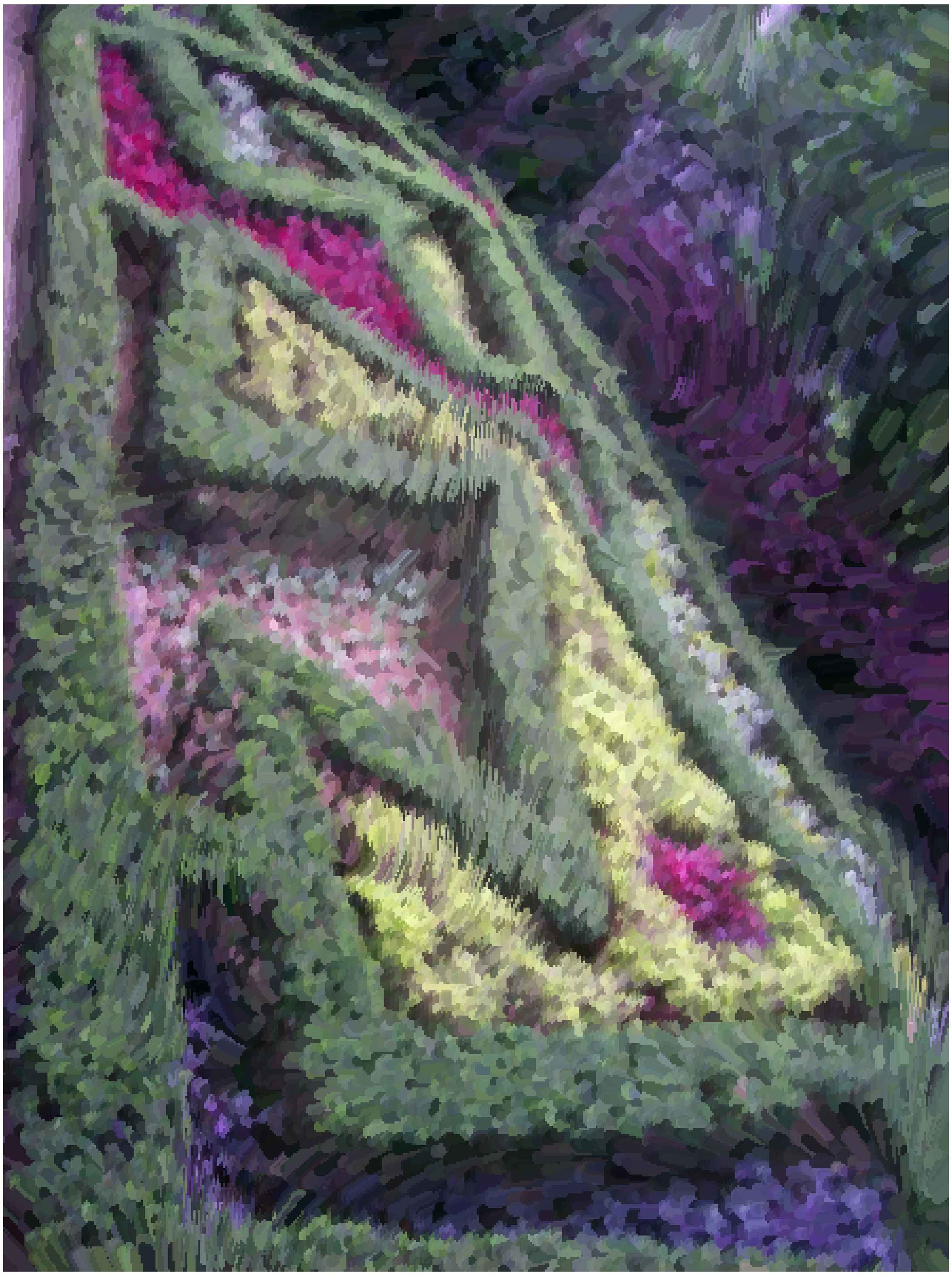

Figure B.7: Garden IMS, Rio de Janeiro - Painterly Rendering. Garden of the Instituto Moreira Salles

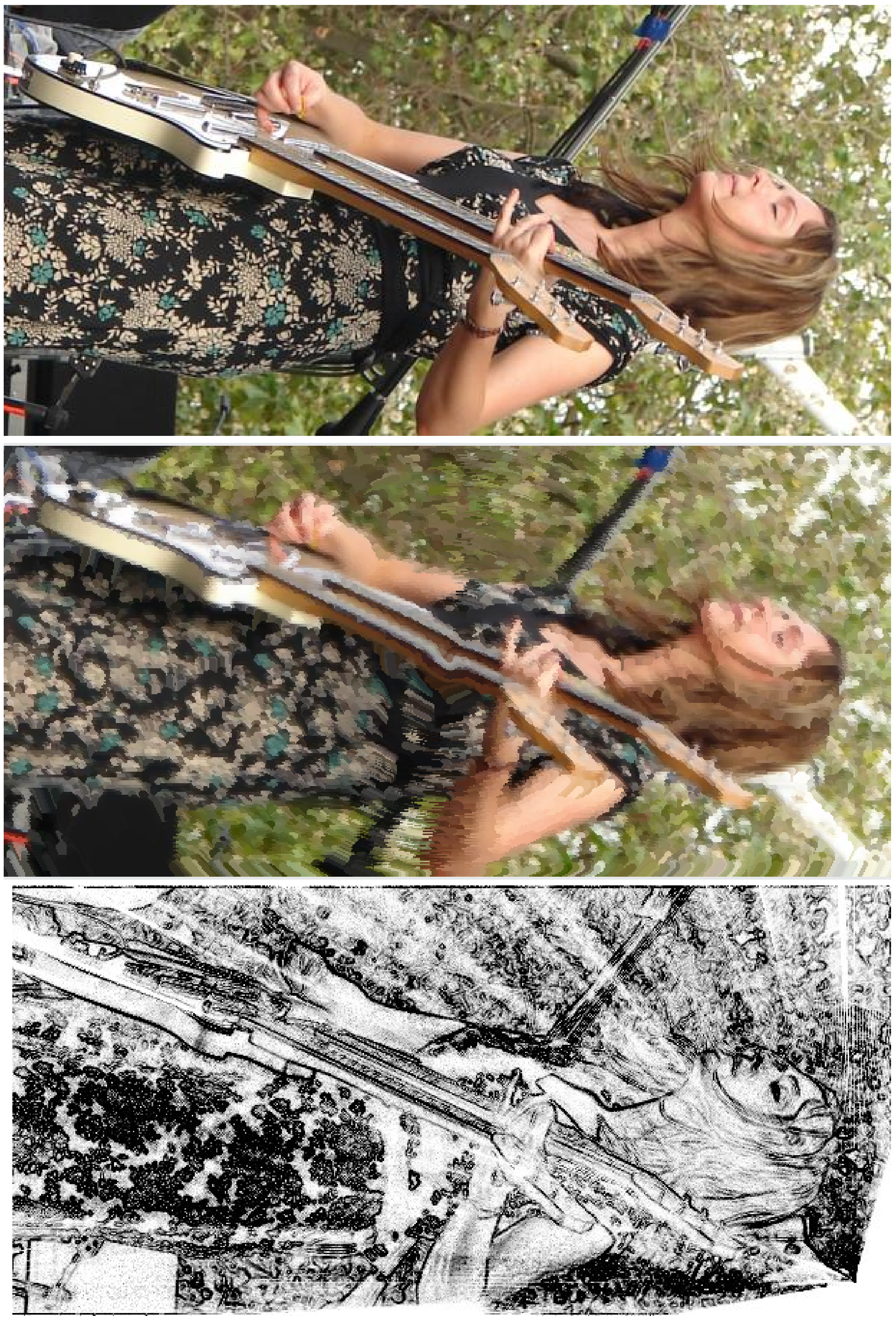

Figure B.8: Live Performance - Top to Bottom-Original (Anita Robinson from Viva Voce), painterly and LIC pencil.

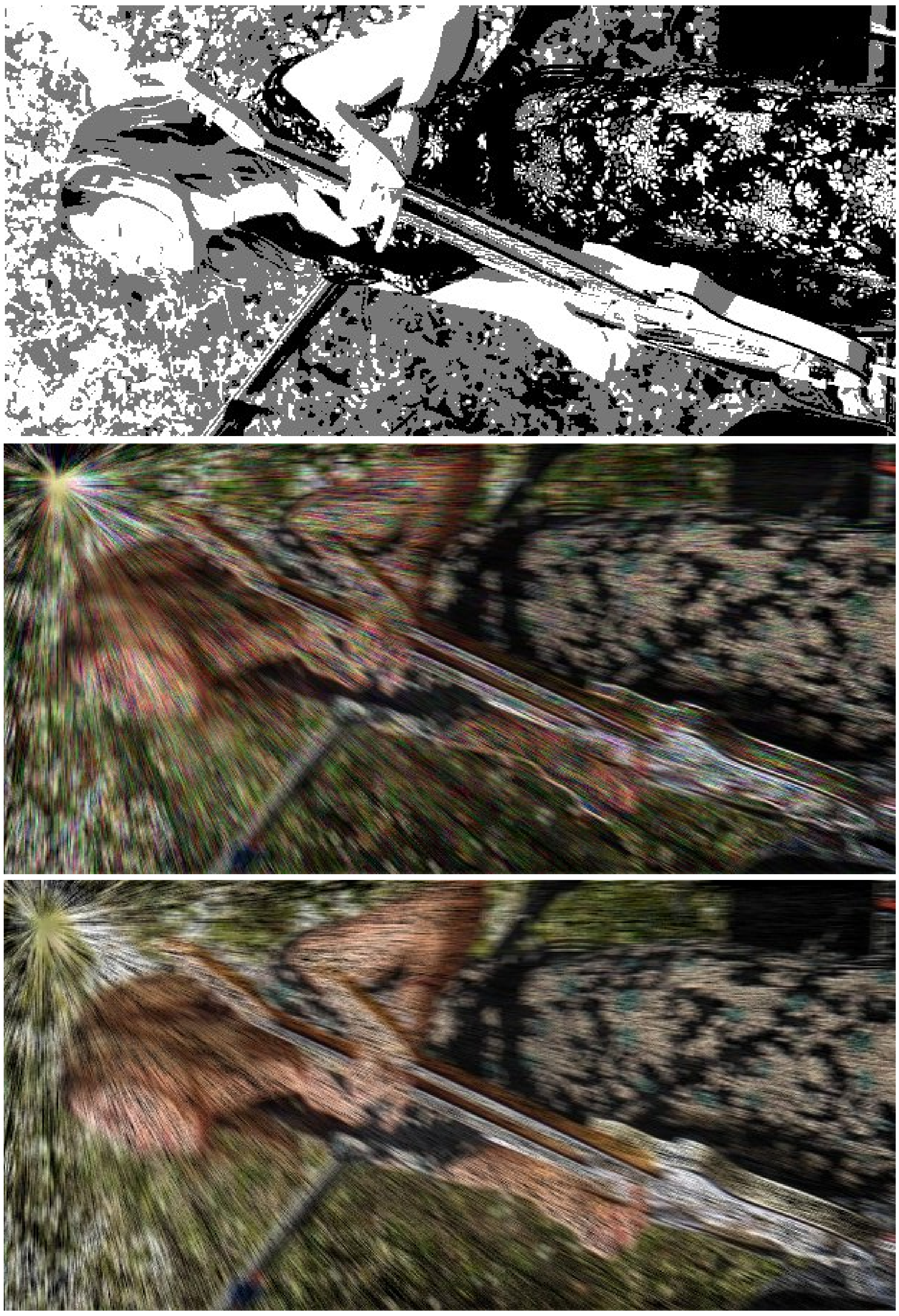

Figure B.9: Live Performance Continued... - Top to Bottom-LIC silhouette, spray on each RGB and simple spray.

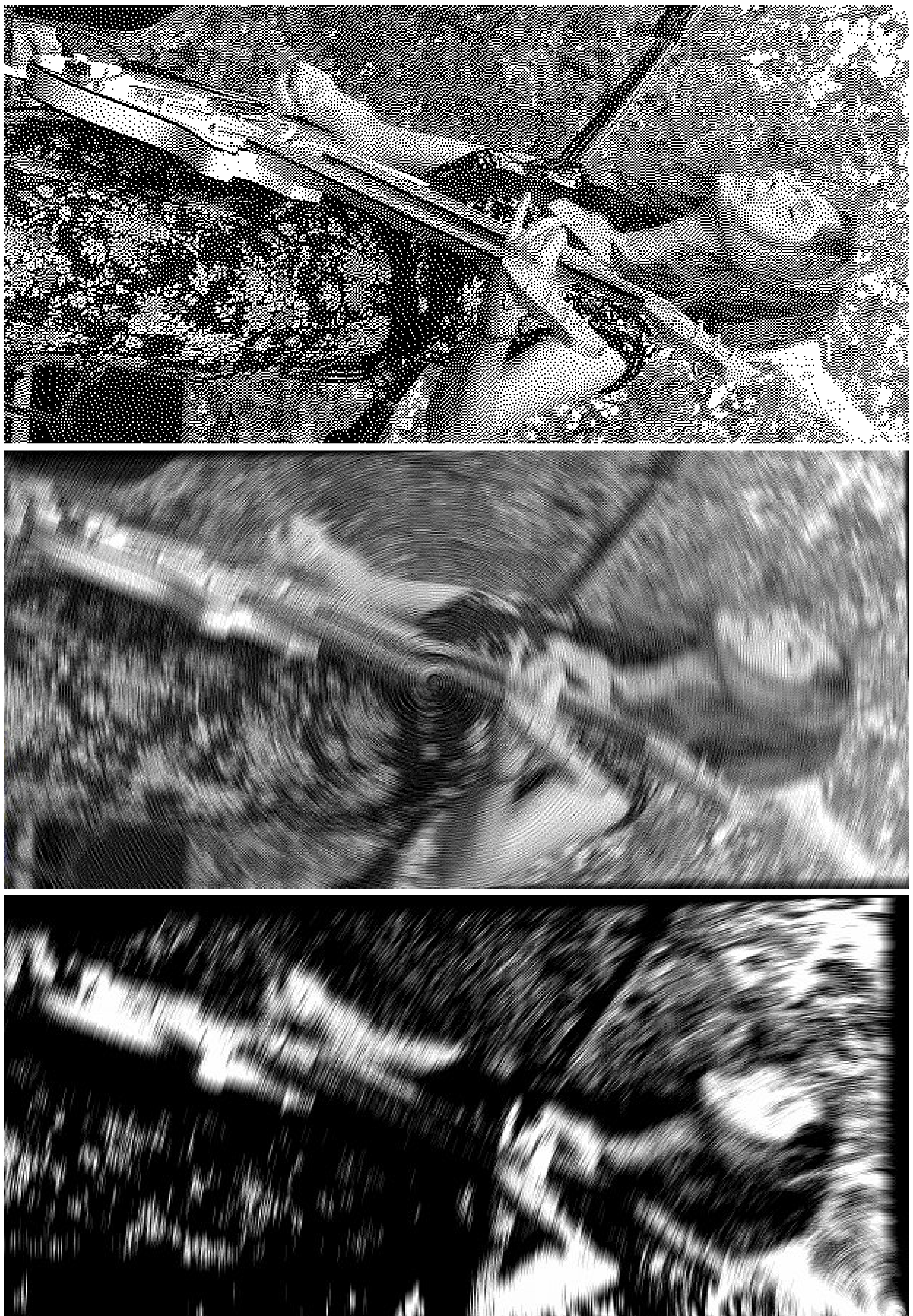

Figure B.10: Live Performance Continued... - Top to Bottom-Dithered and LIC on dithered

References

[1] Brian Cabral, L. C. L. Imaging vector fields using line integral convolution. *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (1993). 1, 3, 5, 6

[2] Carmo, M. P. D. *Differential Geometry of Curves and Surfaces.* IMPA, 1976. 5

[3] Eugene Zhang, James Hays, G. T. Interactive tensor field design and visualization on surfaces. *IEEE Transactions on Visualization and Computer Graphics.* (2006). 28

[4] Eugene Zhang, Konstantin Mischaikow, G. T. Vector field design on surfaces. *Transactions on Graphics* (2006). 16

[5] Hans-Christian Hege, D. S. Fast and resolution independent line integral convolution. *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995). 10

[6] Hans-Christian Hege, D. S. *Mathematical Visualization - Algorithms and Applications.* Springer-Verlag Berlin, 1998. 3, 9, 10

[7] Hertzmann, A. Painterly rendering with curved brush strokes of multiple sizes. *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998). 27, 39

[8] Mario Costa Sousa, J. W. B. Computer-generated graphite pencil rendering of 3d polygonal models. *Computer Graphics Forum* (1999). 24, 25

[9] Mario Costa Sousa, J. W. B. Observational models of graphite pencil materials. *Computer Graphics Forum* (1999). 24

[10] van Wijk, J. J. Image based flow visualization. *Transactions on Graphics* (2002). 21

[11] William H. Press, S. A. T. *Numerical Recipes in C: The Art of Scientific Computing, 2nd Edition*. Cambridge University Press, 1992. 9