Painterly Rendering Using Cellular Automata

Siniša Petrić

***Abstract*—This paper presents further development and new results in artistic rendering using cellular automata concept. New rules and options are introduced in order to obtain variety of non-photorealistic painterly effects on digital photography: moment based stroke placement, attractor points, external gray-scale bitmap tracking, etc. All examples in this paper are created by "Image Infector v1.2" program, which is a plugin for Pixopedia 24 image editor and painter (www.sigmapi-design.com).**

***Index Terms*—cellular, automata, image, painterly, rendering, effects, artistic, growth, color, infection, automaton, NPR**

I. Introduction

IN my previous paper "Image Effects with Cellular Automata", I have introduced cellular automaton as a tool for achieving various artistic effects on digital photography. This paper brings some new results and experimentation made in this field. Main algorithm has changed, new rules have been added, as well as new seeding and infection color options. Some parts from previous article will be revisited and discussed again in order to keep this paper more consistent, without too much references to previous version of "Image Infector".

II. Seeding

SEDING mechanism describes the way how the initial cells are distributed over grayscale mask.
Besides various types of seeding styles, there are parameters used for additional seeding control. here is the list of seeding styles:

1) Random seeds: the cells are distributed randomly across the mask .

2) Grid seeds: equidistant cells seeding. All cells are separated from each other by fixed distance (if no dithering is used).

3) Double seed high intensity areas: original image is converted to BW image using threshold value of 128 (0x80) and seeding is performed in two passed. On the first pass, mask is seeded uniformly and on the second pass, only areas with values greater or equal to 128 are seeded again.

4) Double seed low intensity areas: same as above, but with second pass seeding values below 128.

5) Seed with dithered original image (Bayer): original image is converted to grayscale and Bayer dithering algorithm is applied with level=2 and noise=0.

6) Seed with dithered original image (Jarvis-Judice-Ninke): same as above, with different dithering algorithm.

7) Contour seeding: if some 8-bit grayscale contour of original image is created (option inside Pixopedia 24), contour pixels, which usually have value 0xff are modified with selected automaton rule values. Contour mask is used as a container mask.

8) High-Pass 3x3 seeding: original image is converted to grayscale image. High-pass 3x3 filter is applied on the grayscale image with ordered threshold. Resultant image is saved is saved as container mask.

Parameters that give user additional control over seeding process are:

1) *SeedWindow*: used to calculate the number of seeds (*SeedNum*). *SeedNum* is calculated using formula:

$$SeedNum = \frac{ImageWidth * ImageHieght}{SeedWindow^2}$$

2) Dither grid seeding: this option gives possibility to dither seeding points inside the grid. Uniformly distributed seeds are dithered inside the grid cell to give some more fuzziness. Seeding points are sill inside the grid cell, but are randomly displaced from center point.

3) Reduce seed density: reduces seed density for Bayer, Jarvis-Judice-Ninke, contour and High-Pass seeding styles. As those styles produce very high density seeding, this option is very useful, as it reduces seed density in order to give some space for cell infection.

4) Fixed contour: if set to true, set values on image contour to 0xff in order to preserve image edges (contour clipping).

III. Algorithm

ALGORITHM described in my previous article, has gone through some changes. Instead of 3 images (original image and two grayscale masks *LM* and *RM*) there is another 24-bit depth image, which is a copy of original image at the start. As Pixopedia 24 always works with two copies of original image and two grayscale masks, Image Infector does not create any additional images as they are passed from Pixopedia 24 to plug-in. Anyway, for the simplicity of algorithm presentation, process of creating images is presented in the first two steps.

One color image is used as output, while another is

used as input. We will denote these images as s $OutImg$ and s $InpImg$ respectively. Value of pixels at given position is denoted as $IMAGE\,(i,j)$, where *IMAGE* can be either *OutImg* or *InpImg*.

Another difference in algorithm is that branching occurs regardless of slot state. However, branching is controlled by additional parameter $BranchProbability$. No matter if slot is occupied or not, cell will try to branch itself $BranchNum$ times with probability $p = BranchProbability/100$. The new algorithm goes like this:

1) Copy original image *InpImg* to *OutImg* (clone image).
2) Create two 8-bit masks of the same size as original image: *LM* and *RM*.
3) Populate *LM* with initial cells according to seeding method and set seed values according to automaton rule selected.
4) Copy *LM* to *RM*.
5) Scan through *LM* (*i = 0 to ImageHeight, j = 0 to ImageWidth*)
6) If $(LM\,(i,j) > 0x00)$ AND $(LM\,(i,j) \leq LastRuleValue)$ find free slot according to $LM\,(i,j)$ value (check *W* array vs. *R* array). Get free slot position $\left(i^f, j^f\right)$.
7) Set $InpImg\left(i^f, j^f\right) = InpImg\,(i,j))$. Set $OutImg\left(i^f, j^f\right) = F(InpImg_{i,j}, OutImg(i.j))$. Set $LM\,(i,j) = 0xff$ and set $RM\left(i^f, j^f\right)$ to new mask value.
8) According to $BranchProbability$, branch cell $BranchNum$ times.
9) Copy *RM* to *LM*.
10) Repeat steps from 5. to 9. until maximum number of iterations (*MaxIter*) is reached, or all cells are set to fixed (*0xff*).

Branching probability parameter gives some additional fuzziness in the system. Furthermore, it's obvious that less branching implies more iterations. This fact is of crucial importance, as some rules "ask" for less branching occurrence, while others expect more branching in order to achieve desired painterly effects.

Function *F* in step 7. changes pixel value according to infection color option selected and will be explained in separate section.

The reason for adding another image is mainly because of some inconsistency regarding infection color functions, especially with convolution option and new intensity changing options. As you can see from algorithm described here, *InpImage* now contains pixels values spread thorough automaton rules without any color modifications. On the other hand *OutImage* contains pixels modified by infection color function.

IV. Automaton rules

RULES are the crucial part of Image Infector. In previous article and in previous version of "Image Infector" there were three groups of rules: Preferred direction, Strict direction and Calculated directions. First two rules are kept the same in new version. The third group has evolved with some new rules, while some rules have only changed the names. In this section, I will only give a brief explanation of rules and those rules that need more detailed explanation will be covered in separate section.
Regardless of rule chosen, infection mechanism, generally works by scanning container mask and looking for a non-fixed cell.

When such cell is found and focused, cell propagation will take place in focused cell's neighborhood which can be visualized by 3x3 matrix:

$$N = \begin{bmatrix} NW & N & NE \\ W & X & E \\ SW & S & SE \end{bmatrix}$$

X is focused cell value, while surrounding cells are denoted according to their position from focused cell (northwest, north, northeast, etc...). Inside Image Infector, this wind rose matrix is implemented as a one row matrix with omitted middle element *X*, so that position of the next infected cell can be easily calculated:

$$W = [NW, N, NE, W, E, SW, S, SE]$$

Wind rose matrix (array) W, depending on the rule selected, can correspond to single byte value, or to similar one row matrix (array). For instance, if preferred direction rule is chosen, wind rose array corresponds to:

$$R = [0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08]$$

single row byte matrix.

Choosing direction for next cell depends on the selected rule. If chosen new cell is occupied, automaton will try to find alternative free cells and will eventually perform branching. If no free slot is found, automaton proceeds to next scanning step. Here are the rules available, grouped in three sections:

1) Preferred direction: X takes its value from array $[0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08]$.
   Focused cell value will be used to calculate direction of propagation. It means that cell with value 0x01 will try to occupy nearest slot in the northeast direction, if value is 0x02 it will try to infect nearest free slot in the north, etc... If neighbor pixel in this direction is occupied, first free slot will be occupied and the new

direction will be set (according to free slot position).

Example: Let's assume that $BranchNum > 0$. While scanning container mask we have found the pixel with value X=0x02 and surrounding neighborhood is:

$$N = \begin{bmatrix} 0x01 & 0xff & 0x11 \\ 0x03 & X & 0x00 \\ 0x00 & 0x00 & 0x00 \end{bmatrix}$$

This neighbor matrix corresponds to array:

$$W = [0x01, 0xff, 0x11, 0x03, 0x00, 0x00, 0x00, 0x00]$$

and our rule array is:

$$R = [0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08]$$

As $X = 0x02$, which is $R(2)$, automaton look at $W(2)$value, but $W(2) = 0xff$, which is fixed cell and this place can't be infected. Automaton will search for the first free place int he array, which is east. The resultant matrix is

$$N = \begin{bmatrix} 0x01 & 0xff & 0x11 \\ 0x03 & 0xff & 0x05 \\ 0x00 & 0x00 & 0x00 \end{bmatrix}$$

Automaton will repeat infection until repeat number reaches $BranchNum$. You'll notice, that focused cell value is fixed to 0xff and east neighbor is set to corresponding byte value from R.

2) Strict direction: *X* takes its value from array:

$$R = [0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10]$$

and it works the similar way as preferred direction, but initial direction is carried to the free slot, id est: automaton tends to keep initial (strict) direction.

Example: Let's assume that $BranchNum > 0$. While scanning container mask we have found the pixel with value $X = 0x0c$ and surrounding neighborhood is

$$N = \begin{bmatrix} 0x01 & 0xff & 0x11 \\ 0x03 & X & 0x00 \\ 0x00 & 0x00 & 0x00 \end{bmatrix}$$

This neighbor matrix corresponds to array

$$W = [0x01, 0xff, 0x11, 0x03, 0x00, 0x00, 0x00, 0x00]$$

and our rule array is:

$$R = [0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10]$$

As $X = 0x0c$, which is $R(4)$ automaton look at $W(4)$, but $W(4) = 0x03$ and can not be infected. Automaton will search for the first free place, which is east. This time, free slot is occupied with initial X value in order to force initial direction. The resultant matrix is

$$N = \begin{bmatrix} 0x01 & 0xff & 0x11 \\ 0x03 & 0xff & 0x0c \\ 0x00 & 0x00 & 0x00 \end{bmatrix}$$

Automaton will repeat infection until repeat number reaches $BranchNum$.

3) Calculated direction: this group of rules are not represented by array corresponding to wind rose matrix, but rather by a scalar value:

   a) Random: this rule is represented with byte value 0x11 and it means that all directions from the wind rose are equally possible, direction is randomly chosen. If this place is not occupied, value 0x11 is distributed to the new cell position. Old cell position is set to "fixed value".

   b) EatLow: rule is represented with byte value 0x12. This value tells automaton to search original image neighboring pixel with the lowest intensity value (regarding the focused cell position). If such value is found automaton will populate container mask with the same vale 0x12. In previous program version this rule was mentioned under the name "High". The new name better describes actual cell movement.
   c) EatHigh: similar as above. Value is 0x13 and tells automaton to search for highest intensity value neighboring cell (around the current focused cell position). In previous program version this rule was mentioned under the name "Low". The new name better describes actual cell movement.
   d) Closest: byte value 0x14. This rule tells automaton to search for closest color value among original image neighboring pixels. This rule will be discussed in more details in separate section.
   e) Implode: byte value 0x15. Forces cells to grow towards the center of image.
   f) Explode: byte value 0x16. Forces cells grow from the center of image. Both Implode and Explode rules will be covered in more detail under Attractors section.
   g) Attract: byte value 0x19. Forces cells to grow toward closest attractor point in the image.
   h) Repel: byte value 0x1a. Forces cells to grow away from closest attractor point in the image. Both Attract and Rep ell rules will be covered under Attractors section.
   i) TrackB: represented by byte value 0x17, forces cells growth toward the closest intensity value of tiled grayscale background image obtained from Pixopedia 24. Details will be covered under separate section.
   j) TrackG: byte value 0x19. Forces cells growth toward the closest intensity value of grayscaled representation of the original image. Details will be covers under separate section.
   k) MBSP: moment based stroke placement. Represented by byte value 0x18, forces cells growth along rectangle strokes created by calculating image moments and placed on the special grayscale

mask. This rule really needs detailed explanation and will be covered under section Moment Based Stroke Placement.

Described rules are not in exclusive relations. Values from selected rules are added to "seed sack". Values from seed sack are randomly picked in seeding process. Additional parameters for Closest, MBSP, Attract and Repel rules will be covered in sections regarding rules mentioned.

V. Closest Color Rule

DIRECTION of cells growth covered with this rule is calculated by selecting the smallest Euclidean distance (in RGB color space) between InpImg color at position i,j and its neighboring pixels.

1) Collect $InpImg\,(i,j)$ RGB channels in variables: *r, g, b* .
2) Scan through the neighborhood (*k = 1,..,8*) , for pixel at given neighborhood position *k* , collect RGB channels r_k, g_k, b_k. If position *k* is occupied slot, continue to next position and find color distance between central and neighboring pixel: $dist = \sqrt{(r-r_k)^2+(g-g_k)^2+(b-b_k)^2}$. Remember neighboring pixel position that has minimum distance from central pixel. If all neighboring cells are occupied, halt cell propagation.
3) If $dist > ColorTolerance$, halt cell propagation, else propagate cell in direction *k* with minimum difference.

Cell propagation halts in two cases: when all neighboring cells are occupied or when minimum distance is greater then *ColorTolerance* value. This variable is controlled by user and can take value from interval $[0, 250]$.

VI. Attractors

RULES covered in this section are: explode, implode, attract and repel. First two rules are the simplest rules in this group as they use only one attractor, while attract and repel may use arbitrary number of attractors. Attractor is the point somewhere inside the image that forces cells in attractor region to propagate toward this point.

The simplest method to fulfill such requirement would be to draw a line from seeding point to given attractor, but we have two problems: first, we don't know if some points along this line are occupied by other cells and second, cellular automaton does not remember starting (seeding) position. Also, as directions of cell's neighborhood are limited to wind rose, we have a situation of "short barrel". We cannot "fire" a cell by vector along the line seed-attractor as we do not deal with subpixel accuracy. So, to overcome this limitation we will use the concept: "if you have a short barrel, you need a smart bullet", which means that we will add some randomness in cell propagation towards attractor.

We will examine the situation of single attractor, as dealing with multiple attractors should be almost the same. When cell with attractor rule is encountered through the scanning process, we will first calculate the angle from cell position to attractor position α.

Now, as we don't have subpixel accuracy in our direction matrix, we will not relay on single direction as it will surely miss the target. We will divide our wind rose in the sectors with angle value $\frac{\pi}{4}$. Depending in what sector our angle α lies, we will choose direction randomly from respected sector and two nearby sectors.

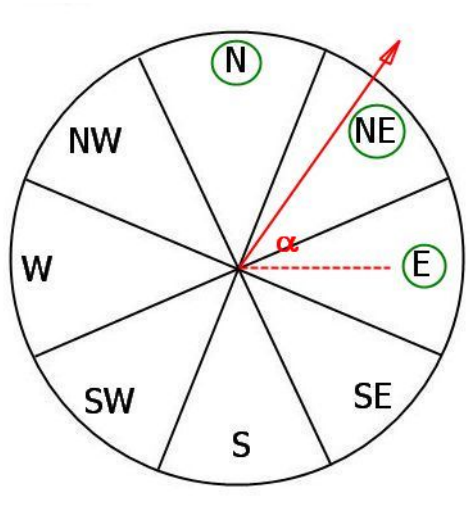


Figure 1. attractor sectors

In Figure 1. we see that our attractor angle is inside sector *NE*. We will choose direction randomly from sectors *N, NE, E:* $direction = random(N, NE, E)$. This algorithm, without modifications is used for rules *implode* and *attract*. For rules *explode* and *repel*, algorithm is slightly modified in the sense that we use mirrored sectors. Instead of angle α, we use angle $\alpha' = \alpha + \frac{\pi}{2}$, which forces cell growth from attractor, instead towards attractor.

Explode and implode rules use single attractor which lies in the center of image and cell growth is forced from the center of an image (explode) or towards the center of an image (implode).

Attract and *repel* rules use multiple attractors distributed across the image, spaced by *AttrWin* value and dithered inside the given window size (similar to grid seeding with dithering). After attractors are distributed, Voronoi tessellation is performed and polygonal regions within attractors "domain" are filled with respective attractor coordinates. For this purpose, we use two floating point images: *floatX* and *floatY*. Once filled, we don't need any additional calculations, as from the position of scanned cell $LM\,(i,j)$ we can immediately get $floatX\,(i,j)$and $floatY\,(i,j)$values, that give us closest attractor position $P_A(x,y) = [floatX\,(i,j)\,, floatY\,(i,j)]$. We only need to calculate the angle and respective sectors.

Again, *attract* rule drives cells towards attractor, while *repel* rule drives cells from the attractor.

Why all those sectors and all those randomness? Well, if we use only one sector, after few steps, we will be far away from target (attractor), as all angles inside $\frac{\pi}{4}$ sector will lead to the same direction. Adding this fuzziness to cell trajectory, gives the propagation cell more chances to correct the trajectory. In other words: the more we try to miss, the better chance we have to hit the target. This can be clearly seen if you use implode rule with small branching probability (more iterations). You'll notice that line drawings look like some kind of Brownian motion attracted by gravity field from the image center.

VII. Trackers

TRACKERS are rules that use some external grayscale map *EM* for calculating cell propagation directions, similar to Line Integral Convolution (LIC) where vector field is represented by some grayscale mask. As you'll see from some examples, even the output is very similar to some LIC-based painterly renderings, but the algorithm is much simpler. Cell propagation actually tracks the areas of similar intensity in *EM*.

The algorithm is similar to closest color rule, but instead of looking for closest color on input image, we pick up $EM\,(i,j)$ 8-bit value through the scanning process when $LM\,(i,j) = TrackByte$ (in case of TrackB rule $TrackByte = 0x17$). Then, we search through cell neighborhood to find $EM\,(i_k,j_k)$closest to $EM\,(i,j)$. If $EM\,(i_k,j_k) = 0x00$, or slot is occupied, slot is skipped. Selected slot is used to propagate the cell. If no slot is found, propagation halts. There are two types of tracking rules: TrackB and TrackG:

1) TrackB rule uses external seamless grayscale map supplied by Pixopedia 24 (background tiles). Grayscale image is filled with background tiles and passed to automaton. Prior to use the background tile, you must select one in Pixopedia 24 (background tiles button).
2) TrackG rule uses grayscale mask created from original image by applying color to grayscale transform using luminance channel values.

If no external map is supplied, trackers will work as random rule.

VIII. Moment Based Stroke Placement

IMAGE moments play an important role in image processing. In this article, we will not discuss the significance of image moments, as there are articles that deal with this topic in detail (François Chaumette [1], Rocha et al. [2]). System used in "Image Infector" is based on articles by Shiriaishi and Yamaguchi [3] and Nehab and Velho [4].

Spatial image moment of order *m,n* is defined as:

$$M_{m,n} = \sum_{i=0}^{nRows-1} \sum_{j=0}^{nCols-1} x_j^m y_i^n P_{j,i}$$

Central image moment of order m,n is defined as:

$$\mu_{m,n} = \sum_{i=0}^{imgH-1} \sum_{j=0}^{imgW-1} (x_j - x_0)^m (y_i - y_0)^n P_{j,i}$$

where $P_{j,i}$ are pixel values (in our case it's again grayscale 8-bit depth image), *imgH* is image height, *imgW* is image width and $x_0$and y_0 are gravity center's coordinates: $x_0 = M_{1,0}/M_{0,0}$ and $y_0 = M_{0,1}/M_{0,0}$.

Nice, so what about that? Well, we will use some of those image moments to get the direction of possible "brush stroke" regarding selected seeding point color with respect to some windowed neighborhood (not to be confused with our cell's neighborhood). We could use gradient method as well, but this is another approach.

First of all, in the process of seeding, we'll create so called difference image, by picking the color of seeding point at position i,j: $InpImg\,(i,j)$ (not to be confused with *i,j* position in infection process). Similar to closest color, but with neighborhood inside moment window size *mWin* value, we will calculate the distance in RGB color space over all pixels from

$$x = j - \frac{mWin}{2}, .., j + \frac{mWin}{2}$$

and

$$y = i - \frac{mWin}{2}, .., i + \frac{mWin}{2}$$

and calculate "difference" intensity: $I(y,x) = f(dist(C(i,j), C(y,x))$. Where *dist* is Euclidean distance, *f(d)* is

$$f(d) = \begin{cases} 1 - (d^2/d_0)^2 & d \in [0, d0] \\ 0 & d \in (d0, \infty) \end{cases}$$

and $d_0$is *ColorTolerance* value. Calculated intensity values are written in temporary graycale map *TM* width *width* and *height* equal to *mWin*. This function however, does not take into account situation where pixels of similar color may be "disconnected" from central pixel, by area of different color (possibly beyond tolerance value). For high moment window value, this will produce an unreasonable thick "strokes" and in most cases incorrect angle. If we involve higher order moments for "big" window and adaptive window size, we will lost simplicity, so the general rule of thumb is not to use "big" moment window and if used, it must be combined with contour clipping. From *TM* image we will calculate image moments of interest in order to get stroke width, length and angle, which are:

$$\alpha = \frac{1}{2} \arctan(\frac{b}{a-c})$$

$$w = \sqrt{6(a + c - \sqrt{b^2 + (a-c)^2})}$$

$$l = \sqrt{6(a + c + \sqrt{b^2 + (a-c)^2})}$$

where:

$$a = \frac{M_{2,0}}{M_{0,0}} - x_0^2$$

$$b = 2(\frac{M_{1,1}}{M_{0,0}} - x_0 y_0)$$

$$c = \frac{M_{0,2}}{M_{0,0}} - y_0^2$$

We could obtain angle α using central moments as well:

$$\alpha = \frac{1}{2}\arctan(\frac{\mu_{1,1}}{\mu_{2,0} - \mu_{0,2}})$$

But, in any case this angle is partially correct, as formula maps all angles in interval $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$. In order to get correct angle (for our purposes), we will transform angle according to formula:

$$\alpha^{'} = \begin{cases} \alpha & \mu_{0,2} \leq \mu_{2,0} \\ \alpha + \frac{\pi}{2} & \mu_{0,2} > \mu_{2,0}\,,\ \theta \leq 0 \\ \alpha - \frac{\pi}{2} & \mu_{0,2} > \mu_{2,0}\,,\ \theta > 0 \end{cases}$$

in order to remap our angle in interval $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$. The next step is to construct equivalent rectangle. The binary image of the equivalent rectangle has the same zeroth, first and second moments as our *TM* grayscale map and because of this fact we can replace grayscale window *TM* with equivalent rectangle of length *l*, width *w*, rotated by angle $\alpha^{'}$ from horizontal line.

Equivalent rectangles are placed in another grayscale moment mask the same size of original image. As equivalent rectangles constructed in seeding process may overlap, we will outline each rectangle with black color to disable unwanted cell propagation. Cells growth now goes the same way as with tracker rules, but with different grayscale map.

Additional parameters attached to this rule are "blur MBSP mask" which blurs rectangles with given kernel (if checked) and "fixed MBSP" which will (when checked) use fixed *l* and w: $l = mWin\sqrt{2}$ and $w = fixedWidth$, where *fixedWidth* value is supplied by user (spin edit field). This option gives nice results when combined with contour clipping.

Mask with equivalent rectangles is passed back to Pixopedia 24 as current mask so that additional actions on the mask can be performed (color bumping through mask, etc...). This is the main reason "blur MBSP mask" option is added (see examples in results section).

Moment based stroke placement is also an option in Pixopedia 24 (retouch→autobrush), but with "real" brush strokes placement.

Equivalent rectangle can be replaced by equivalent ellipse as well. This possibility will be added in one of the next Image Infector versions.

Figure 2. Original image

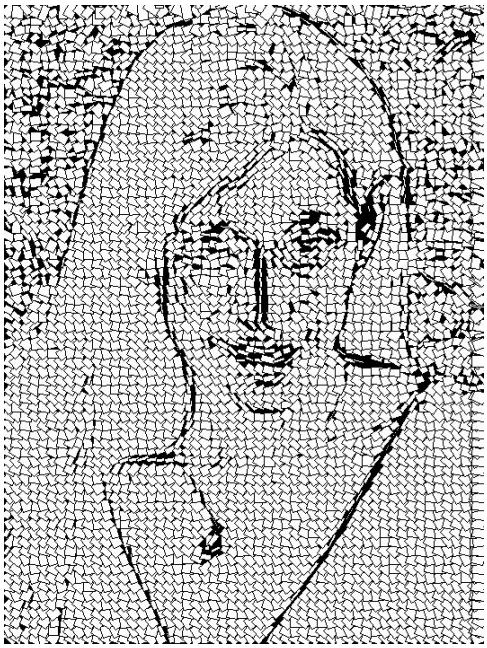


Figure 3. Equivalent rectangles (size=14) with grid seeding without dithering (size=10). As MBSP window size is greater then grid seeding size, you can notice equivalent rectangles overlaping.

IX. Infection color

OUTPUTpixel color of *OutImg* described in algorithm section is a function of *InpImg* pixels and current *OutImg* pixel value. As stated in algorithm section, *OutImg* pixel color at some new slot position is obtained using pixel color modification formula:

$$OutImg\left(i^f, j^f\right) = F(InpImg_{i,j}, OutImg(i.j))$$

The output pixel modification function depends on *InfectionStyle* chosen. The first argument of this function,

$InpImg_{i,j}$is 3x3 matrix obtained from InpImg, with central element $InpImg\,(i,j)$ surrounded by neighboring pixels. The default option in Image Infector (use original pixel value) can be overridden by selecting another method that modifies *OI* value:

1) Use original pixel value: simply copies $InpImg_{i,j}$ central pixel to the new slot:$OutImg\left(i^f,j^f\right) = InpImg\,(i,j)$.

2) Soften using 3x3 kernel: pixel value is obtained by convolving $InpImg_{i,j}$ matrix with kernel

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 9 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

where central element is replaced by $OutImg(i.j)$. Resultant pixel is then divided by 16 and saved as $OutImg\left(i^f,j^f\right)$.

3) Average two pixels: a weighted average of two pixels. New slot pixel value $InpImg\left(i^f,j^f\right)$from $InpImg_{i,j}$ is selected, and output pixel value of the new slot is calculated as: $OutImg\left(i^f,j^f\right) = OutImg\,(i,j)\,p + InpImg\left(i^f,j^f\right)q$, such that $p+q=1$.

4) Custom kernel: pixel value is obtained by convolving $InpImg_{i,j}$ matrix where central element is replaced by $OutImg(i.j)$, with user specified kernel, divided by *div* value. *Bias* is added and new pixel value is saved as $OutImg\left(i^f,j^f\right)$.

Regardless of selected option, output pixel color can be additionally modified by means of color variation (mutation). Two parameters control this option: *CVaryProb* and *CVaryIntensity*. *CVaryProb* controls probability of variation occurrence (in percentage), while *CVaryIntensity* controls maximum intensity of output color variation. Probability of variation occurrence is calculated as $p = \frac{CVaryProb}{100}$, while output pixel color is modified by formula: $OutImg\left(i^f,j^f\right) = OutImg\left(i^f,j^f\right) + CVaryIntensity - rand\,(CVaryIntensity * 2)$. Also, variation may be performed after every iteration, or only after branching occurs. This option is chosen by user, by checking appropriate check box.

Another option is possibility of changing output pixel color intensity by means of linear decay/raise of pixel intensity after each iteration. Separate slopes (*LowerSlope* and *UpperSlope*) can be set for intensities below and above value 128.

In this case, average pixel intensity (may be luminance as well) is calculated from $InpImg\left(i^f,j^f\right)$. If intensity is less then 128, first slope value (see Image Infector GUI) is used and intensity change is calculated by formula:

$$IntChange = CIN * LowerSlope$$

If intensity is higher or equal 128, intensity change is calculated as:

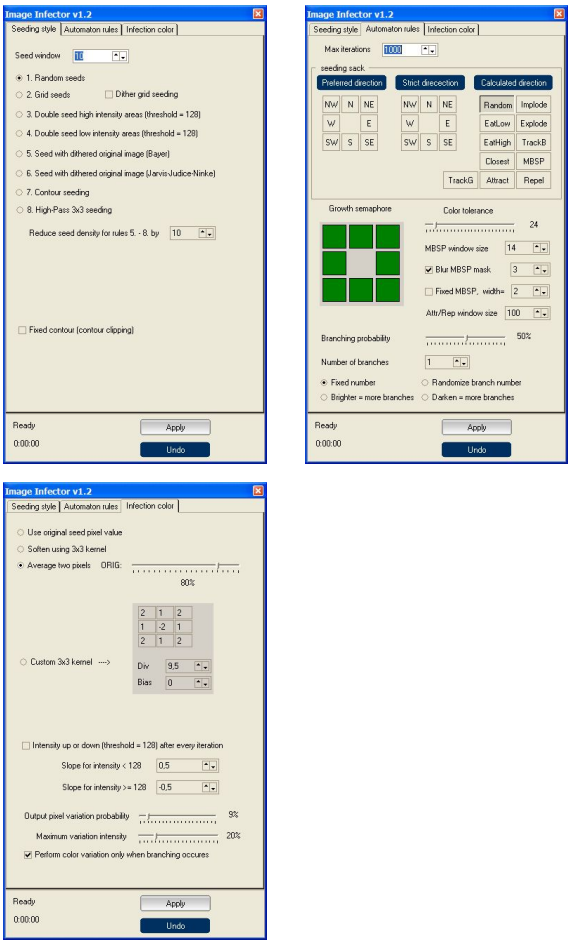
$$IntChange = CIN * UpperSlope$$

where *CIN* variable presents current iteration number and $OutImg\left(i^f,j^f\right)$ is modified using formula:

$$OutImg\left(i^f,j^f\right) = OutImg\left(i^f,j^f\right) + IntChange$$

X. Image Infector GUI

IMAGE Infector user interface follows previously described control groups: seeding, rules and infection color which are implemented as page tabs:



As you can see, groups explained in previous sections (seeding style, automaton rules and infection color) can be easily identified in the GUI. When all parameters are set, by clicking **Apply** button cellular automaton start working. To revert to original image, click **Undo** button.

XI. Results

EXAMPLES in this paper are attached in their original size. To see full-sized images, use zooming tool, or you can simple copy and save images externally.

Figure 4. Original image

Figure 5. Infected image
Example of TrackB with some twirl grayscale image and contour clipping (Laplace-Median).

Figure 6. Original image

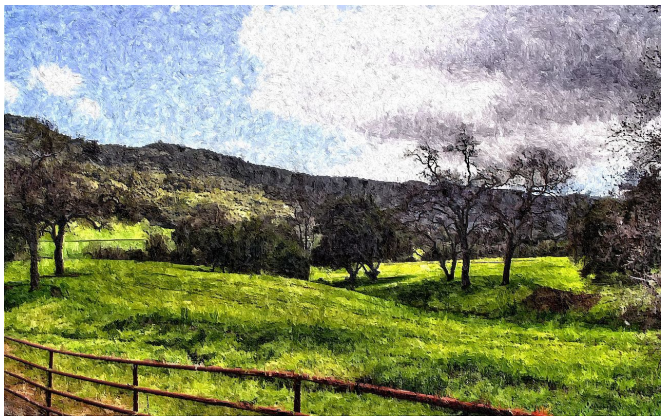


Figure 7. Infected image
Combination of TrackB with contour clipping, intensity change and image post-processing, using Pixopedia 24 pixels shifting with color bumping.

Figure 8. Original image

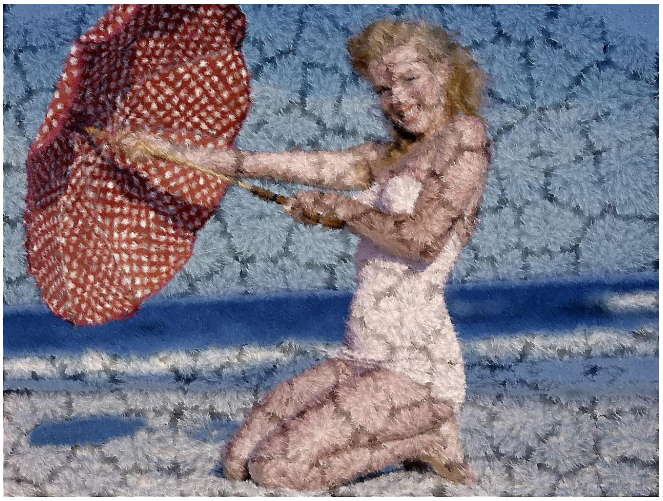


Figure 9. Infected image
Example of Attractor usage with intensity change and post-processing, using Pixopedia 24 pixels shifting with color bumping. You can clearly notice Voronoi diagram in image "craquele".

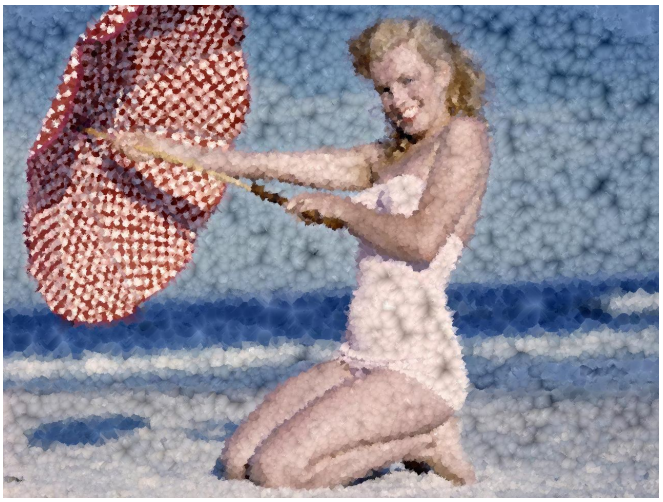


Figure 10. Infected image
Example of Repel rule with high branching probability, intensity change and pixel variation without post-processing.

Figure 11. Original image

Figure 12. Infected image
Combination of TrackG and Fixed MBSP=2 with pixel variation and Jarvis-Judice-Ninke seeding.

Figure 13. Original image

Figure 14. Infected image
Explode rule with intensity change.

Figure 15. Original image

Figure 16. Infected image
MBSP rule with fixed width width(4) and equivalent rectangle blurring (3).

Figure 17. Original image

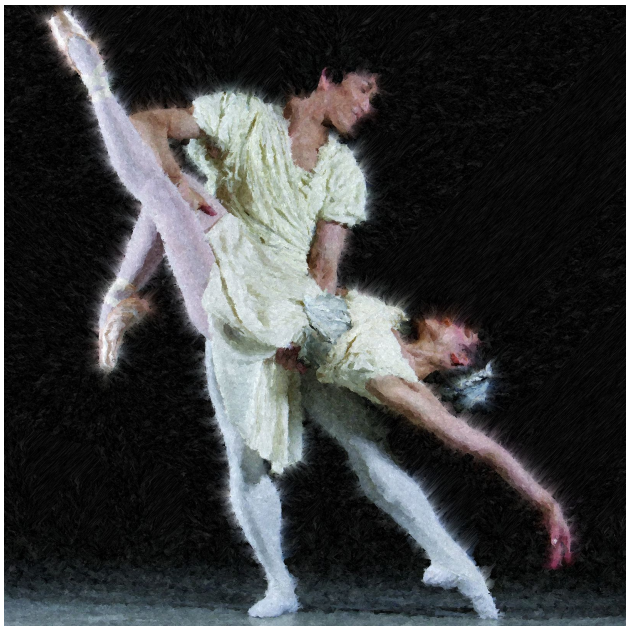


Figure 18. Infected image
Another combination of TrackG and Fixed MBSP=2 with pixel variation and Jarvis-Judice-Ninke seeding.

Figure 19. Original image

Figure 20. Infected image
Example of MBSP with blurred mask and contour seeding (Shen-Castan) with reduction factor 10.

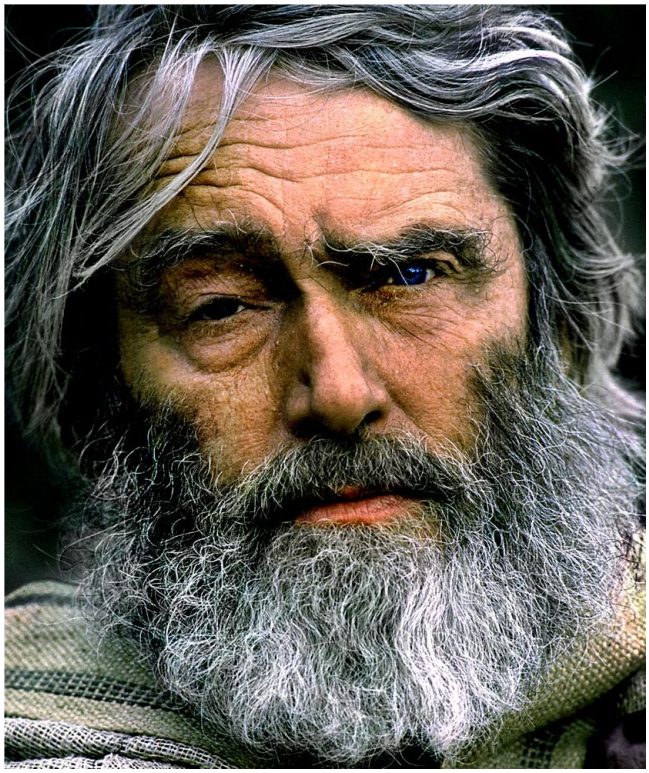


Figure 21. Original image

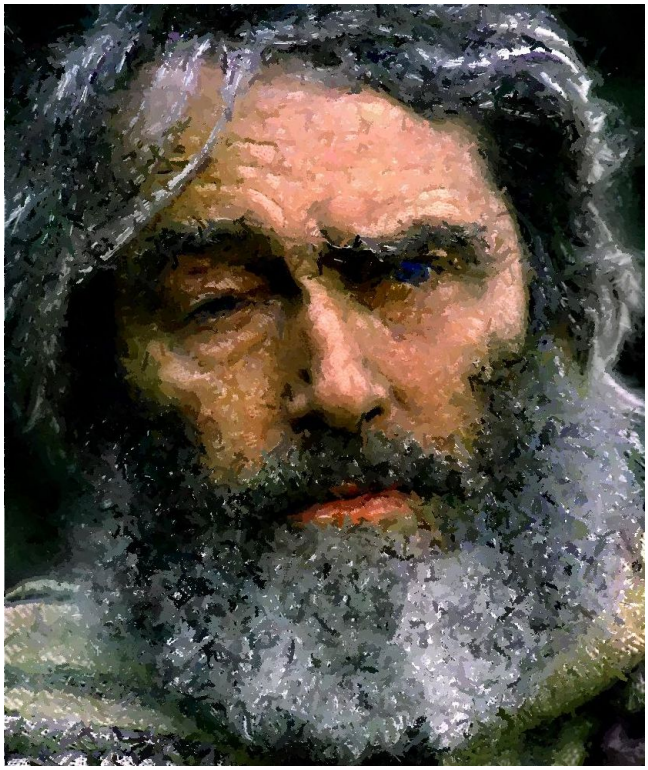


Figure 22. Infected image
Another example of contour seeding (Shen-Castan) using MBSP rule with fixed width (4) without blurring. Infection was repeated 3 times in secession.

XII. Conclusion

CELLULAR automata rules and examples covered in this article are just a small scratch on the wide surface of CA implementation in the field of NPR. Parameters used for examples in previous section are not given in full details, because it may force the user of "Image Infector" to copy parameters value, which is not my intention. The user needs to experiment with parameters values in order to achieve desired results. Small changes in parameters values may produce quite different results. Not every type of image is suitable for all rules and it again depends on user experimentation.

Anyway, here are some tips for users:

- less branching produces more tinny lines distributed over image, while higher number of branches and higher branching probability produces thicker and grouped lines (areas) of same color.
- try to experiment with various seeding types and various contours to get desired result. Note: seeding types from 5. to 8. may produce very dense seeding. You can reduce seeding density using spin editor.
- high MBSP window value may drastically increase infecting time, please keep that in mind.
- work first with smaller images or with some part of bigger image until you got desired result, especially with very dens seeding types.
- when using contour seeding, uncheck contour clipping check box.

In the time this paper is written, Image Infector version was 1.2.

References

[1] Francois Chaumette. Image moments: A general and useful set of features for visual servoing. *IEEE TRANSACTIONS ON ROBOTICS*, 20, 2004.
[2] Lourena Rocha, Luiz Velho, Paulo Cezar, and P. Carvalho. Image moments-based structuring and tracking of objects. *Instituto Nacional de Matematica Pura e Aplicada, Rio de Janeiro, Brasil*.
[3] Michio Shiraishi and Yasushi Yamaguchi. Image moment-based stroke placement. *University of Tokyo*.
[4] Diego Nehab and Luiz Velho. Multiscale moment-based painterly rendering. *PUC-Rio, IMPA*.