Coupled Map Lattice Brush

Siniša Petrić

SigmaPi Design 2015, Revision 1.
www.sigmapi-design.com

***Abstract*—This paper describes interactive digital painting technique where brush stroke is controlled via coupled map lattice model. Coupled map lattice model is an extension of cellular automata model where discrete (integer) state values of CA cells are replaced with continuous real values. Using this technique, various brush styles and strokes can be achieved.**

***Index Terms*—non-photo realistic, rendering, painterly, cellular, coupled, map, lattice, brush, stroke, non-linear, dynamical, system, artistic.**

I. Introduction

COUPLED map lattice model was proposed by Kanenko [1], as general model for the complex high-dimensional dynamics, such as biological systems, networks of DNA, economic activities, neural networks, evolution, and so on. In the field of computer graphics, CML is successfully used for image colorization/re colorization [2] and for image segmentation and encryption. Strictly speaking, Coupled Map Lattice (CML) is defined as a n-dimensional lattice where each site evolves in time through a map (or recurrence equation) of the form:

$$S^{t+1} = F(S^t) \tag{1}$$

where S is continuous variable while t is discrete. In the case of digital image, we have a two dimensional lattice, so the state of cell (pixel) at position *(i, j)* evolves in two steps:

1) Each cell is updated in discrete time step:
$$\tilde{S}^{t+1}_{i,,j} = F(S^t) \tag{2}$$
2) Subsequently each cell is averaged over set of neighbors $\tilde{S}^{t+1}_{k,l}$, so we finally get:
$$S^{t+1}_{i,,j} = (1-\epsilon)\tilde{S}^{t+1}_{i,j} + \frac{\epsilon}{N}\sum \tilde{S}^{t+1}_{k,l} \tag{3}$$

where *(k, l)* are indexes of neighbor states and ϵ stands for coupling parameter used to control relative weight of site at *i, j* and it's surrounding neighborhood .

Although the first term in CML stands for *coupled*, when the second step is omitted, we are actually speaking of uncoupled map. Roughly speaking, CML model stands for both cases, although without coupling, only difference between CML and CA is *continuous* (real) vs. *discrete* (integer) state value.

In this article, we will examine both cases (coupled and uncoupled) in order to get a wider palette of artistic brushes.

II. Process overview

BRUSH stroke rendering process starts with seeding points over input one channel floating point image by simply changing pixel value along the stroke's path. Seed points spacing is determined by some step variable, i.e. distance between each subsequent seeding point. After seeding process, we perform iterations over the stroke area, which is determined by stroke length and brush tip size. For each pixel inside our stroke, a new pixel state is calculated using some function as described in **step 1.** from previous paragraph. So what function to choose? I have decided to use CA logic and again the new pixel state *t+1* is calculated from previous pixel state *t* depending on neighborhood situation and given rule.

At this point, we introduce three possible rules and each one depends on current pixel value: *creation*, *destruction* and *grow* rules. Each rule examines the state of pixel's neighborhood and if the rule is satisfied, pixel value (state) is changed:

1) If current pixel value is 0, new pixel state (value) is determined by *creation* rule. If creation rule is satisfied, new pixel value is calculated, otherwise pixel value stays the same (0). New pixel value may be calculated from it's neighborhood or by changing pixel value to some given starting value.
2) If current pixel value is equal to some maximum intensity value (given as parameter *fMaxValue*) or *decay flag* is **true**, a destruction (or decay) rule is examined. If rule is satisfied, a new pixel value can be calculated from it's neighborhood or changed (lowered) by some decay factor ($S^{t+1} = (1-\epsilon^{decay})S^t$). After the value is changed, a *decay flag* for this pixel at position (*i, j*) is set to **true**. When new pixel state reaches minimum intensity (given as parameter), destruction process is stopped.
3) If current pixel value greater then 0 and less then some maximum intensity (given as parameter) and *decay flag* is **false**, a new pixel value can be calculated from it's neighborhood or simply changed using some constant grow factor ($S^{t+1} = (1+\epsilon^{grow})S^t$). If the new pixel value is greater then maximum value, pixel value is set to maximum value.

If we closer examine previous steps, we can see that there is actually no *grow* rule at all. Pixels growth occurs when first two rules are not satisfied. If we introduce some kind of *grow* rule, we may enter the situation where nothing happens, i.e. no rules are satisfied and nothing is drawn at all. Also, we can allow pixels *regrow* when pixel value reaches minimum value (as described in step 2.) by simply setting *decay flag* back to **false**.

Of course, everything needs to be drawn somewhere So, how many images we need? First of all, we need one 8-bit grayscale image (mask) where our brush stroke is drawn. We also need two single channel floating point images to keep new and current pixels states (images are swapped after each iteration), one 1-bit monochrome image for stroke bounding and one 1-bit monochrome image for storing decay flags. Finally, we need one 24-bit image (canvas), where brush stroke is colorized (rendered). All images are the same size (N x M). When we speak about images, we speak about two dimensional arrays of size N x M.

If coupling is involved, we need another (intermediate) floating point single channel image, to store $\tilde{S}^{t+1}$, prior to calculating S^{t+1}as given by equation (3). Also, our state values are normalized, i.e. $S \in [0, 1]$.

III. Inside the rules

RULES used for *creation* and *destruction* process are specified as 3x3 matrix with “n-ary” **Boolean** operators as elements. Matrix mid point is irrelevant as it presents current pixel at position *i, j*. I wrote “n-ary” Boolean operators in quotes, because they are not true n-ary Boolean operators.

As you may know, in Boolean algebra, besides well-known **binary** operators, there are also operators of higher order. The number of n-nary operators is 2^{2^n}, so there are 4 unary operators, 8 binary operators, etc. Those familiar with programming languages may know that most programming languages use one unary operator: NOT, mostly three or four binary operators: AND, OR, XOR, NAND and one trinary operator: IF THEN ELSE (from the set of 256). Also, in logic, a two more binary operators are used: => (implication) and <= (explication).

Operators such as OR, AND and NAND can be applied to multiple operands in old-fashioned binary way, as result of *A or B or C or D* is the same as *(((A or B) or C) or D)*. The same stands for AND and NAND operators. Now, here we need operator that would satisfy this condition:

Expression $A \oplus B \oplus C \oplus D$ is **true** if only one of the operands is **true**, otherwise is **false**.

It looks like binary XOR, but it’s not. I will use the name XOR for this operator, but keep in mind that it’s not binary XOR operator. Above expression can be evaluated using function that would count elements that are **true** and if count is exactly 1 function would return **true**, otherwise **false**. However, to be able to evaluate complete matrix in one routine, above method is slightly changed. Finally, our rule matrix elements take value from set of operators {OR, AND, NAND and “XOR”}. So, the rule matrix can look something like this:

OR	OR	OR
OR		OR
OR	OR	OR

or

XOR	XOR	XOR
XOR		XOR
XOR	XOR	XOR

or a mixture of various operators:

NAND	OR	XOR
OR		OR
XOR	OR	NAND

Each operator corresponds to respective neighbor pixel. To get Boolean value from floating point pixel value at position *k, l*, we will use simple inequality: $S^t_{k,l} > fTolerance$. Constant *fTolerance* is usually set to some small floating value, like 0.0001, so if condition is satisfied we’ll get a **true** value, otherwise **false**. Int he first example, all those values will be OR-ed. Second matrix example, uses our famous “XOR” and result will be **true** only if exactly one element in the neighborhood is **true**, otherwise **false**. The third matrix is a mixture of various operators. To avoid unnecessary and expensive **if**s, we will use an array of structure (C++):

```
struct structCmlLogic {
   BYTE Container[8];
   bool Result;
   int Count;
};
```

Logical operator are mapped to respective array indexes: {OR, AND, XOR, NAND} →{0, 1, 2, 3}. So, *logic[0]* represents OR, *logic[1]* AND, and so on. Initially, far all array elements, *Container[0]* is set to **0x01**. Neighbor pixels are examined in a left-right/up-down order, so that two-dimensional *k, l* indexes are mapped to one-dimensional array. For each neighboring pixel, we cast inequality $S^t_{k,l} > fTolerance$ to a Boolean variable. If this variable is **true**, we set respective *Container* element to **0x01**, otherwise to **0x00**:

```
void addLogicContainer(structCmlLogic *logic, int index, bool v1)
{ logic[index].Container[logic[index].Count++] = v1 ? 0x01 : 0x00; }
```

After neighborhood is examined, we perform final rule calculation:

```
bool calculateLogicContainer(structCmlLogic *logic)
  {
  int i;
  // OR
  for (i = 1; i < logic[0].Count; i++)
       logic[0].Container[0] = logic[0].Container[0] | logic[0].Container[i];
   logic[0].Result = (bool)logic[0].Container[0];
  // AND
  for (i = 1; i < logic[1].Count; i++)
       logic[1].Container[0] = logic[1].Container[0] & logic[1].Container[i];
  logic[1].Result = (bool)logic[1].Container[0];
  // XOR
  for (i = 1; i < logic[2].Count; i++)
logic[2].Container[0] = logic[2].Container[0] + logic[2].Container[i];
  logic[2].Result = (bool)(logic[2].Container[0] & 0x01);
  // NAND
  for (i = 1; i < logic[3].Count; i++)
       logic[3].Container[0] = logic[3].Container[0] + logic[3].Container[i];
  logic[3].Container[0] = (logic[3].Container[0] >> 1);
  logic[3].Result = !(bool)logic[3].Container[0];
  // Result
  return logic[0].Result && logic[1].Result && logic[2].Result && logic[3].Result;
  }
```

When the rule is satisfied, we will change pixel value at position *i, j* by some function. If the rule is *creation*, we simply set pixel value to constant seed value *fSeed*Value. If the rule is *destruction* we can decrease pixel value by some amount. Otherwise, we have *growing* process and pixel value is increased by some amount. For example:

1) If *creation* rule *Cr* is satisfied: $\tilde{S}_{i,j}^{t+1} = fSeed$
2) If *destruction* rule *Dr* is satisfied: $\tilde{S}_{i,j}^{t+1} = (1 - \epsilon^{decay})S_{i,j}^{t}$
3) Else, *growing* is performed:$\tilde{S}_{i,j}^{t+1} = (1 + \epsilon^{grow})S_{i,j}^{t}$

We have used linear value transformation, but we can apply any kind of function to get desired result. So, our recurrence mapping (2) would look something like this:

$$\tilde{S}_{i,j}^{t+1} = \begin{cases} S_{i,j}^{t} = 0 \wedge Cr = true \Rightarrow fSeed \\ S_{i,j}^{t} > 0 \wedge Dr^{*} = true \Rightarrow (1 - \epsilon^{decay})S_{i,j}^{t} \\ S_{i,j}^{t} > 0 \wedge Dr^{*} = false \Rightarrow (1 + \epsilon^{grow})S_{i,j}^{t} \end{cases}$$

Where symbol Dr^{*}denotes *destruction* rule in combination with *decay flag* $\delta_{i,j}$. If *decay flag* is false, $Dr^{*}$is always **false**, otherwise Dr rule is applied. Now, we can finally write down an algorithm(s) for our stroke.

IV. Stroke drawing

ALGORITHM for stroke drawing is quite straight forward. We will first examine an algorithm for *uncoupled* map lattice and with few changes we'll get an algorithm for *coupled* map lattice.

Before we start our process, we must set all values to zero, for all images except 24-bit output image used for stroke rendering. After initialization, we draw seeding points along the stroke's path. For each iteration we process our CML. After iteration counter reaches maximum value, we perform stroke rendering:

```
Main routine:
  1) Set all values to zero (all images except 24-bit output image).
  2) Seed stepwise points following stroke's path with value fSeedValue in fInput image.
  3) Process CML.
  4) Swap fInputImage and fOuputImage.
  5) Repeat step 3. until maximum number of iterations is reached.
  6) Render stroke onto canvas.
```

Now, we will go into detail regarding step 3. In case of uncoupled lattice, we simply call *processUncoupled* method which accepts two parameters, a pointer to output image and Boolean variable that specifies shell we put pixels in brush/stroke mask or not.

```
processUncoupled(paramOutput, paramDrawMask):
  1) Loop through stroke bounding rectangle and fetch value $S_{i,j}$from fInput image.
  2) If $S_{i,j} > 0$ examine decay flag and check destruction rule:
       a) If $\delta_{i,j} = true \wedge Dr = true \Rightarrow$ $\tilde{S}_{i,j} = (1 - \epsilon^{decay})S_{i,j}^{t}$.
            i) If $\tilde{S}_{i,j} < fMinValue \Rightarrow$. $\tilde{S}_{i,j} = fMinValue$
           ii) Put a new value into paramOutput image.
       b) If $\delta_{i,j} = false$, proceed to next step.
       c) Perform pixel growing: $\tilde{S}_{i,j} = (1 + \epsilon^{grow})S_{i,j}^{t}$.
       d) If $\tilde{S}_{i,j} \geq fMaxValue \Rightarrow$
          $\tilde{S}_{i,j} = fMaxValue$. Set decay flag to true: $\delta_{i,j} = true$.
       e) Put a new value into paramOuput image.
  3) If $S_{i,j} = 0$, apply creation rule:
       a) If $Cr = true \Rightarrow \tilde{S}_{i,j} = fSeedValue$.
       b) Else, $\tilde{S}_{i,j} = S_{i,j}$.
       c) Put a new value into paramOutput image.
  4) If $paramDrawMask = true$ multiply new pixel value by 255 and put it into brush/stroke mask (8-bit grayscale image).
  5) End loop when $i, j$ reach the end of bounding rectangle.
```

So, instead of step 3.in main routine, we simply put a call to our *"uncoupled"* method: *processUncoupled(fOutput, true);*

In case of *coupled* lattice, we will first call *"uncoupled"* method, but with different parameters: *processUncoupled(fIntermediate, false);* and then we will execute following steps:

```
processCoupled()
  1) Loop through stroke bounding rectangle and fetch
     value $\tilde{S}_{i,j}$ from fIntermediate image.
  2) Perform coupling:
     $S_{i,,j}^{t+1} = (1-\epsilon)\tilde{S}_{i,j}^{t+1} + \frac{\epsilon}{N}\sum \tilde{S}_{k,l}^{t+1}$
  3) Put the new value into fOutput image.
  4) Multiply new pixel value by 255 and put it into
     brush/stroke mask (8-bit grayscale image).
  5) End loop when $i, j$ reach the end of bounding rect-
     angle.
```

So, for *coupled* lattice, step 3. in main routine is replaced by these steps:

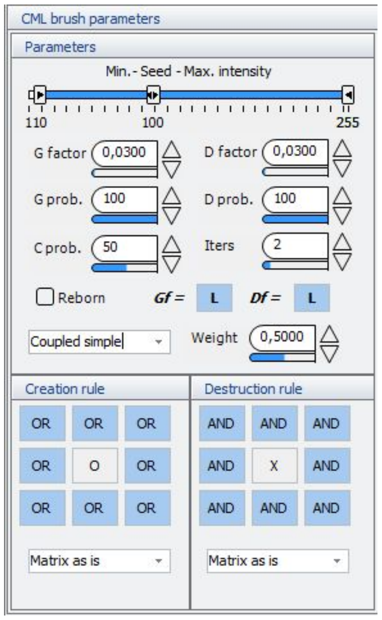
- *processUncoupled(fIntermediate, false);*
- *processCoupled();*

Mate.

V. Modifications and implementation

SOME modifications to our algorithm can give quite a boost to artistic effects that we can achieve using this technique. First of all, we can introduce some randomness in our model. We will do so by adding probability parameters regarding creation, destruction and growth rule. Each parameter specifies the probability of executing it's respective rule. Also, We can let user to select function (2): linear, quadratic, etc. Rule matrix can be modified as well: elements can be rotated (counter clockwise) or shuffled.

Implementation is done in Pixopedia 2014 from version 0.4.1. and up: By selecting "CML brush" following parameters appear:


CML brush parameters

Parameters

Min.- Seed - Max. intensity: 110, 100, 255

G factor 0,0300 — D factor 0,0300

G prob. 100 — D prob. 100

C prob. 50 — Iters 2

☐ Reborn — *Gf* = L — *Df* = L

Coupled simple — Weight 0,5000

Creation rule

OR	OR	OR
OR	O	OR
OR	OR	OR

Matrix as is

Destruction rule

AND	AND	AND
AND	X	AND
AND	AND	AND

Matrix as is

We will briefly explain parameters meaning and conjunction to variables from previously described algorithms:

- **Min.-Seed-Max. intensity** (divided by 255) correspond to *fMinValue, fSeedValue, fMaxValue* respectively.
- **G factor** and **D factor** sliders correspond to ϵ^{grow}, ϵ^{decay}respectively.
- **G prob., D prob., C prob.** are probabilities for grow, destruction/decay and creation. Values range from [0,100]. Internally, probabilities are normalized (scaled to [0,1]).
- **Iters** parameter specifies number of iterations.
- **Reborn** (regrow) parameter specifies if decay flag will be set back to false after minimum intensity in destruction/decay process is reached in order to allow re-growing of pixel.
- ***Gf*** and ***Df*** are used to select recursive function (linear, quadratic, neighbor bulling and inverse neighbor) for growing and destruction/decay respectively. Function type is changed by clicking on appropriate sky-blue colored rectangles.
- Combo box below Reborn check boxes is used to select *uncoupled, coupled simple* or *coupled Nzn* mode. *Uncoupled* and *Coupled* simple modes are the same as described in previous sections and *Coupled Nzn* mode slightly differs from simple coupling as it sums only neighboring pixels that are greater then zero (NzN - non-zero neighbors).
- **Weight** slider is used to set ϵvalue in coupling equation (3).
- **Creation** and **Destruction** rules matrices correspond to *Cr* and *Dr.* Operators are changed by clicking on appropriate sky-blue colored rectangles.
- Two combo boxes on the bottom are used to specify matrix modification: *as is, rotate* and *shuffle.*

When all global (brush size, step, bumping, filters, etc...) and specific (as describe above) parameters are set, simply draw the stroke over image. Until mouse button is released, previously described process is performed. You will notice that stroke appearance changes depending on how fast you move your mouse. Because of wast amount of available parameters, lot of experimentation form the user's side is required in order to get acquaintance with CML brush. In the next section we will describe few settings that give very pleasant artistic effects.

VI. Examples and conclusion

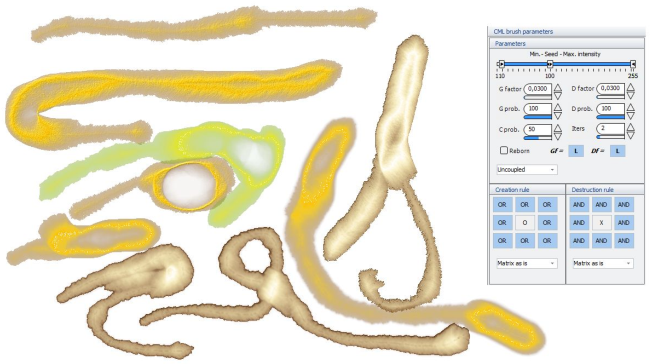
FOR the sake of simplicity, most examples presented in this article use the same brush size set to 60. With every figure, picture of currently used brush parameters is attached.

To conclude this article, some thoughts on future work:

- Besides using CML only for brush stroke construction (which is actually 8-bit grayscale mask), we can also couple (in rendering process) original image, full color 24-bit pixels value with brush color, to achieve interesting "colorizing" effects.
- We can use some "non-blank" image and apply CML on randomly seeded points, or use image contour as stroke

path, performing our rules and coupling methods on full color 24-bit images.
- Use background tiles to simulate various papers/canvases and let CML stroke interact with them.

Of course, readers feedback and implementation of any news and achievement in the field of CA and CML is highly recommended..


CML brush parameters

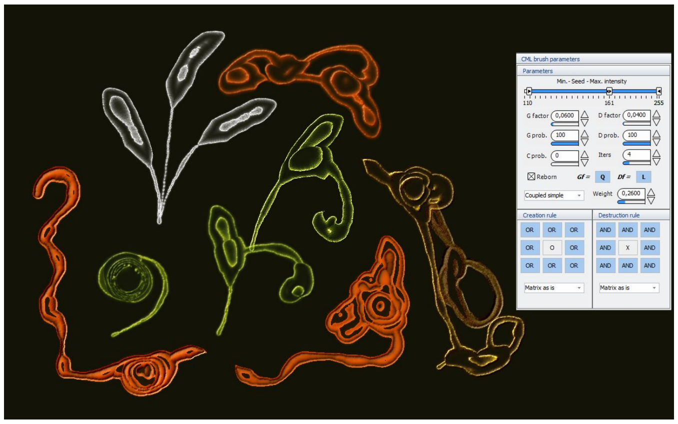
Parameters

Min.- Seed - Max. intensity: 110 – 100 – 255

G factor 0,0300 · D factor 0,0300 · G prob. 100 · D prob. 100 · C prob. 50 · Iters 2 · ☐ Reborn · *Gf* = L · *Df* = L · Uncoupled

Creation rule			Destruction rule		
OR	OR	OR	AND	AND	AND
OR	O	OR	AND	X	AND
OR	OR	OR	AND	AND	AND
Matrix as is			Matrix as is		

Figure 1. Uncoupled lattice using default CML settings with brush bumping, blur filtering and stroke inversion.


CML brush parameters

Parameters

Min.- Seed - Max. intensity: 110 – 161 – 255

G factor 0,0600 · D factor 0,0400 · G prob. 100 · D prob. 100 · C prob. 0 · Iters 4 · ☒ Reborn · *Gf* = Q · *Df* = L · Coupled simple · Weight 0,2600

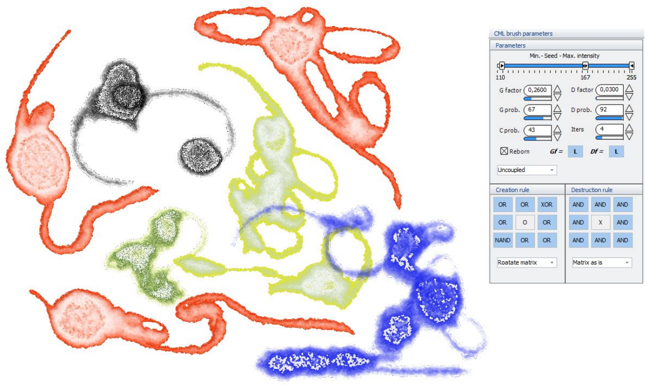
Creation rule			Destruction rule		
OR	OR	OR	AND	AND	AND
OR	O	OR	AND	X	AND
OR	OR	OR	AND	AND	AND
Matrix as is			Matrix as is		

Figure 2. Coupled lattice (simple) using reborn option without creation process.

References

[1] K. Kanenko (Ed.). Theory and applications of coupled map lattices. *Wiley, New York*, 1993.

[2] V. Vezhnevets V. Konushin. Interactive image colorization and recoloring based on coupled map lattices. *Graphics and Media Lab., Lomonosov Moscow State University, Moscow, Russia*, 2010.


CML brush parameters

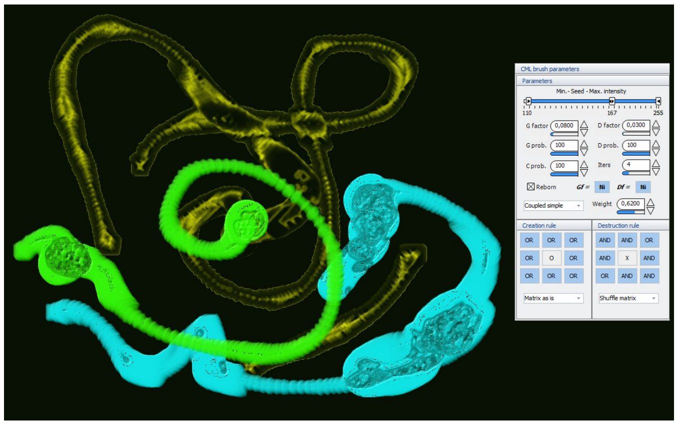
Parameters

Min.- Seed - Max. intensity: 110 – 167 – 255

G factor 0,2600 · D factor 0,0300 · G prob. 67 · D prob. 92 · C prob. 43 · Iters 4 · ☒ Reborn · *Gf* = L · *Df* = L · Uncoupled

Creation rule			Destruction rule		
OR	OR	XOR	AND	AND	AND
OR	O	OR	AND	X	AND
NAND	OR	OR	AND	AND	AND
Roatate matrix			Matrix as is		

Figure 3. Uncoupled lattice with altered creation matrix


CML brush parameters

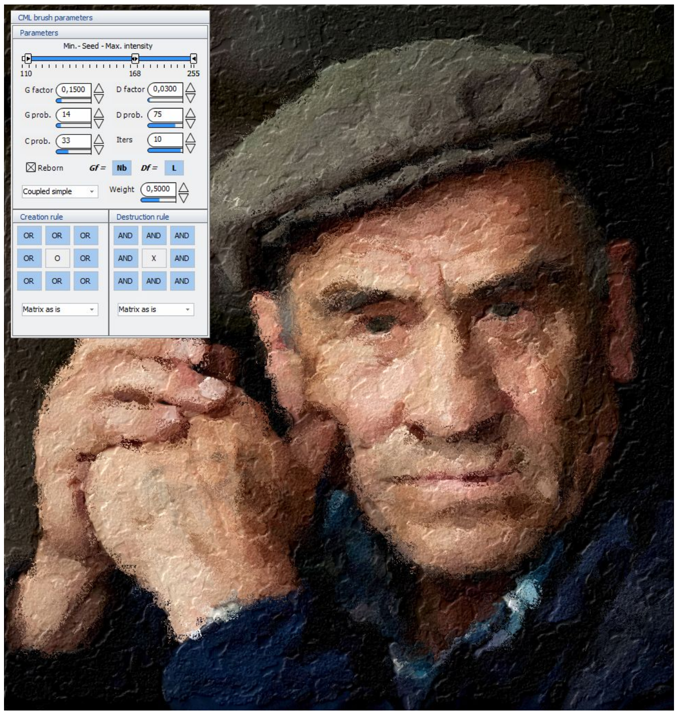
Parameters

Min.- Seed - Max. intensity: 110 – 167 – 255

G factor 0,0800 · D factor 0,0300 · G prob. 100 · D prob. 100 · C prob. 100 · Iters 4 · ☒ Reborn · *Gf* = Ni · *Df* = Ni · Coupled simple · Weight 0,6200

Creation rule			Destruction rule		
OR	OR	OR	AND	AND	OR
OR	O	OR	AND	X	AND
OR	OR	OR	OR	AND	AND
Matrix as is			Shuffle matrix		

Figure 4. Coupled lattice with altered destruction matrix


CML brush parameters

Parameters

Min.- Seed - Max. intensity: 110 – 168 – 255

G factor 0,1500 · D factor 0,0300 · G prob. 14 · D prob. 75 · C prob. 33 · Iters 10 · ☒ Reborn · *Gf* = Nb · *Df* = L · Coupled simple · Weight 0,5000

Creation rule			Destruction rule		
OR	OR	OR	AND	AND	AND
OR	O	OR	AND	X	AND
OR	OR	OR	AND	AND	AND
Matrix as is			Matrix as is		

Figure 5. Coupled lattice (simple) with brush size set to 14, using autobrushing technique