Bringing Images to Life with Two-D Cellular Automata by Paul Reiners

A direct cellular automata animated convolution, using Life or Brian's Brain, of Van Gogh's Sunflower paintings adds a subtle, scintillating effect to the originals.
A gentle introduction to convolution filters

Most digital artists, even if they don't know the term, are familiar with image convolutions, and, in fact, probably use them almost every day in their work. Image convolutions are commonly called filters. Some commonly used convolution filters are those that blur an image, lighten an image, darken an image, or increase contrast in an image.

For example, let's look at how you might blur an image. As everyone knows, a digital image is simply a grid of pixels. If you think about it, to blur an image, you would want to make each pixel in its image a little bit more like its neighboring pixels. Blurred images tend to not have sharp edges. For simplicity, let's consider black and white images only. In a black and white blurred image, there won't be a black region directly adjacent to a white region (we use red lines to delineate the pixels---the red isn't part of the image):

Figure 1

There will be a transitional gray area between the black and white regions.

Figure 2

So, how can we create the second image from the first? That is, how can we blur the first image, thus obtaining the second. The unhelpful answer would be to say that it is done using a convolution filter. So, what is a convolution filter, exactly, and how does it do what it does?

First of all, we see pixels as white, black, or gray (of some shade) dots (in the case of black and white images, which is what we're restricting ourselves to for now). However, the computer thinks of a pixel as a number from, say, 0 to 255 (inclusive). 255 could represent the color white, while 0 could represent the color black. (Or the computer could consider 0 white and 255 black or use more than 256 possible values. For our purposes, that's irrelevant.) Any values between 0 and 255 would be shades of gray with higher numbers being lighter shades of gray.

So, what does our original non-blurred image look like to a computer? Well, it would look like this:

Figure 3

And what does the blurred version look like to the computer? Well, it looks like this:

Figure 4

So, if we wanted to program a computer to blur an image, we would essentially be asking it to derive a grid like the second from a grid like the first. And how can we compute the second grid from the first grid? Here is a hint: the new numeric value of each pixel depends only on its current value and the current values of its 8 neighbors (above, below, to the left, to the right, to the upper left, to the lower left, to the upper right, and to the lower right).

To blur an image, for each pixel, the new value of the pixel is simply the average of the old value of the pixel and the values of its 8 immediate neighbors. Note that a pixel can compute its new value locally by only using its current value and the current values of its immediate neighbors. This is an important fact and will have an analogue in cellular automata, which we will look at in the next section.

Convolutions are defined by convolution kernels, which are simply matrices (usually 3 x 3 matrices, although they don't need to be). Here is a convolution kernel for blurring:

Figure 5

This kernel allows us to derive the new value for a pixel from its old value and the old value of its 8 neighbors. Let's look at how to compute the new value of a particular pixel in our original non-blurred image. Consider the highlighted pixel below and suppose we want to compute its value in the blurred image:

Figure 6

Let's apply the convolution to this pixel. We will also need to know the pixel's neighbors' values. We get the new pixel value by multiplying each of the 9 pixel values by the corresponding number in the convolution filter and adding up all these products:

Figure 7170 = (0 * 1/9) + (0 * 1/9) + (0 * 1/9) + (255 * 1/9) + (255 * 1/9) + (255 * 1/9) + (255 * 1/9) + (255 * 1/9) + (255 * 1/9)

So get a new pixel value of 170.We apply this operation simultaneously to each pixel in the image. (It gets a little complicated for the pixels on the borders of the image, because they won't have 8 neighbors. However, if you think for awhile, you should be able to come up with strategies for dealing with this.)

The blurring convolution filter is a good example because it's intuitively clear how it works. It's also not such a good example, because all nine values in the filter happen to be the same, 1/9. Suppose we wanted the new image to not be quite so blurred. One way we could do this is to let the original value of the pixel weigh more heavily than the values of the neighboring pixels. This filter would do the job:
Figure 8

One thing to note here is that the sum of the squares have all added to 1 in these two filters. This doesn't have to be the case, though, although it is common. If the sum is 1, then the average level of the image will not change after the filter is applied. Sometimes it is useful to have filters in which the values of the squares do not sum to 1 and then normalize the output of the filter so that they do sum to 1. Consider the following arbitrary convolution kernel, which consists of only 0s and 1s:

Figure 9

The sum of the cells is 4, not 1. To normalize this, we would divide each cell by 4, obtaining the following normalized convolution kernel:

Figure 10

This leads us to our next topic, cellular automata. A gentle introduction to cellular automata

Before giving a more general definition of cellular automata (CA), we will look at a particular cellular automaton, Brian's Brain. Brian's Brain is a cellular automaton invented by Brian Silverman. Brian's Brain consists of a 2-D grid of individual cells (or squares). This grid goes off to infinity in both dimensions. Each cell is either on or off. This can be represented by having either a 1 or a 0, respectively, in the cell. This grid of cells is the cellular automaton's universe. There is also the concept of time in cellular automata. Time consists of discrete ticks of a clock. At each tick of the clock, each cell updates its value depending on its current value and the current value of its eight neighbors according to an update rule. All of these cell updates happen simultaneously.

As explained in an article by Mitchel Resnick and Brian Silverman , the update rule for individual cells (or squares) is: If a square is on, it turns off. If a square is off, it turns on if exactly two neighboring squares are on. But there is one small twist: When a square turns off, it can't turn on in the very next iteration.

Cellular automata in action yield ever-changing, interesting patterns. Evolving patterns that are complex and interesting enough that they can be considered artistic despite their simple and deterministic nature. Take a moment to watch Brian's Brain in this animation created by George Miller :

Figure 11

The update rule used defines the cellular automaton. There are other cellular automata, such as John Conway's Game of Life and Cyclic Space, that have other rules. For more information on other CA, you can consult one of the references listed below. We're looking at one specific cellular automaton, but cellular automata are much more general. For example, the cell values (or states) need not be restricted to just on or off (or 1 and 0).They can range from 0 to n, where n is some arbitrary number. Also, they need not be two-D. For example, they may be three-D.

There is a reason why we're restricting ourselves to two-D, though. That is because images are two-D. Cellular automata clearly are analogous to digital images with cellular automata cells corresponding to image pixels. We will exploit this

analogy in the rest of the article. However, cellular automata have one characteristic that images do not have: they are dynamic and evolve over time. We will use the dynamic nature of CA to animate still images. We will call these animated image convolutions. The technique is somewhat technical. This is unavoidable to a certain extent, but, if you find some of the details heavy going, try to keep the general goal in mind: We are trying to animate still images using the dynamism of cellular automata through the technique of convolution filters. The basic idea is pretty simple once you understand both cellular automata and image convolutions. If you feel you don't understand those, yet, you should read the material above a second time or perhaps read the Wikipedia article on cellular automata and some Flash documentation on convolution filters . Introduction to using cellular automata as convolution filters

We will now examine two types of animated image convolutions that are generated from two-D cellular automata. Traditional convolutions take a static image as input and, using a convolution kernel, create a static image as output. Animated convolutions, as I define them, use a sequence of related kernels and successively apply the kernels to the original image. This results in a sequence of filtered images. We can thus create an animation by showing the sequence of filtered images successively.

We will present two different ways of creating a sequence of related kernels. Both of these use two-D CA. The first technique uses CA to operate on the output of a user-defined 3-by-3 convolution kernel. The second technique uses the states of the CA itself as 3-by-3 kernels.

Overview of CA convolution techniques

Each of these convolution techniques operates on the RGB values of the pixels in the image (the techniques could be easily adapted to operate on HSB or other representations of pixel colors, also). The convolutions operate separately on each of the R, G, and B values and then combine the new values to compute the filtered value. Below, when we talk about applying a kernel k to a pixel p, this will be shorthand for applying k to the individual RGB values of p and then combining the resulting convolved R, G, and B values to create the new convolved pixel color.

Both of the convolution techniques will use standard 3-by-3 kernels.

The CA used will have n states, which are integers from 0 to n - 1 (inclusive). For example, in Conway's Game of Life, the value of n is 2. For other CA, such as Cyclic Space, n can be arbitrarily large.

For a given image, we create a CA of the same width and height. Thus, there is a 1-1 mapping between the pixels in the image and the cells in the CA. (This CA will be seeded initially with random values.)

The code I give in this paper is in Java and uses the Processing libraries. However, the technique can be easily implemented in other languages. If you're not a programmer, do not be too intimidated by the computer code. Concentrate on the comments and just browse the code. Even if you're not a programmer, the overall logic of the code should become clear by doing this.

For both of these techniques, we will need to compute the new RGB value of a pixel in an image using the state of the corresponding CA cell and the pixel's old RGB value. To do this, we will implement the abstract method convolve(int, int, int, int)./**
 * @param y
 * the y-coordinate of the pixel (and CA cell)
 * @param x
 * the x-coordinate of the pixel (and CA cell)
 * @param state
 * the state of the CA cell at (x, y)
 * @param pixel
 * the color value of the pixel at (x, y)
 * @return the new color value of the pixel at (x, y)
 */
abstract public int convolve(int y, int x, int state, int pixel); Convolutions weighted by CA

In our first technique, the filtered value of the pixel is a weighted combination of the original value and the value obtained by applying a user-defined kernel. The weight is determined by the state of the corresponding cell in the CA.

In particular, let k be the user-defined kernel. For an individual pixel, p, let k(p) be the value obtained by applying k to p. Now, let s be the state of the CA cell corresponding to the pixel p (recall that s is an integer between 0 and n - 1). The new value p' of the transformed pixel will be a weighted average of p and k(p) with k(p) weighted by s / n. Hence p' = (k(p) * s + p * (n - s)) / n

(Note that there is an obvious way to generalize this: That is to have a lookup table from the states to weights. Using the states themselves as weights is somewhat arbitrary.)CA-weighted convolution code

In getKernelSum(int, int, Color), we compute a new R, G, or B value in the usual way using the user-defined kernel./**
* Computes the result of applying the convolution to a given pixel.
*
* @param x
* the x-coordinate of the pixel
* @param y
* the y-coordinate of the pixel
* @param rGB
* whether we are operating on the R, G, or B component of the
* pixel color
* @return the new R, G, or B value of the pixel after performing the convolution.
*/
private float getKernelSum(int x, int y, Color rGB) {
float sum = (float) 0.0;
// Iterate over the 3 x 3 convolution kernel
for (int ky = -1; ky <= 1; ky++) {
for (int kx = -1; kx <= 1; kx++) {
// img is a one-dimensional array containing the image // pixels. Need to compute its position in this 1-D array // from its 2-D coordinates. int pos = (y + ky) * img.width + (x + kx);

int pixel = img.pixels[pos];
float val;
if (rGB == Color.RED) {
val = red(pixel);
} else if (rGB == Color.GREEN) {
val = green(pixel);
} else {
val = blue(pixel);
}

sum += kernel[ky + 1][kx + 1] * val;
}
}

return sum;
}

In getWeightedKernelSum(int, int, int, int, Color), we compute a weighted average of the original R, G, or B value with the convolved value (computed in the previous function) using the CA state as the weight. /** * @param x * the x-coordinate of the pixel (and the CA) * @param y * the y-coordinate of the pixel (and the CA) * @param state the state of the CA at (x, y) * @param pixel the (color) value of the pixel at (x, y) * @param rGB * whether we are operating on the R, G, or B component of the * pixel color * @return the new (color) value of the pixel at (x, y) */ private float getWeightedKernelSum(int y, int x, int state, int pixel, Color rGB) { float origRGB; if (rGB == Color.RED) { origRGB = red(pixel); } else if (rGB == Color.GREEN) { origRGB = green(pixel); } else { origRGB = blue(pixel); } float rGBSum = getKernelSum(x, y, rGB); int n = cellularAutomaton.getN(); float weightedRGBSum = (state * rGBSum + (n - state) * origRGB) / n;
return weightedRGBSum; }

Finally, we combine the convolved R, G, and B values. public int convolve(int y, int x, int state, int pixel) { float weightedRSum = getWeightedKernelSum(y, x, state, pixel, Color.RED); float weightedGSum = getWeightedKernelSum(y, x, state, pixel, Color.GREEN); float weightedBSum = getWeightedKernelSum(y, x, state, pixel, Color.BLUE); int c = color((int) weightedRSum, (int) weightedGSum, (int) weightedBSum);
return c; }

You can view this algorithm in action here. Here is a still photo of the algorithm in action:

Figure 12
Direct CA convolutions

Our next technique is simpler, but more profound in a sense. Rather than use the cell states of a CA to weight a traditional kernel, we use the cell states themselves as kernels. Thus, instead of using a single kernel for the entire image, we use a different 3-by-3 kernel for each pixel in the image. The kernel for a given pixel consists of the state of

the CA cell corresponding to the pixel and the states of the eight neighbors of this cell (giving us a 3-by-3 kernel). We then normalize this kernel so that the sum of the kernel elements is 1.

I've found that, if the number of possible states n is large, the resulting animation tends to be smoother. This should be fairly obvious. However, it's also the case with small n that it's possible to change the values of the states to obtain more smooth results.

For example, with Conway's Game of Life (in which n is 2), I first tried giving 'dead' cells a state value of 0 and 'live' cells a state value of 1. Because there tend to be more dead cells than live cells in a typical Life universe, in the resulting animation, the CA tended to dominate the original image. In effect, you basically saw a Life animation where the color of the live cells was the color of the 'underlying' image pixels. This can be fun to watch, but by simply switching the state values and giving live cells a value of 0 and dead cells a value of 1, I obtained much more subtle and aesthetically pleasing results. The image was subtly animated and not dominated by the mechanics, as it were, of the Life CA.

Also, although I've assumed that the CA states are integers from 0 to n - 1, there's no reason why they need be this. Interesting results I'm sure could be obtained using different integer (or floating-point values), including negative values. The easiest way to implement this would be to have a look-up table between states and individual kernel elements. There is a lot of room for exploration in this area. Direct CA convolution code

In getKernelSum(int, int, Color), we use the states of the CA cells as the kernel elements in computing a new R, G, or B value. /** * @param x * the x-coordinate of the pixel * @param y * the y-coordinate of the pixel * @param rGB whether we are operating on the R, G, or B component of the * pixel color * @return the new R, G, or B value of the pixel after performing the convolution */ private float getKernelSum(int x, int y, Color rGB) { float sum = (float) 0.0; int stateTotal = 0; // Iterate over the 3 x 3 convolution kernel for (int ky = -1; ky <= 1; ky++) { for (int kx = -1; kx <= 1; kx++) { // img is a one-dimensional array containing the image // pixels. Need to compute its position in this 1 array // from its 2-D coordinates. int pos = (y + ky) * img.width + (x + kx); int pixel = img.pixels[pos]; float val; if (rGB == Color.RED) { val = red(pixel); } else if (rGB == Color.GREEN) { val = green(pixel); } else { val = blue(pixel); } // Use the corresponding CA state as the kernel element. int state = cellularAutomaton.getState(ky + y, kx + x); sum += state * val; stateTotal += state; } } // Normalize float kernelSum = sum / stateTotal; return kernelSum; }

Combine the individual R, G, and B values as before. public int convolve(int y, int x, int state, int pixel) { float rSum = getKernelSum(x, y, Color.RED); float gSum = getKernelSum(x, y, Color.GREEN); float bSum = getKernelSum(x, y, Color.BLUE); int c = color((int) rSum, (int) gSum, (int) bSum); return c; }

You can view this algorithm in action here. Here is a still photo of the algorithm in action (which doesn't show a lot, since I'm using a fairly subtle filter and this technique is for animating still pictures):

Figure 13

As I said, with this technique we use the states of the CA cells as the kernel elements. However, this is needlessly restrictive and using the integers 0 up to n in the kernel is quite arbitrary. A more general technique would be to construct a mapping between the states of the CA and the values placed in the kernel. You could get much more varied results using this technique.

Display code

The display code is quite straightforward and is the same for both types of convolutions. We simply iterate over the pixels and convolve each. draw() is called repeatedly by the Processing framework, causing animation. During each call to draw(), we update the CA by one step. public void draw() { for (int y = 1; y < img.height - 1; y++) { for (int x = 1; x < img.width - 1; x++) { int state = cellularAutomaton.getState(y, x); int pos = y * img.width + x; img.pixels[pos]; int c = cAConvolution.convolve(y, x, state, pixel); newImg.pixels[pos] = c; } } newImg.updatePixels(); image(newImg, 0, 0); cellularAutomaton.update(); }

Other artists Other artist/programmers using cellular automata to generate art include:

- Paul Brown
- Scott Draves Conclusion

I've obtained aesthetically interesting results with both of these techniques. With CA-weighted animated convolutions, I've tried both Cyclic Space and Brian's Brain. With Cyclic Space, the CA tended to dominate the resulting animation at the expense of the image as filtered by the user-defined kernel. This can be okay, if that's what you're aiming for. However, I obtained more subtle and interesting results using Brian's Brain. With Brian's Brain, it looked like the image as transformed by the user-defined kernel was being subtly animated and it was hard to see the actual mechanics of Brian's Brain in the animation (this isn't entirely true---you could see gliders in patches of solid color). I think that using CA-weighted animated convolutions is an interesting way to give life to a traditional convolution.

However, I think direct CA animated convolutions are more interesting both philosophically and aesthetically. There's no a priori reason why using the states of CA cells as convolution kernels should give aesthetically pleasing results. Obviously, if you were to apply randomly changing, random-valued kernels to individual pixels in an image, the resulting animation would probably be a horrible-looking mishmash. However, the coherence and aesthetic beauty of a CA can produce the same coherence and aesthetic beauty in an image animated by the CA. By using different CA and different integer values for the states, you make the animation as varied and as subtle or garish as you wish.

For example, a direct CA animated convolution, using Life or Brian's Brain, of van Gogh's Sunflower paintings adds a subtle, scintillating effect to the original paintings. If you were to look at a single still image from the animation, you probably couldn't easily distinguish it from the original painting (this is why there are limited screenshots in the article---please try out the applets (here and here) or watch a Quicktime movie of the applets, instead). In the animation as generated by Life or Brian's Brain, the original painting dominates, but it's like viewing a real sunflower as the subtly changing sunlight and perhaps a slight breeze give it animation. References

- Dewdney, A. K. The Magic Machine: A Handbook of Computer Sorcery. Published by W. H. Freeman, 1990.
- Reas, Casey and Ben Fry. Processing: A Programming Handbook for Visual Designers and Artists. Published by The MIT Press, 2007.
- Reiners, Paul. "Pointillism meets pixelation: Using the Java two-D API to animate art", IBM developerWorks, 18 November 2008.
- Resnick, Mitchel and Brian Silverman. "Exploring Emergence". Published by the MIT Media Laboratory, 1996.
- Whitelaw, Mitchell. Metacreation: Art and Artificial Life. Published by The MIT Press, 2004.