A Fast Algorithm for Creating Turing-McCabe Patterns

Markus Schwehm
Institute for Exploratory Systems
ExploSYS GmbH, Otto-Hahn-Weg 6
70771 Leinfelden-Echterdingen, Germany
markus.schwehm@online.de

Abstract

A GPU algorithm can generate Turing-McCabe patterns significantly faster, and with more parametrized variety than previously described. After tying the algorithm to the user capture function of a Kinect, an interactive art installation and performance was created and shown in museums and art galleries in Ludwigshafen, Germany.

Introduction

Turing-McCabe patterns offer intricate detail within a large scale range (Figures 1 to 3). The pattern generation process relies on reaction-diffusion as commonly used to model chemical, physical and biological phenomena. McCabe did not try to mimic a particular natural process. His reaction-diffusion system uses just one substance acting at the same time as activator, inhibitor and pigment. He connects a cascade of activator-inhibitor pairs with varying diffusion gradients and selects just one of them to fire a reaction. Despite this lack of similarity to any real natural process it is astonishing how much his patterns resemble life-like artifacts like, for example, electron microscope images of cellular tissues.

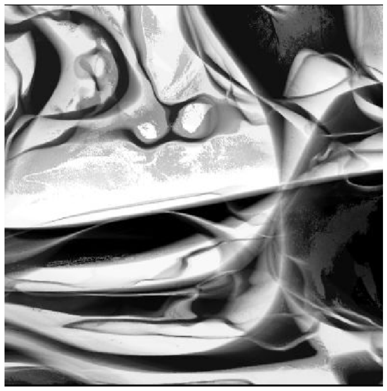
Figure 1: *Coarse pattern*

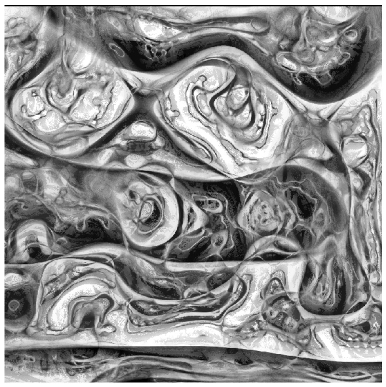
Figure 2: *Standard pattern*

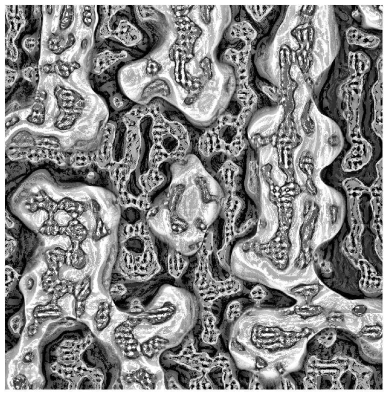
Figure 3: *Fine grained pattern*

It is possible to find several examples of these Turing-McCabe-Patterns on the web [2]. There are images and videos as well as some sites that allow the viewer to play around interactively with Java-based applets. It is possible to dive into the intricate details of high resolution patterns as well as to observe the evolution of the patterns over time. However some of the videos that can be found on the web used hours of rendering time for just a few minutes of video outcome [3]. This is especially unsatisfying if one tries to interact with the pattern process – this is so far only possible with very low resolution images. There exist web sites that allow interactive exploration of Turing-McCabe patterns. Some of these sites are difficult to use today because they require a Java plug-in that is no longer supported by current browsers, for example [4].

We wanted to use the Turing-McCabe patterns in an art project. Would it be possible to generate these patterns in Full HD with enough frames per second to allow interaction with the pattern generation process in a live performance?

The Basic GPU Algorithm

Given the available hardware it was clear that the algorithm would need to be implemented on a GPU. Since we wanted to have this algorithm eventually running within a web application, we used WebGL for this implementation.

The pattern generation process is controlled by the number of `levels` to be computed and a `stepsize`. For each level, we need to define a `radius` (corresponding to a diffusion coefficient of the substance) and a `step` (corresponding to the reaction rate for production or inhibition of this substance). Beautiful patterns emerge if the radii and steps are evenly distributed on a logarithmic scale. The range of radii can be controlled using the `minRadius` and `maxRadius` parameters (see Figure 4).

The GPU algorithm consists of a set of shader programs that look up values in some textures and write out results to other textures. The algorithm uses the textures *pattern*, *activator*, *inhibitor*, *integrator* and *minimizer* with the size of the target screen (1920 × 1080 pixels). The algorithm (Figure 5) is started with a randomly generated *pattern* using decimal values between −1 and 1. In each step of the algorithm, the *minimizer* is initialized with ones and the *pattern* is copied into the *activator*. For each level the *activator* needs to be blurred with the precomputed level radius `radius[l]` using a box blur algorithm. To accomplish this, the *activator* is copied into the *integrator*, integrated horizontally and vertically and then the blurred value can be looked up using the given `radius[l]` and stored in the *inhibitor*. Now we compute the difference between *activator* and *inhibitor*. If the absolute value of this difference is smaller than any previous difference stored in the *minimizer*, we store this new difference together with the current level in the *minimizer*. Because the difference is always below 1, it is possible to store the level in the integer part of the *minimizer* and the new minimum difference in the fraction part of the *minimizer*. At the end of the loop, the *inhibitor* is copied into the *activator* such that it can serve as activator in the next loop. After execution of the last loop, the *pattern* can be updated by looking up the smallest difference in the *minimizer*. However the update does not use this minimal difference, but uses the pre-computed `step[l]` for the level that had computed this smallest difference. Finally the resulting *pattern* needs to be normalized to keep the values between −1 and 1. To save computing time, we replaced this normalization step with the multiplication by a factor that can be controlled during the performance, such that we can have additional control over the contrast of the resulting pattern. The resulting pattern is then copied into the video buffer for display on the screen. Using this algorithm, it was possible to generate six Full HD frames per second.

```
levels = 15
stepSize = 0.005
minRadius = 0.0
maxRadius = 0.8
factor = 0.06
maxLength = min(width, height)
delta = 1-minRadius / (levels-1)
for (l : levels) {
    r = MinRadius * l * delta
    radius[l] =
        floor(exp(r*log(maxRadius*maxLength)))
    step[l] = (log(r)+1) * stepSize
}
```

Figure 4: *Parameters*

```
minimizer ← init(1)
activator ← copy(pattern)
for (l : levels) {
    integrator ← copy(activator)
    integrator ← integrate_horizontal()
    integrator ← integrate_vertical()
    inhibitor ← lookup (integrator, radius[l])
    minimizer ← minimize(activator, inhibitor)
    activator ← copy(inhibitor)
}
pattern ← update(minimizer, step)
pattern ← normalize(factor)
show(pattern)
```

Figure 5: *Pseudocode*

Using All Color Channels to Increase the Frame Rate

Six frames per second is not fast enough for an interactive performance. As described, the algorithm generates only monochrome images and thus uses only one of the four color channels (R, G, B and alpha) of the GPU. In order to increase the frame rate further, we had to use all color channels by splitting the image into four equal sized segments and to computing each segment in a separate color channel. Most of the code could stay the same, only the `integrate_horizontally` and `lookup` methods become somewhat more complicated to deal with the borders of the segments. We called this variant of the algorithm "Folded." It generated slightly more than 20 frames per second, enough to be used in our live performances.

Running several pattern generation algorithms in parallel

An alternative to using the four color channels of the GPU was to run full-sized pattern algorithms in parallel in the four color channels. This would not speed up the algorithm but it allowed us to play around with colors. Running three independent processes for the three color channels with no color coupling or very strong color coupling did not yield interesting results (Figure 6 and Figure 8). Very colorful images were achieved with weak color coupling (Figure 7).

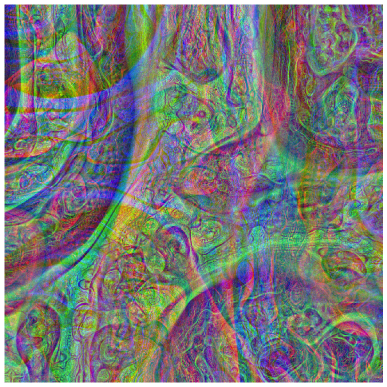
Figure 6: *No color coupling*

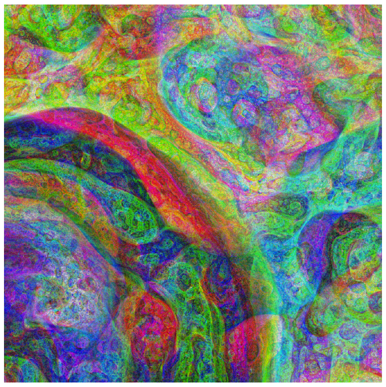
Figure 7: *Weak color coupling*

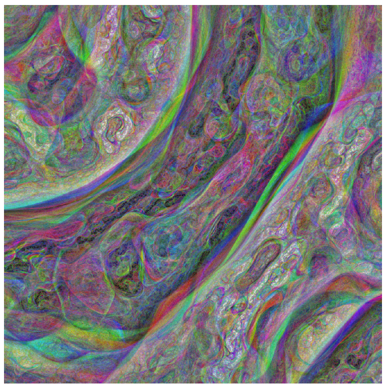
Figure 8: *Strong color coupling*

Interaction by Injecting Kinect User Capture Data

For a performance we wanted some interaction between visitors and the pattern generation process. Using a video camera did not work well, since it is difficult to extract body shapes and gestures from a video stream (Figure 9). Instead we used a Kinect [5] to capture a depth image of a scene that would allow us to isolate a body shape (Figure 10). We used the shape of all captured users to interact with the pattern generation process by making the pattern darker within the extent of a detected user. The Kinect data was streamed to the browser, where pattern and user shapes were merged (Figure 11).

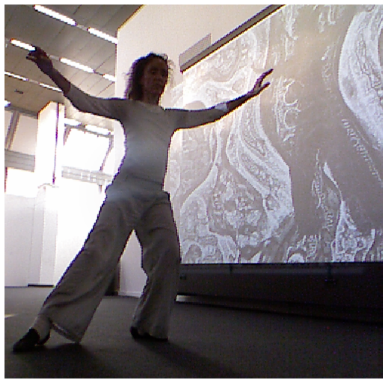
Figure 9: *Video image*

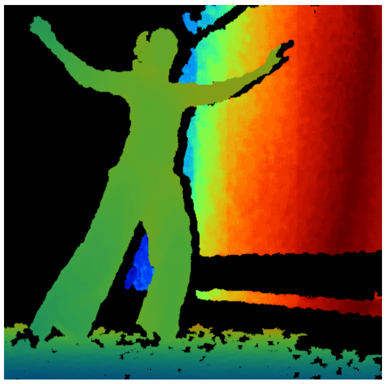
Figure 10: *Depth image*

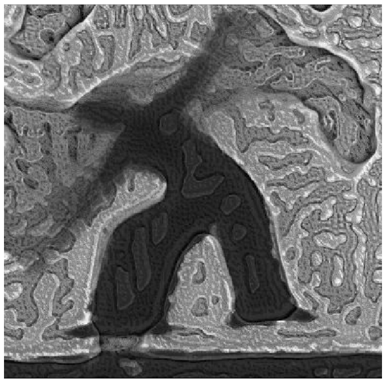
Figure 11: *Generated pattern*

Art Performances

A first version of this pattern generation algorithm was used in an art performance together with Janna Schimka (dance and vocals, Figure 13) and Rolf Schmuck (electronic music). The performance took place during a summer art festival in a vacant shop in the pedestrian area of Ludwigshafen 2013. Visitors could enter or leave the performance at any time, sometimes being captured by the Kinect and therefore influencing the pattern generation process (Figure 14). But in this first performance the frame rate was not fast enough, so that only patient visitors recognized the interactive potential (Figure 12). More images and some video clips from this performance can be found at [6].

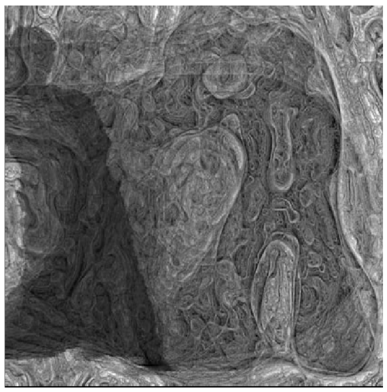
Figure 12: *Dancer*

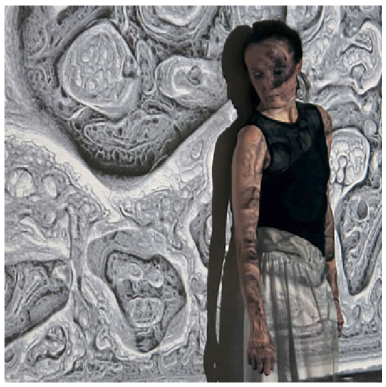
Figure 13: *Photo*

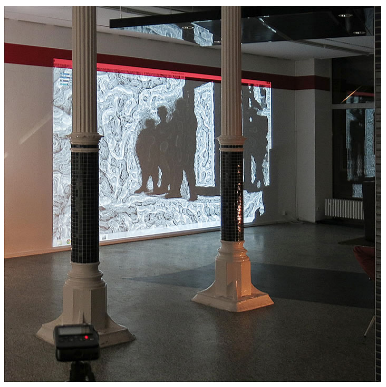
Figure 14: *Audience*

In Summer 2014 we were invited by museums and art galleries in Ludwigshafen to stage four additional performances [7]. This time we had the fast algorithm as described in this paper in place. Each performance took around three to four hours and visitors could enter or leave the performance at any time. The interaction between dancer/visitor and the pattern generation process was immediately recognizable by the audience and allured all participants to playfully interact with this system.

References

[1] Jonathan McCabe, "Cyclic Symmetric Multi-Scale Turing Patterns," Proceedings of Bridges Pécs: Mathematics, Music, Art, Architecture, Culture; G. W. Hart and R. Sarhangi, Eds., 2010, pp. 387-390.

[2] Jonathan McCabe, Artist Homepage. http://www.jonathanmccabe.com/ (as of 30/04/2016)

[3] Cornus Ammonis, McCabeism: Fast Multiscale Turing Patterns (2015). https://www.youtube.com/watch?v=4Sz-iEdNFDc (as of 30/04/2016)

[4] OpenProcessing sketch by user bitcraft: Turing-McCabe Pattern Explorer (2011). http://www.openprocessing.org/sketch/33444 (as of 30/04/2016)

[5] Jana Abhijit, Kinect for Windows SDK Programming Guide, Packt Publishing, 2012

[6] Janna Schimka, Rolf Schmuck, Markus Schwehm, c.temp | 1, Installation & Performance (2013) http://ctemp1.exploratory-systems.de (as of 30/04/2016)

[7] Janna Schimka, Rolf Schmuck, Markus Schwehm, :migration, Installation & Performance (2014) http://m.orbit31.de/ (as of 30/04/2016)