Cell Noise and Processing

Carl-Johan Rosén, Linköping University, 2006

Abstract

The cell noise, discovered by Steven Worley, has potential to be an important tool in creating electronic art. Implementing it in the Processing environment will hopefully result in a wider use of cell noise. How to convert it into Java/Processing and simple usage of the noise is described in this paper, along with some brief introduction to what cell noise is. A lot of effort has been put into promoting the cell noise library to Processing users, and the most important result of the project, next to the library, is the web page describing the cell noise.

Introduction

Processing is a development platform for visual arts, developed by Casey Reas and Ben Fry. It is a open source projekt soon to be released in its first non-beta version. The environment is built up by a core, containing the basic tools for creating visual arts. Besides the core there is the posibility for each user to add libraries containing extended tools. This report describes the process and the result of the development of such an add-on library.

The project is done in the scope of the university course *Proceedural methods for images* at the Linköping University, Sweden. It is also intended to be used by developers and artists in the Processing environment.

The library is an implementation of a method for creating cell noise described by Steven Worley in *Texturing & Modeling – A Procedural Approach*. The implementation is a modification of the code described on page 149 an forward.

Cell Noise

The basic idea of cell noise is to spread a varying number of feature points in space and calculate the distances to these points from every possible point in space. This can then with success be used to simulate, among many things, cell-like structures. As a complement to this basic idea, Worley describes ways to connect each feature point with a unique ID number. This number can be used to generate a surface with solid colored areas, looking a lot like military canvas or glass mosaic, depending on the coloring function.

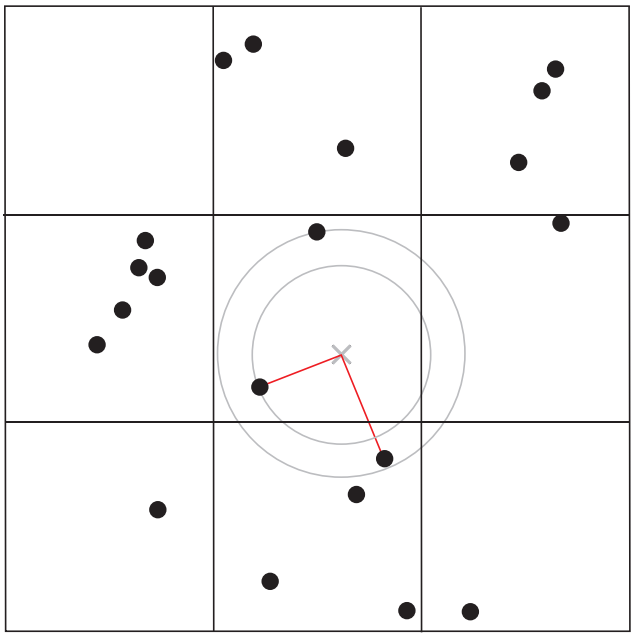

Figure 1. The feature point spread of a typical cell noise. Sample point marked with x.

Figure 1 describes a space with feature points scattered across the plane. For a position on the plane, the *x* in the figure, there is always a closest feature point. There is a second closest as well as a third and so on. The algoritm searches the plane to find them and returns the distance, the relative position and the ID number of each feature point.

Shown in figure 1 are also the integer lines of the plane, that is where the positions are integers in any dimension. The integer squares (cubes in three dimensions) are central in the algoritm.

When we want to know the distance to the two feature points closest to the point **(3.6, 5.7)** we start by looking at the integer square **(3, 5)** and compare its feature points to each other to find the closest. Let's say that the point **(3.6, 5.7)** is the *x* in figure 1. It's obvious that the two feature points in that square aren't the two closest. One of them is, but the second closest belongs to the integer square **(3, 6)**.

This is no problem using the algorithm by Worley. Having found the feature points in square **(3, 5)** the algorithm goes on to compare the position with its integer borders, to determine if it's even possible that there is a closer feature point in the adjacent squares. As seen in figure 1, where the distances to the two points of the center square are marked, it's only possible to find closer feature points in the squares **(4, 5)**, **(3, 6)** and **(4, 6)**. Only these squares have to be searched. The benefit of this is efficiency. The integer line comparison is extremely fast compared to the in-square search.

Since the number of feature points to search for is variable, the number of squares having to be searched varies. The number of feature points in each square is set in the algorithm to be between none and five, and average to 2.5.

For more information on the cell noise algorithm see *Texturing & Modeling – A Procedural Approach*, as mentioned above, or the paper *A cellular texture basis function* by Worley, 1996. There he lays out the principles of cell noise, in both text and example code.

Processing

Processing was released in its first beta version In April 2005 and has since drawn enormous attention considering it not yet beeing released in an official stable version. It is already used in education at many universities around the world, both for teaching art and technology.

Processing is built on Java, but offers a simplyfied syntax to the users new to programming. In fact it is possible to write code in three different levels, where the simplest one takes practically no time to create simple screen output in. The levels above (or below, depending on how you see it) are more complex, but also allows more control. None of these levels is as complicated as standard Java code, and doesn't offer the same control. But for the tasks Processing is intended to perform its simplicity and level of control is well suited.

When an animation or a process is ready it is exported to an applet or an application by a simple click on the expoert button. No preferences has to be set. The applet is complete with a html-page and can simply be copied to a web server, or run locally. Applications are exported for Mac OS X, Windows and Linux.

Developers can distribute their work as libraries to be included with projects. These libraries are written and compiled as a normal Java program, and packed into a .jar archive. The Processing environment recognizes libraries if placed in the correct directory and a user can then easily import the library to create visualizations.

For a deeper understanding of what Processing is, how to use it and to download the software, I recommend *processing.org*, where both texts, code and examples can be found. There is even a gallery of animations and interactive processes created with Processing.

Extensions of the original code

The first and most obvious change to the original code was to interpret the original code from C to Java. Since some of the types and functions used didn't exist in Java, they had to be converted into their Java equivalents.

Modulo as integer overflow

The most complicated part was to convert the use of unsigned integer overflow in C to something similar in Java. Since Java doesn't handle unsigned integers, signed data types had to be used.

But unsigned 32 bit integers handle larger numbers than signed, so the 64 bit integer long type was used in combination with a modulo operation. The function **u32()** was added to the main class, converting a 64 bit signed integer to what it would have been if it where a unsigned 32 bit integer.

CellDataStruct as pointers

Since Java doesn't handle pointers the class **CellDataStruct** is used to pass data to, from and between the functions in the main class. The **CellDataStruct** together with the main class (**CellNoise**) are the two parts of the cell noise package as visualized in figure 2. **CellDataStruct** contains public variables representing all variables passed to and from the main function (named Worley) in the original code.

This class allows using variables as pointers, and not allocating new memory all the time. What we loose is a bit of structural overview, but winning efficiency. During the process only one instance of the class is instanciated, but the variables are constantly changed.

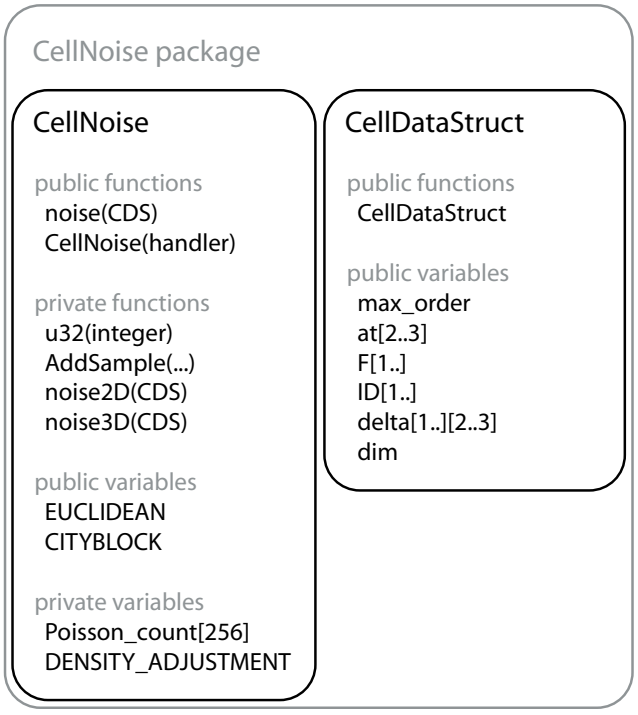

Figure 2. Class structure of the CellNoise library.

Distance measures

As seen in figure 2, there are public constants representing different distance measures available to determine which feature points are closer. The default one, euclidean, is one of four available. The others are city block, manhattan and quadratic distance. Depending on the distance measure, the visualizations character changes.

The main problem with different distance measures is that they usually are quite heavy on the CPU. Except for speed there is nothing hindering devellopers to add more complicated distance measures, like weighted measures.

Three and Two dimensions

In the original code there is only one noise function and it's for three dimensions. Since flow in the animation and frame rate is of great importance in the Processing environment, an extra, simpler, noise function was added. It considers only two dimensions, but still maintain much of the cell like appearence. It is about 2 times faster than the three dimensional noise function using euclidean distance measure.

Figure 3. The feature point spread.

Examples

Here are some examples to show the diversity of possible interpretations of the feature point positions. Some are very simple and illustrates the character of the noise, while some other are more extravagant. But I'm sure there are both more and less extravagant patterns to be explored.

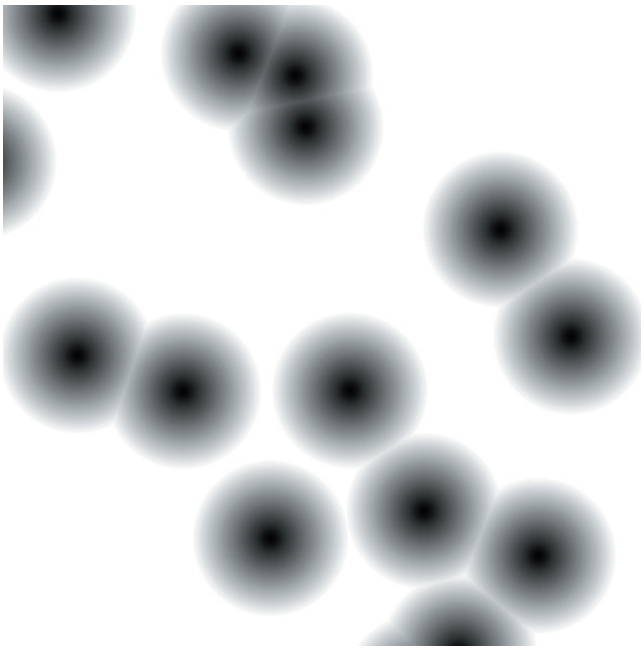

Figure 4. First order distance.

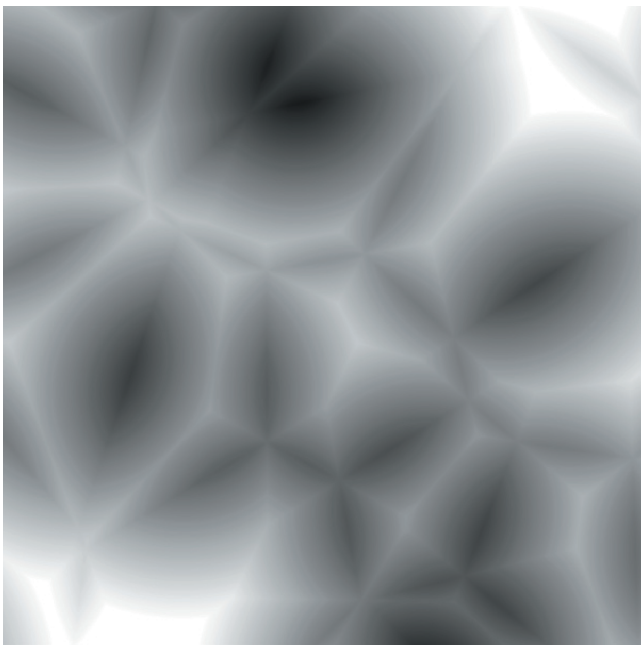

Figure 5. Second order distance.

Point spread

Figure 3 visualizes the spread of the feature points in two dimensions. It's based on a very simple formula where the color of the pixels vary linearly with the distance to the nearest feature point. A part of the code, where the coloration is done, looks like this:

```
cd = CellDataStruct(
    this,
    1,
    at,
    cn.EUCLIDEAN
);
...
cn.noise(cd)
pixels[x + y*width] = color(
    255,
    ((float)cd.F[0] * 2560),
    ((float)cd.F[0] * 2560),
);
```

First order distance

In figure 4 it's fairly obvious why this kind of noise is called cell noise. This is one of the most basic transformations of a distance measure to a color (in this case not even a color, but a grayscale value). The code for this looks like:

```
cd = CellDataStruct(
    this,
    1,
    at,
    cn.EUCLIDEAN
);
...
cn.noise(cd)
pixels[x + y*width] = color(
    (float)cd.F[0] * 350
);
```

Second order distance

This is a linear mapping of the distance to the second nearest feature point on a gray scale. Compared to the first order distance, this has a more abstract look. But still very interesting. Code below:

```
cd = CellDataStruct(
    this,
    1,
    at,
    cn.EUCLIDEAN
);
...
cn.noise(cd)
pixels[x + y*width] = color(
    (float)cd.F[1] * 120
);
```

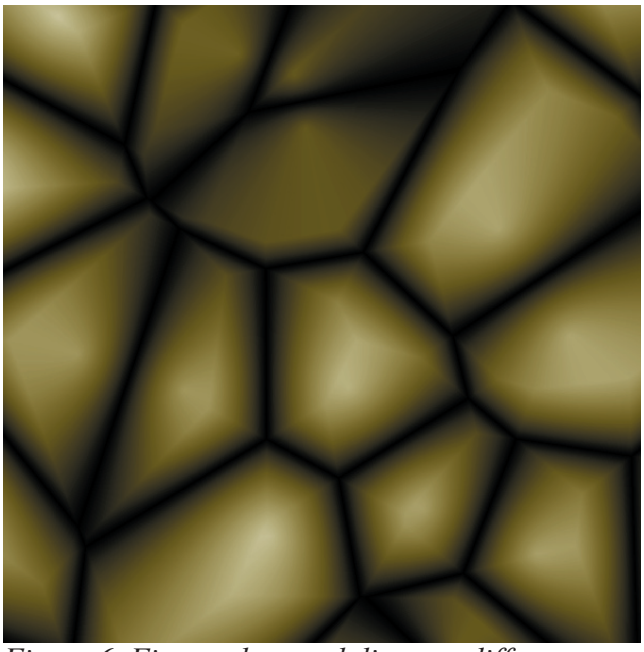

Figure 6. First and second distance difference.

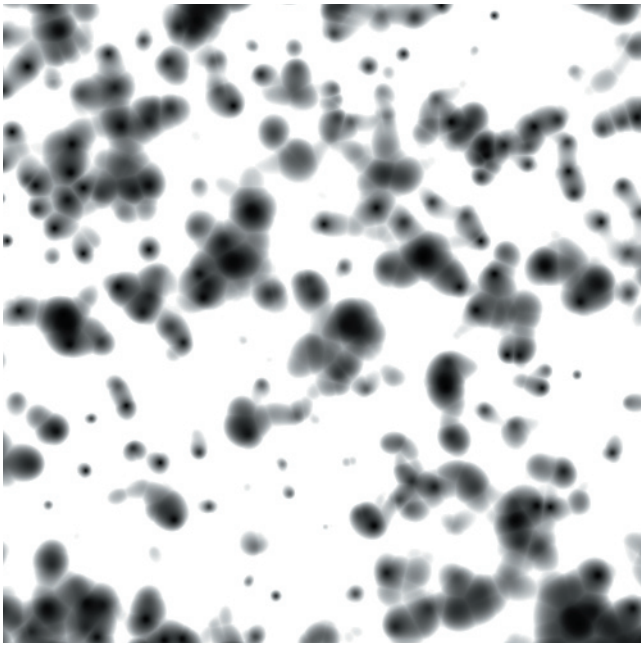

Figure 7. First and second distance multiplied.

First and second distance difference

This is the plot of the difference between the distance to the second order feature point and the distance to the first. It creates this iceberg-like structure.

```
cd = CellDataStruct(
    this,
    1,
    at,
    cn.EUCLIDEAN
);
...
cn.noise(cd)
pixels[x + y*width] = color(
    (float)(cd.F[1]-cd.F[0]) * 120,
    (float)(cd.F[1]-cd.F[0]) * 120,
    (float)(cd.F[1]-cd.F[0]) * 30
);
```

Fractal multiplication

Using many sets of noise generations and multiplying them renders a interesting cell like structure. As is visible in the code, there are many parameters available to adjust. Also see the animation of this pattern on the web page.

```
cd = CellDataStruct(
    this,
    1,
    at,
    cn.EUCLIDEAN
);
...
cn.noise(cd)
double sum = 1;
for (int i = 0; i < 4; i++) {
  at[0] = 0.01*(i*2+1) * (x+20);
  at[1] = 0.01*(i*2+1) * (y+700);
  cd.at = at;
  cn.noise(cd);
  sum *= (cd.F[0]);
}
pixels[x + y*width] = color(
    (float)(sum)*255
);
```

ID number coloring

Each feature point has a unique ID number. Using the ID number of the closest feature point as the color value renders a mosaic of color areas. In this case the manhattan distance measure is used to determine which feature point is the closest.

```
cd = CellDataStruct(
    this,
    1,
    at,
    cn.MANHATTAN
);
...
cn.noise(cd);
pixels[x + y*width] = color(
    (cd.ID[0] % 255),
    (cd.ID[0] % 155),
    (cd.ID[0] % 100)
);
```

Noisy noise

Letting the noise decide the amplitude of another noise will generate something like figure 9. Or something compleately different. This is a pattern discovered by chance by combining one noise result with another, letting the frequency of the second noise be dependent of the first. In code it looks like this:

```
cd = CellDataStruct(
    this,
    1,
    at,
    cn.EUCLIDEAN
);
...
cn.noise(cd)
at[0] = 0.01*cd.F[0] * (x + 20);
at[1] = 0.01*cd.F[0] * (y + 700);
cd.at = at;
cn.noise(cd);
pixels[x + y*width] = color(
    (float)(cd.F[1])*150,
    (float)(cd.F[0]+cd.F[1])*50,
    (float)(cd.F[0])*10
);
```

I imagine this looking really nice beeing animated. But i was not able to try it since my computer managed only a framerate way too low.

Figure 8. ID number coloring.

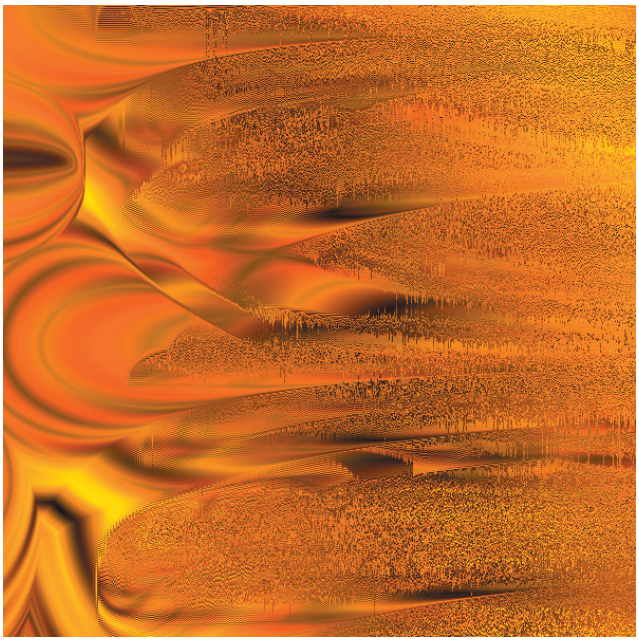

Figure 9. Noisy noise.

Lines and gravity

This example, figure 10, is a very good example on how cell noise can be used in Processing in a less conventional way. Here it's not used to generate a texture, but rather as a gravity field. Lines are shot randomly into the frame using the **random()** function in Processing. Of course the cell noise function could have been used for this as well,

Figure 10. Lines and gravity.

but I like the fact that the animation is new every time. The code for this animation is too long to put in this report, but both the code and the animation are available on the web page.

Promoting the library

One part of this project is to promote the use of the cell noise in the Processing environment. The most efficent means of promotion for something like a Processing library is probably a web page. A web page including background theory, interesting images and patterns combined with explanations and code. Such a web page has been set up at *www.student.itn.liu.se/~carro360/processing*.

On the page users will find the examples described above plus a few more. They are also viewable, together with its code. Users can read about the theory of cell noise, see the library code along with the original code by Worley.

Listed at *processing.org* are some of the basic tools for generating graphics. Often with good examples and cool animations. It's probably not easy, as a new developer, to get the opportunity to add a tool, but the cell noise library is sent to the main developer for consideration. Hopefully he'll find it interesting enough to put it up at the *processing.org/reference/libraries*.

But there is also a public forum for diskussion at *processing.org/discourse*. The forum is full of lively diskussions about tools and techniques used by artists and developers. The cell noise now has its own thread in the forum describing its advantages and will hopefully be shown interest.

Discussion

The project has been very interesting and I believe Processing is a good environment to explore the possibilities of cell noise in. If developers and artists can be brought to understand the noise I think they will take it places we didn't know it could go.

The problem, and I will continue that work even after this project is formally ended, is to spread the understanding of the cell noise. To not scare anybody away, I've tried to tune down the level of programming discussions on the web page. It's better if the focus is on what the tool can do, rather than on how it does it.

The promotion of the cell noise library will, as mentioned, be done at the web page. From now on I will publish what I do with Processing/CellNoise on that page. My wish is that it will inspire creative people to make new interpretations of the cell noise.

Litterature

Ebert, Musgrave, Peachey, Perlin & Worley, (2003). *Texturing & Moduling - A Procedural Approach*. Morgan Kaufmann Publishers. Third edition.

Fry, Ben & Reas, Casey (2006). *Processing* [www] http://processing.org, 10/3 2006.

Worley, Steven. A cellular texture basis function. *International Conference on Computer Graphics and Interactive Techniques* (1996), p. 291 - 294.