

Intro to Visual and Generative Art (In Processing)

Course Notes by Nathan Lachenmyer

Last Updated: March 6, 2012

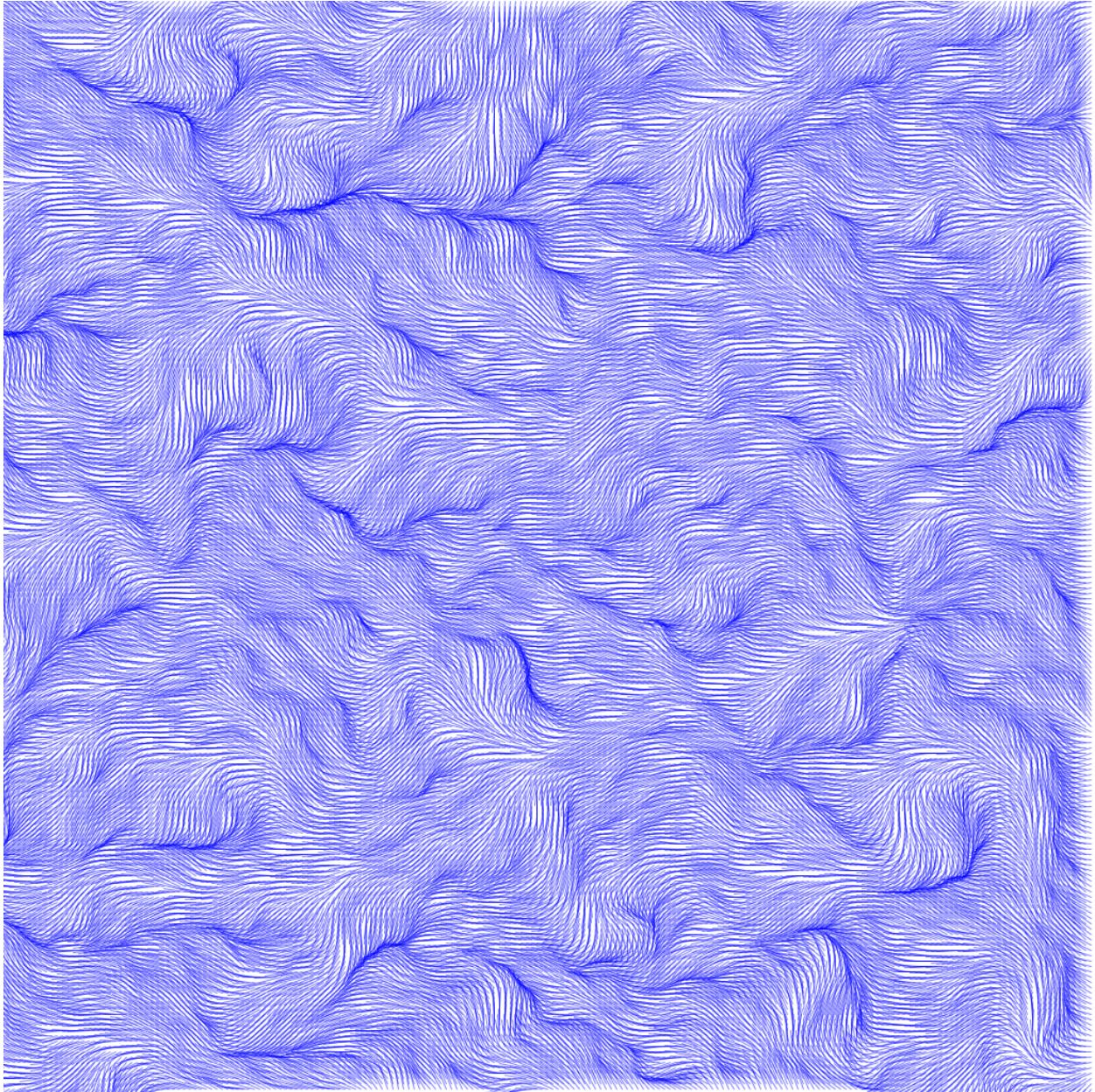


Figure 1: Perlin Noise Visualization by Nathan Lachenmyer

Contents

1	2D Drawing and Control Flow	3
1.1	Introductory Stuff	3
1.1.1	What is Processing?	3
1.1.2	Resources	3
1.2	First Sketch	3
1.2.1	Hello World	3
1.2.2	Example: Simple Droplet	4
1.2.3	Programming Tasks: Drawing Primitives	5
1.3	Introductory Control Flow	6
1.3.1	Example: Simple Droplet (again)	6
1.3.2	Programming Tasks: Rectangles I	7
1.4	Animating Your Sketch	8
1.4.1	Example: Simple Droplet II	8
1.4.2	Programming Tasks: Rectangles II	8
1.5	Saving Your Work	9
1.5.1	Still Images	9
1.5.2	Sharing Sketches	9
1.6	Homework Project – Spiral	9

1 2D Drawing and Control Flow

1.1 Introductory Stuff

1.1.1 What is Processing?

Processing.org really explains what processing is best:

Processing is an open source programming language and environment for people who want to create images, animations, and interactions. Initially developed to serve as a software sketchbook and to teach fundamentals of computer programming within a visual context, Processing also has evolved into a tool for generating finished professional work. Today, there are tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning, prototyping, and production.

Processing is a simple yet powerful programming language that can be used to create breathtaking pieces of visual and generative art. The key to Processing is that it forces you to think algorithmically and mathematically about art – making it ideal for pieces that utilize complexity and randomness. It also has a variety of versatile libraries that allow it to easily integrate with Arduino, sensors, and a variety of inputs (including audio and visual inputs!). Processing is awesome!

1.1.2 Resources

- <http://processing.org/> – the homepage of Processing, with lots of documentation and tutorials available
- <http://openprocessing.org/> – a site for sharing Processing sketches. This is a great place to post your own sketches, and find ideas to influence your own sketches.
- <http://stackoverflow.com/questions/tagged/processing> – StackOverflow is a free Q&A website for programming questions. Anyone is free to ask a question, and the community will try to help you solve your problems.
- <http://cemmi.org/index.php/forum/index> – Feel free to discuss any work related to this course in the Education forums, and any personal projects in the Programming forums. The CEMMI forums are a great way to contact other people in the class, and other people in the Greater Camberville Area that are interested in Processing.

1.2 First Sketch

1.2.1 Hello World

In the Processing Development Environment, type:

```
ellipse(25, 25, 50, 50);
```

Don't forget the semicolon! Then click, Run (the first round button that looks like 'Play'). An circle should show up in a separate window. Congratulations, this is your first sketch! Welcome to the World of Processing!

`ellipse` is a *function*; it is a piece of (prewritten) code that performs a certain task. It takes four *arguments* (or *parameters*) – `x`, `y`, `width`, and `height`. In this case, we placed the center at (25,25) with a width and height of 50. Processing has literally thousands of function pre-written; all of these are made to make expressing your creative ideas as easy as possible!

Rather than go through everything you might possibly want to know about Processing, let's go through a simple example.

1.2.2 Example: Simple Droplet

This example is called ‘Simple Droplet’ – you’ll see why later! But for now, bear with me as we start from the beginning. Copy and paste the following code into the Processing IDE:

```
int diam = 100;
float centX, centY;

void setup(){
  size(500,300);
  background(64,0,196);
  centX = width/2;
  centY = height/2;
  stroke(0,0,255);
  strokeWeight(1);
  fill(255,25);
  ellipse(centX,centY,diam,diam);
}
```

Before `setup()`, I have two lines of code declaring *variables*. Variables are names that we can assign an arbitrary value; it’s extremely useful if we want to use a certain number (like 100) multiple times in the code. By making it a variable, we can change only one line of code to change the value of 10 everywhere we specified its variable name! Variable initialized outside of a function are *global variables*; these are variables that don’t just exist within a function (like `setup()`); they exist throughout your entire Processing sketch. This becomes extremely important later when we start using multiple functions for our sketches, including the `draw()` function to animate our sketches.

When you ‘initialize’ a variable in Processing, you must:

- Give the variable a name (like ‘diam’)
- Give the variable a type (like ‘int’)
- Initialize the variable to a value (like 100). If you don’t specify a value, the variable initializes to 0.

For now, we will only be using the data types ‘int’ and ‘float’. ‘int’ specifies an integer (like 1, 2, 3, -1, 10), and ‘float’ specifies a floating point (decimal) number. Use the ‘int’ variable type when you’re counting things or defining pixels, and use the ‘float’ variable type when doing math (or else you’ll get unexpected behavior – like $1 / 1.5$ is zero!).

Next comes the `setup()` function. It takes no arguments – this is why the parenthesis following it are empty. However, even if it has no arguments, you still have to provide the parenthesis! Everything within this loop will run exactly once during your sketch – before the sketch window launches, in fact. This is a good place to put stuff that isn’t meant to repeat, or that configures aspects of your window – such as its size and background color.

`size()` is a function that takes two arguments – the desired *width* and *height* of your sketch, measured in pixels. It sets the size of your ‘palette’, so to speak. It also creates the *local* variables `width` and `height` – these can only be used in `setup()`, because they are *localized* to that function (to the computer, this means that it can throw these variables away once it is done with `setup()`). If you want to use them later on in your sketch, you’ll have to store them in a global variable.

`background()`, `stroke()`, and `fill()` are all functions that take color arguments and set the color of various parts of your sketch. `background()` sets the background color, while `stroke()` and

`fill()` set the color of the shapes you draw (until you change it with another `stroke()` or `fill()` command).

There are two ways to define color in Processing: in greyscale (black and white) or RGB color¹. In greyscale color, 0 represents black and 255 is white; every integer value between 0 and 255 is a shade of grey. If any of the above functions are called with just one number, Processing assumes it is greyscale. On the other hand, you can give Processing *three* colors (such as `stroke(0, 0, 255)`) in which case it interprets the numbers as values for **R**ed, **B**lue, and **G**reen (hence, RGB). In RGB color, 0 means that the color is not present at all, while 255 means that the color is maximally present. Both greyscale and RGB color modes support an optional extra option to set the transparency – 255 means the object is fully opaque, while 0 is fully transparent.

There are a variety of other parameters that can be set to affect the way your drawing looks. A few useful ones include:

- `strokeWeight(weight)` – sets the thickness of your stroke lines (weight is between 0 and 255).
- `noStroke()` – removes the stroke from your drawings
- `noFill()` – removes the fill from your drawings.

Note that if you remove the stroke *and* the fill, all of your shapes will disappear!

There are four primitives shapes in Processing that you will use in most of your sketches. They are:

- `point(x, y)` – x,y draws a single point
- `line(x1, y1, x2, y2)` – draws a line from (x₁, y₁) to (x₂, y₂).
- `rect(x, y, width, height)` – draws a rectangle with given width and height, with the top-left corner at (x,y).
- `ellipse(x, y, rx, ry)` – draws an ellipse centered at (x,y) with axes r_x and r_y.

As you can see, each of these primitives takes arguments of *coordinates*. These coordinates correspond to pixels in your Processing sketch. It helps to think of your sketch as a piece of graph paper, with a pixel at each integer (x,y) location. Now it's your turn to start drawing!

1.2.3 Programming Tasks: Drawing Primitives

Open up `template.pde`. **Make a copy of it**, and use this copy as an outline for your code. If you get stuck, solutions are provided.

1. Draw a rectangle that is the size of your sketch with a solid fill.
2. On top of that, draw two lines going across the diagonals of the rectangle (make these lines a different color).
3. On top of the two lines, draw a transparent circle centered in the rectangle.
4. Finally, draw a point inside the circle but (visibly) off-center.

¹Okay, this isn't necessarily true. If you want to learn more about how Processing handles colors, check out <http://processing.org/learning/color/>

What this exercise should have taught you is that coding every shape you want separately is slow and inefficient. Not to mention, gorgeous pieces of visual art don't come from six shapes – you might want hundreds or even thousands of shapes in a single sketch! We're here to make not just visual art, but *generative* art – we need code that will help us generate patterns, and perhaps even make decisions about the patterns that its generating. *Control Flow* helps us accomplish the task of coding up many shapes in a structured way.

1.3 Introductory Control Flow

In this section we will introduce the concepts of *conditionals* and *loops*, two extremely important concepts of programming. You will use these in nearly every sketch from here on out – so learn them well!

A *conditional* is a statement that asks a question – and receives an answer of whether the statement is True or False. This is done with *operators* which are used to compare two values. Examples are:

- `==` – equal to (a `==` b means a equals b)
- `!=` – not equal to
- `<` – less than
- `<=` – less than OR equal to

conditionals are used to allow your program to make decisions. An example of a decision that can be made with conditionals is an *if statement*:

```
if (conditional){  
  //code to run if the conditional is TRUE  
}  
else {  
  //code that runs if the conditional is NOT TRUE  
}
```

In this loop you put a conditional in parenthesis right after the word 'if'; if this conditional is `True` then the block of code insides of the braces runs. The `else` portion is optional – it presents an option in case the conditional is determined not to be true. If no `else` statement is specified, Processing just proceeds with the code.

In addition to conditionals, we also have *loops* available to use. Loops allow us to repeat commands in a controlled fashion. These are better explained through an example.

1.3.1 Example: Simple Droplet (again)

```
int diam = 100;  
float centX, centY;  
  
void setup(){  
  size(500,300);  
  background(64,0,196);  
  centX = width/2;  
  centY = height/2;  
  stroke(0,0,255);  
  strokeWeight(1);  
  fill(255,25);
```

```
while(diam < 400){  
  ellipse(centX, centY, diam, diam);  
  diam += 10;  
}  
}
```

This code is very similar to the example shown earlier, so let's skip to the interesting stuff. At the bottom we have a `while` loop. This loop takes one conditional statement, and as long as that statement resolves to `True`, then the code inside the braces continues to run. This code starts by drawing an ellipse with diameter 100, and then increases the diameter by 10 after every ellipse it draws until the diameter is 400; at this point Processing exits the `while` loop and then finishes the program.

If we know how many steps we want to take, rather than just loop through the same code until a condition is met, we can use a `for` loop instead. The structure of a `for` loop is as follows:

```
for (start condition; continue condition; iterate condition) {  
  //code to run while iterating  
}
```

Again, an example is probably better than words for explaining this. So let's look at an example:

```
float centX, centY;  
  
void setup(){  
  size(500, 300);  
  background(64, 0, 196);  
  centX = width/2;  
  centY = height/2;  
  stroke(0, 0, 255);  
  strokeWeight(1);  
  fill(255, 25);  
  for(int diam = 100; diam < 400; diam +=10){  
    ellipse(centX, centY, diam, diam);  
  }  
}
```

Our start condition is that `diam` is set to 100; we continue as long as `diam` is less than 400; and each time we complete the contents of the loop we increase `diam` by 10. The contents of each loop is simply to draw an ellipse, so the code ends up alternating between drawing an ellipse, increasing `diam`, drawing an ellipse, etc until `diam` is equal to 400.

You'll notice that `diam` is no longer a global variable – it is declared *inside* the `for` loop, and exists only inside the loop. Outside of the `for` loop, no other bit of code can access what value `diam` is.

Another thing to notice is that both pieces of code (the `for` and `while` loops) achieve the same result – however, the way each loops approaches the problem is slightly different. Feel free to use whichever loop is more intuitive for you!

1.3.2 Programming Tasks: Rectangles I

Now it's your turn to get some practice in with control flow.

Open up `rectangles.pde`. Use this file as a template for these tasks (make a copy!). I highly recommend that you save each completed task as a separate file. If you get stuck, solutions are provided.

1. Using a `for` loop, edit the code such that the rectangles change color from left to right.
2. Using a `if` loop, make the rectangles alternate between two colors.
3. Using a `for` loop, make the rectangles change color according to a sine function.
 - HINT: Think about the possible values of a sine function.

1.4 Animating Your Sketch

So far all of our programs have generated static images. While these are great (and we will continue to use static sketches extensively throughout this class!), creating dynamic animations is where Processing really shines.

`draw()` is the function we use to create animations. Everything within `draw()` is performed for each and every frame (you can set the framerate with `frameRate()`; a `frameRate` of at least 24 is recommended for smooth animations); within `draw` you can also have as many loops or conditionals as you like. Just remember – the more stuff you put in `draw`, the slower `draw` will be! All of the code has to be run for every single frame, so try not to put anything too computationally intensive in there!

Let's look at an example (and now you'll see why I called the earlier sketch 'simple droplet'):

1.4.1 Example: Simple Droplet II

```
int diam = 100;
float centX, centY;

void setup() {
  size(500, 300);
  frameRate(24);
  smooth();
  background(64, 0, 196);
  centX = width/2;
  centY = height/2;
  stroke(0, 0, 255);
  strokeWeight(1);
  fill(255, 25);
}

void draw() {
  if (diam < 400) {
    ellipse(centX, centY, diam, diam);
    diam += 10;
  }
  if (diam == 400) {
    diam = 0;
  }
}
```

1.4.2 Programming Tasks: Rectangles II

These tasks will build upon the earlier programming tasks. Again, solutions are provided if you run into trouble.

1. Animate the rectangles so that they simultaneously fade smoothly from one to another color.
2. Animate the rectangles so that a uniquely colored square moves from left to right.
 - HINT: You may want to use `delay()` (http://processing.org/reference/delay_.html)
3. Animate the rectangles so that the color fades continuously from left to right.
 - HINT: This should build upon the third exercise you did earlier.

1.5 Saving Your Work

1.5.1 Still Images

If you would like to save a still image of your work, just include the line:

```
saveFrame('filename.jpg');
```

Processing can save your image as either a .jpg or .png file (.jpgs are compressed, .pngs are not). Just remember that if you put this in the `draw()` loop, you'll replace that same image for every frame! Therefore, this is best to use for sketches where you will only use `setup()`.

1.5.2 Sharing Sketches

<http://openprocessing.org/>, mentioned above, is an excellent resource for publishing your sketches. It is also the easiest way to share videos of your sketches – the sketch will run in the user's browser, allowing them to see your sketch in full animated glory!

openprocessing has some excellent instructions for how to upload a sketch at <http://www.openprocessing.org/sketch/upload/>.

1.6 Homework Project – Spiral

Each class will have a homework assignment to reinforce the topics learned during class, and give you a chance to practice! This week's project will be to make an animated spiral. Your project should do the following:

1. Have the circle follow a spiral path towards the center of the sketch
2. Use alpha transparency values to create a trail behind the circle
3. Change colors in some fashion

However, if you finish early, there is a lot more you can do! How about trying:

- Make the spiral change speeds.
- Make it change directions.
- Add multiple 'arms' to your spiral.
- Use a shape other than circle – have it spiral in a rectangular or triangular pattern.
- Instead of doing a spiral, make a spirograph generator! (ask me if you want help with this)

I've provided a template in `spiral_template.pde` that draws a single ball moving in a circular ring. Feel free to use it or start from scratch! I'll provide an example solution later this week.

Potential resources:

- <http://processing.org/learning/trig/>
- http://en.wikipedia.org/wiki/Polar_coordinate_system
- <http://mathworld.wolfram.com/Hypocycloid.html>

When you're done with your sketch, load it onto <http://openprocessing.org/> and post it in the CEMMI Forums under 'Show and Tell' (located here – <http://cemmi.org/index.php/forum/25-show-and-tell>).