Intro to Visual and Generative Art (In Processing)

Course Notes by Nathan Lachenmyer

Last Updated: March 20, 2012

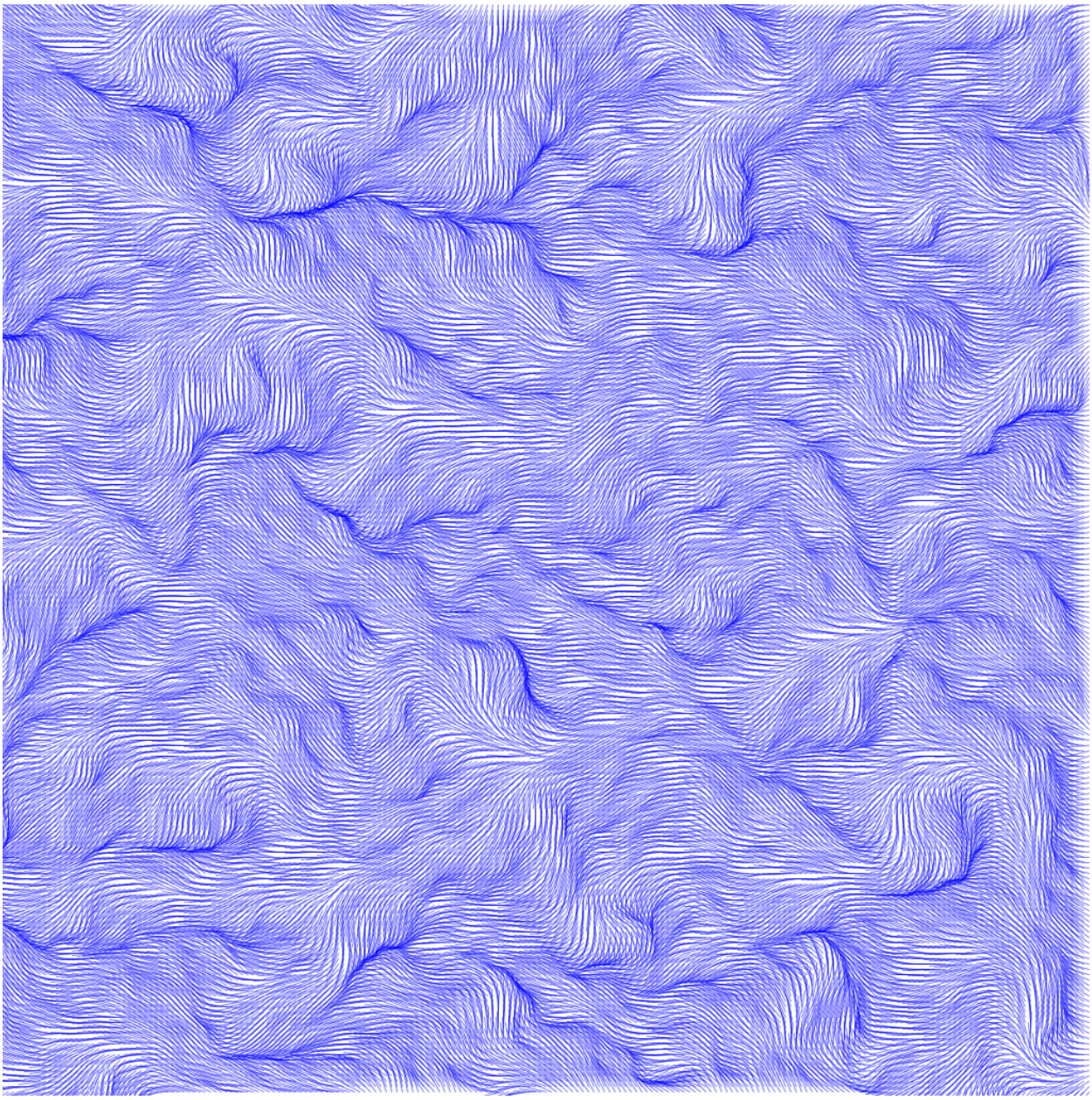

Figure 1: Perlin Noise Visualization by Nathan Lachenmyer

Contents

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 specificied 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 color[1]. 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(`x_1, y_1, x_2, y_2`)` – draws a line from (x_1, y_1) to (x_2, y_2).
- `rect(x,y,width,height)` – draws a rectangle with given width and height, with the top-left corner at (x,y).
- `ellipse(`x, y, r_x, r_y`)` – 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.

[1] Okay, 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).

2 Generative Art – Randomness

Last class we covered drawing static images and animations using primitive shapes and control flow; however, as you may have noticed, these produce very 'rigid' shapes that are somewhat unappealing aesthetically. Computers excel at following instuctions accurately; however, it's this accuracy that can also provides an dull mechanical feel to images generated with them. The natural world exhibits plenty of random and emergent phenomena; so we'll follow suit and try to add randomness and complexity to our art to give it more aesthetic appeal.

2.1 Randomness

The simplest way to add randomness is to use the `random()` function that is built in to Processing. `random()` takes a one or two arguments that tell the function what range to generate a random number in. By default, the range is 0 to 1; a single argument produces an upper limit and two creates upper and lower bounds. `random(10)` generates a random number between 0 and 10; `random(20,25)` between 20 and 25. Let's check a quick example:

```
void setup(){
  size(500,500);
  background(255);
  ellipse(random(width),random(height),100,100);
  ellipse(width/2,height/2,random(100),random(100));
}
```

Try running this code a few times. You'll find that one ellipse is always drawn at the center, but with a random size – the second circle is always the same size, but jumps around the screen. We can easily turn this into an animation:

```
void setup(){
  size(500,500);
  background(255);
}

void draw(){
  ellipse(random(width),random(height),100,100);
  delay(500);
}
```

You'll notice I snuck a `delay()` in there – this causes the program to wait a certain number of milliseconds before continuing.

Let's examine how to best utilize random numbers. Start with a simple line:

```
void setup(){
  size(500,100);
  background(255);
  strokeWeight(5);
  stroke(20,50,70,128);
  line(20,height/2,480,height/2);
}
```

And now let's add some randomness to it!

```
void setup(){
  size(500,100);
  background(255);
  strokeWeight(5);
  stroke(20,50,70,128);
  line(20,random(height),480,random(height));
}
```

But again, this isn't very exciting. Despite the orientation being random, the lines is still too mechanical. So how can we make the line less mechanical looking? The answer is noise!

2.2 Noise

Traditionally, noise refers to an unwanted random addition to a signal (like static pickup in a radio antennae). However, for us, noise will be a *wanted* addition to our signal (the primitive shapes we learned to draw last class). However, first, we'll have to relearn how to draw a line:

2.2.1 Example: Noisy Lines

```
float lastY = 50;

void setup(){
  size(500,100);
  background(255);
  smooth();
  strokeWeight(5);
  stroke(20,50,70,128);
  line(20,50,480,50);
  stroke(20,50,70);
  noisyLine(20,50,480,10);
}

void noisyLine(int startX, int startY, int endX, int step){
  for(int x = startX;x < endX;x += step){
    float randomY = random(25);
    line(x,lastY,x+step,startY+randomY);
    lastY = startY + randomY;
  }
}
```

You'll see at the bottom I defined a new function called `noisyLine()`. The structure of a function is simple – you give it a type (the type of variable it returns; 'void' in the case of a shape), a name ('noisyLine'), and a list of arguments and their types. Then you type a list of instructions just like you would in `setup()` or `draw()`, and every time you call this function, all of those instructions run.

This function breaks up our 'line' into many smaller lines. We start at the point `(startX,startY)`, and from there we do the following:

1. Add `step` to the current value of `x`.
2. Add a random number between 0 and 25 to the previous value of y (`lastY`).
3. Draw a from the previous `x,y` to the new `x,y`.

4. Save the current random `y` as `lastY`.

5. Repeat until we've reached `endX`.

When we make a line like this, we have the opportunity to add noise at a specified interval. The line is starting to approach the aesthetic we're looking for!

However, you'll notice that `noisyLine` always generates a line below the reference line. This is because our noise is always positive (it's between 0 and 25!). If we wanted both positive and negative noise, we'll have to do something a bit trickier.

```
void noisyLine(int startX, int startY, int endX, int step){
  for(int x = startX;x < endX;x += step){
    float randomY = random(50) - 25;
    line(x,lastY,x+step,startY+randomY);
    lastY = startY + randomY;
  }
}
```

We need to generate a random number larger than the size we want, and then we subtract a number from it! In this instance, we'll get noise between -25 and 25 – if the random number generator gives us 50, `randomY` gives us 25; if the generator gives us 0, then `randomY` returns -25. In general:

```
noise = random(maxNoise - minNoise) + minNoise
```

We've taken yet another step closer to that organic aesthetic we're looking for! Now the noise causes our shape to sway back and forth along the rigid line it would otherwise trace. However, this noise is still awfully jagged... wouldn't it be nice if it was a bit smoother? Enter Perlin Noise!

Perlin noise is a random-number generating function that was designed to create natural-looking textures in computer graphics[2]. It generates a sequence of numbers that have just enough variance to look seemingly random, but are closely enough packed together to give the appearance of a smooth change from point to point. Processing already has Perlin noise implemented as `noise()`. `noise()` takes a series of 'seed' values, and then generates a sequence of psuedo-random numbers in the range 0 to 1. A few notes on its use:

- In order to get the smoothest noise, you'll want to vary the seed value incrementally. Best results are when you vary the noise by a small amount each time – less than 0.1 gives nice, smooth variations.

- `noise()` generates a number between 0 and 1; you can't change this. However, you can generate the same type of positive and negative noise by using:

  ```
  noise = (maxNoise - minNoise)*noise(seed) + minNoise
  ```

  This will properly scale your noise.

- Perlin noise is implemented with a pre-determined sequence of noise values; therefore, you'll get the best behavior by using a random starting seed each time you run it.

An example of Perlin noise (see `perlinLine`):

[2]If you want to learn about how it works, check out `http://mrl.nyu.edu/~perlin/`

```
void noisyLine(int startX, int startY, int endX, int step){
  for(int x = startX;x < endX;x += step){
    seedY += 0.2;
    float randomY = 50*noise(seedY) - 25;
    line(x,lastY,x+step,startY+randomY);
    lastY = startY + randomY;
  }
}
```

2.2.2 Programming Tasks:

1. Make a noisyEllipse function.
2. Add positive-only random noise.
3. Add positive/negative random noise.
4. Add Perlin noise to the sketch.
5. Modify the radius, color, size, thickness – whatever you want!

2.3 Applications of Noise

2.3.1 Noisy Particle

```
float r = 150;
float theta = 0;
float theta_vel = 0.02;
float rad_vel = 7;
float centX, centY;
float radNoise;

void setup() {
  //setup the sketch parameters
  background(255);
  size(500, 500);
  frameRate(30);
  smooth();
  centX = width/2;
  centY = height/2;
  //seed noises
  radNoise = random(17);
}

void draw() {
  fill(0,43);
  rect(0,0,width,height);
  //increment noise
  radNoise += 0.1;
  r += (noise(radNoise) - 0.5)*rad_vel;
  theta += theta_vel;
```

```
  float x = r*cos(theta) + centX;
  float y = r*sin(theta) + centY;
  //setup the parameters for drawign the ellipse
  noStroke();
  fill(255*noise(x),255,255*noise(y));
  //draw the ellipse
  ellipse(x, y, 16, 16);
  //update the position for the next frame
}
```

A simple modification of our friendly particle from last class. He's still moving in a circle; however, his path is being modulated by Perlin noise! Notice how it makes his path look much more random and natural. The color of our particle is also beign modulated by the noise values – it's an extremely subtle but pleasant effect.

2.3.2 Example: Noisy Spiral

```
float rad = 0;
float theta = 0;
float centX, centY;
float lastx = 250;
float lasty = 250;
float seed = random(17);

void setup() {
  //setup the sketch parameters
  size(500, 500);
  background(255);
  //object color properties
  stroke(30, 50, 70, 128);
  noFill();
  strokeWeight(5);
  smooth();
  //setup some reference variables
  centX = width/2;
  centY = height/2;
  //draw a reference circle
  ellipse(centX, centY, 100, 100);
  //now draw the part-by-part circle
  stroke(30, 50, 70);
  fill(255,3);
}

//-----------------Main Loop
void draw() {
  //rect(0,0,width,height);
  seed += 0.5;
  rad += 20*noise(seed) - 5;
  theta += 5;
```

```
  float x = centX + rad*cos(radians(theta));
  float y = centY + rad*sin(radians(theta));
  line(lastx, lasty, x, y);
  lastx = x;
  lasty = y;
  if(abs(rad) > 250){
    rad = 0;
    lastx = centX;
    lasty = centY;
  }
}
```

Here's an example of putting it all together. This sketch draws a noisy spiral by increment radius by a noise value, and drawing a line between the previous `(r,theta)` pair and the newly generated pair. The noise can be positive or negative, which results in the spiral lines expanding and contracting, and sometimes even crossing each other. The spirals are drawn piecewise in each frame, giving the appearance that the spiral is expanding outwards. This is a case of where *not* writing a function is useful – if I wrote a function to draw the entire spiral, than each frame would draw an entire spiral and I couldn't see the spiral expand outwards!

Now try uncommenting the line that draws the transparent rectangle (and play with the final `fill()` command in `setup()`!). You can get a variety of effects just from playing with the transparency, ranging from individual particles expanding outwards to complex superpositions of spirals. This is what is commonly called in generative art a *particle emitter* (we'll take about these more in our next class).

2.3.3 Example: Noise Visualization

```
float delta = 0.05;
float xnoise = 0;
float ynoise = 0;
float xstart = 0;
float len = 0;

void setup(){
  size(500,500);
  background(255);
  smooth();
  //frameRate(24);
  xnoise = random(17);
  xstart = xnoise;
  ynoise = random(17);
  noStroke();
  for(int x = 0; x < width; x+=5){
    ynoise += delta;
    xnoise = xstart;
    for(int y = 0; y < height; y+=5){
      xnoise += delta;
      len = 10*noise(xnoise,ynoise);
      fill(128,0,255);
      rect(x,y,len,len);
```

```
      }
    }
}
```

Here's another example of using noise to generate interesting visual textures. Here we used nested `for` loops to iterate over a 2D grid, and place a rectangle at each point in space with a size corresponding to the noise value. Here you see that we've used two arguments for `noise()` – it can actually support up to three for three-dimensional noise.

We can spice it up further by modifying `fill()` to use a noise-generated argument to determine the color! Another example of a noise visualizer can be found under `angle.pde`; this one changes the angle of the lines based on noise values.

2.3.4 Programming Tasks:

Make your own noise visualization sketch! Feel free to use the provided examples as a template. Some thoughts:

- Color and/or transparency are good places to start if you don't know what parameter you want to feed the noise into
- You can animate the sketch by resetting your starting noise seed at the beginning of every `draw()` cycle (these correspond to `xstart` and `ystart` in the example sketches).

2.4 Project – Animated Noise

Take either the Noise Visualization or one of the NoisyParticle sketches and expand on it! The final sketch should be:

- Animated
- Utilize Perlin noise
- Be posted to `http://openproccesing.org/`

We'll (briefly) share our sketches at the next class to share some ideas!

3 Generative Art – Complexity and Emergent Phenomena

Emergence, in the sense of generative art, is the diametric opposite of randomness. Where randomness is chaos – the lack of order, and presence of the unexpected – emergence is a strict adherence to order. Emergence is the the phenomena where a simple set of local rules create a global order. An example is a school of fish – each individual fish has a small number of rules that it follows based off of its immediate neighbors. But when taken as a whole, the behavior of a school of fish is quite complex.

This section of the course is about how to mimic that behavior in our programs, and how to harness it to create complex behavior. We'll start by learning a new concept: object-oriented programming.

3.1 Object-Oriented Programming

Let's look at an example of a program we should be familiar with.

```
int numCircles = 5;

void setup(){
  size(500,300);
  background(0);
  smooth();
  strokeWeight(1);
  frameRate(30);
  fill(150,50);
  Circles();
}

void draw(){
}

void mouseReleased(){
 Circles();
}

void Circles(){
  for(int i = 0; i < numCircles; i++){
   float x = random(width);
   float y = random(height);
   float radius = random(75) + 25;
   noStroke();
   ellipse(x,y,2*radius,2*radius);
   stroke(0,150);
   ellipse(x,y,radius/2,radius/2);
  }
}
```

This program is pretty easy to follow, with perhaps the exception of `mouseReleased()`. This function just run a command when you release a mouse button (after a click); we'll go into this sort of interactive command in more detail next time. But for now, let's concentrate on the rest of the sketch.

Each time we click, some concentric circles appear in random position in the sketch – the number is determined by `numCircles`, and our `for` loop draws all of the circles for us. But these circles can't

really do much – if we clear the background, these circles are lost forever. And even if we don't clear the background, once we throw these circles onto the screen they're pretty immobile. Object-Oriented Programming solves this problem in a rather elegant way.

In object-oriented programming, we create *classes* that defined *objects*. These classes serve as templates to tell us all of the properties of the object; we can then create as many copies (called *instances*) of the object as we want using that template. Here's an example circle class:

```
class Circle {
  float x,y,radius;
  color lineColor, fillColor;

  Circle() {
    x = random(width);
    y = random(height);
    radius = random(75) + 25;
    lineColor = color(random(255));
    fillColor = color(0,random(255),0);
  }

  void displayCircle() {
   noStroke();
   fill(fillColor);
   ellipse(x,y,2*radius,2*radius);
   stroke(lineColor,150);
   noFill();
   ellipse(x,y,radius/2,radius/2);
  }
}
```

The class is divided into three components: object properties, the constructor, and the methods. The first two lines are the *object properties* – they defined what properties every instance has. Every instance has exactly these properties; no more, no less. In this case, every circle has an x and y position, a radius, and a color for its stroke and fill.

The next chunk of code (starting with `Circle()`) is a function called the *constructor*; this function defines how to create an object. This function tells us that to create a circle, we give it zero arguments, and the parameter are all essentially chosen randomly. You could define this function with arguments (like other functions we've written) to give each circle instance an assigned radius, for example.

The last part (`display()`) defines the *methods* of the object – every object has the ability to run these methods, that can modify itself or its environment. This example method just draws the circle using the data we created with the constructor. Let's look at an example of how to use a class.

3.1.1 Example: Drawing Circles

```
int numCircles = 5;

void setup(){
  size(500,300);
  background(0);
  smooth();
```

```
  strokeWeight(1);
  frameRate(30);
  fill(150,50);
  drawCircles();
}

void draw(){
}

void mouseReleased(){
 drawCircles();
}

void drawCircles(){
  for(int i = 0; i < numCircles; i++){
   Circle circleInstance = new Circle();
   circleInstance.displayCircle();
  }
}

class Circle {
  float x,y,radius;
  color lineColor, fillColor;

  Circle() {
    x = random(width);
    y = random(height);
    radius = random(75) + 25;
    lineColor = color(random(255));
    fillColor = color(0,random(255),0);
  }

  void displayCircle() {
   noStroke();
   fill(fillColor);
   ellipse(x,y,2*radius,2*radius);
   stroke(lineColor,150);
   noFill();
   ellipse(x,y,radius/2,radius/2);
  }
}
```

This should look pretty similar to our initial example! The biggest difference is in `drawCircles();` instead of using a for look to manually draw the ellipses, we are now drawing them by creating a new instance of our circle class, and using its `display()` method to make the circle appear. Note that when we create an instance of the circle, we use the `Circle` type, just like we used `int` or `float`.

Okay, so this seems like a lot of work just to be able to draw circles. What else can we do with this? Now that we've created the circle object, each circle has a life of its own – it has its own size, its own color,

its own identity! So now let's give each circle its own unique *speed* – and have them move around!

3.1.2 Example: Moving Circles

The circle object becomes:

```
class Circle {
  float x,y,radius,velocityX,velocityY;
  color lineColor, fillColor;

  Circle() {
    x = random(width);
    y = random(height);
    radius = random(75) + 25;
    lineColor = color(random(255));
    fillColor = color(0,random(255),0);
    velocityX = random(10) - 5;
    velocityY = random(10) - 5;
  }

  void display() {
   noStroke();
   fill(fillColor);
   ellipse(x,y,2*radius,2*radius);
   stroke(lineColor,150);
   noFill();
   ellipse(x,y,radius/2,radius/2);
  }

  void update() {
    x += velocityX;
    y += velocityY;
    if (x > (width + radius)){
      x = 0 - radius;
    }
    if (x < (0 - radius)){
      x = width + radius;
    }
    if (y > (height + radius)){
      y = 0 - radius;
    }
    if (y < (0 - radius)){
      y = height + radius;
    }
  }
```

Now we've added randomly generated velocities to the object properties, and use these properties to move the circle. The last cluster of `if` statements are just there to make the circles wrap around the screen rather than wander off.

The rest of the code becomes:

```
void setup(){
  size(500,300);
  background(0);
  smooth();
  strokeWeight(1);
  frameRate(30);
  fill(150,50);
  drawCircles();
}

void draw(){
  background(0);
  for(int i = 0;i < circleArray.length;i++){
    Circle circleInstance = circleArray[i];
    circleInstance.update();
    circleInstance.display();
  }
}

void mouseReleased(){
 drawCircles();
}

void drawCircles(){
  for(int i = 0; i < numCircles; i++){
   Circle circleInstance = new Circle();
   circleInstance.display();
   circleArray = (Circle[])append(circleArray,circleInstance);
  }
}
```

The important concept that we've added here is one of an *array*. An array is a list of objects, all of the same type. We could have a list of integers (`int[] intArray = 1,2,3;`), or a list of floats (`float[] floatArray = 1.0,1.2,1.5;`) just as easily as an array of Circles. The array is created by the addition of the brackets at the end. Each element in the array has an *index* (starting at zero!) that labels it; `intArray[0]` retrieves 1 and `intArray[2]` retrieves 3, for example. Each array has a length (given by using the `length` method) and we can *append* items to the array as shown in the `drawCircles()` function above.

Arrays are useful for keeping track of a large number of objects – we will use them regularly to keep track of our objects so that we can recall them, update them, and redraw them in every frame.

3.2 Project: Particle Emitter

Now it's your turn to use classes to make useful sketches! We're going to make a *particle emitter*, which is a very common type of sketch (just Google for the term and see for yourself!). The particle emitter should emit particles from the center of your sketch; and they should move away from it in some predefined way (a spiral? a straight line? a noisy line? you choose!). An outline of the sketch is in `ParticleEmitter.pde`. You'll have to write the entire class, and then implement it – use `CirclesOOP2.pde` as a guide if you're feeling a bit lost.