Producing Art from Algorithms

Michael McEllin, April 2020

So, you want to learn a bit about coding? Everyone says that it is the thing to do these days, and maybe even opens up new employment opportunities.

They are not entirely wrong: there are many roles that say nothing about computing in their job titles where a bit of coding knowledge helps to get things done. I spent my life as a professional physicist in the nuclear industry, but a large part of my daily life involved trying to understand what was going on inside nuclear reactors by looking at large amount of data.

The world is drowning in data: modern computer technology means that we can collect it more rapidly and in large bulk than ever before, but there are not enough people who know how to make some sense out of the flood. We even have a new discipline called Data Science" that can be studied at university, which combines computing, mathematics and statistics. (In fact, many people who graduate in these separate disciplines end up doing data-science-type roles anyway.)

Commercial life is also increasingly moving on-line, and that needs ever more people to build web-sites that stand out from the competition, to both promote sales and gather in the money.

Our factories and warehouses are also being taken over by robots that become increasingly sophisticated year by year, and by some estimates even many of the "driving" jobs in modern economies will be shifted to autonomous vehicles within a decade. You would probably prefer to be in the group who makes and programs robots, not the group who they make redundant.

That is all very well, but the truth is that when you are making your first steps in coding these cutting edge applications seem a long way away, and it can be hard to see how the basic tutorials relate to making semi-intelligent automata. It is much the same as the difference between slogging away at basic five-finger exercises on the piano and the skills of the concert soloist. The gaps are, indeed, large and there are few short cuts but as with learning music, the trick is to find something that gives pleasure and reward at an early stage and keeps us wanting to do more while developing higher levels of skill.

I believe that there is no harm in having a bit of fun while we learn, and a lot of people are finding that you can have a lot of enjoyment by using computer programs to generate art. Furthermore, you can start to produce quite impressive and rewarding results with not a lot of learning. What's not to like!

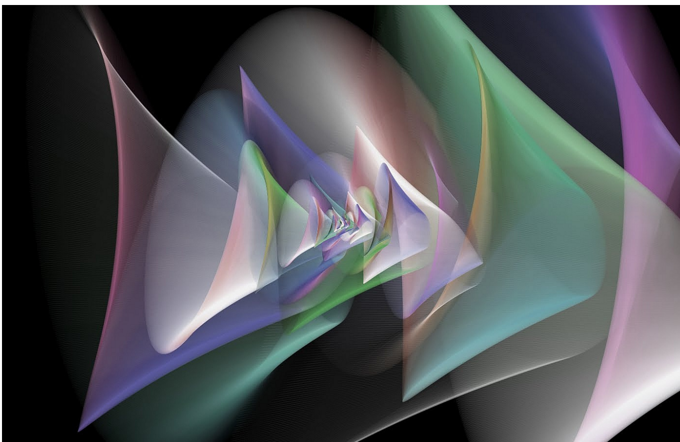
This project will take you from knowing nothing at all, to creating material that you may well want to hang on a wall. We will need a bit of GCSE mathematics, but nothing extreme, just basic trig and maybe Pythagoras. If you doing A-level maths, then you will

have an even wider scope of possibilities, but remember that the quality of the art is more likely to depend on your imagination that your mathematical skills.

The project is a digest of material that can also be found on my personal website *Artful Computing* at https://mcellin.me.uk/artfulcomputing/, which also describes a number of other approaches, ranging from the elementary to those based on university level maths.

Taking a Line for a Walk

The challenge



The image on the left is an example of the type of thing you might choose to produce.

The recipe is simple: we tell the computer to draw a line on the black background. Not a *bright* hard line, but one that is a little bit transparent. then we do it again, but this time we move the position of the line by a little bit, move up or down, side to side and rotate the angle fractionally and perhaps change its colour slightly. And then we do it again, and again and again. Just one addition: as time goes on, we allow the amplitude of the side-to-side movements and the length of the line to decay slowly with time.

Apart from the size decay with time, this is the type of algorithm that was sometimes employed for screen-savers when computer screen were cathode ray tubes and showing a static image could get it permanently burned into the phosphor. The algorithms had to be simple because computers were not very powerful.

So what do we need to do this? There are three elements:

1. We need a graphics application for our computer that lets us draw images.
2. We need to know how to draw a line of a chosen length, angle, position and colour.
3. We need to devise an algorithm, to be programmed in our graphics application, which will move the line around, change its colour and reduce its length over time.

The Tool

I am going to recommend that you install *Processing* (http://processing.org). Processing has several advantages which make it a good choice for first steps:

- It is well supported and widely used, with both good on-line tutorials (video and text) and many books, if that is your preferred learning mode.
- The initial learning curve is fairly low: you can start to get interesting results very quickly.

- It does, however, teach you a variant of the *Java* programming language, which is widely used elsewhere, and is so similar to other major computer languages, such as C++, that learning it is a useful step on the way to mastering some of the skills of professional programmers.
- Processing actually supports a number of alternative languages for those who already have some skills. You can drive it with a variant of *Javascript*, widely used in web programming and create animations that can be embedded in web pages. Or, if you have come across Python in school work, that can also be employed in Processing work.

Go to the website and follow the instructions for downloading and installing Processing.

First, I suggest that you watch the three short video tutorials on the Tutorial Page.

Then, you should work your way through at least the following text-based tutorials:

- Getting Started.
- Processing Overview. You get to draw your first lines here.
- Coordinate System and Shapes. This is about how to place things where you need them to be.
- Color. You have various options for specifying the colour of shapes, including mixing different amounts of red, green and blue. You also learn about opacity - whether drawing a shape hides what it behind it (high opacity) or lets it show through (low opacity).
- 2D Transformations. You can manage without this, but it may make your program easier to write.
- Trigonometry Primer - for those who have forgotten it.

You also need to know about the Reference pages, where you find detailed instructions on drawing different types of shapes etc. - see for example line() for drawing lines.

Having installed Processing and worked through the above tutorials we should have achieved our Elements 1 and 2 above. You can now draw a line where you want it to appear and with a colour and opacity of your choice.

You should also be aware that Processing has already simplified one of our jobs: we want our image to build up over time, and you should have learned that Processing has a built-in time-loop. Its draw() function goes round and round (at about 60 times per second if it can keep up) and draws whatever it is instructed to draw on the screen each time.

In the early Processing tutorials this draw cycle was called implicitly. You should, however, now get into the habit of having explicit setup() and draw() functions in your program. You will need these to be separate for your more complicated images, so you might as well start getting used to it now.

Remember that the instructions inside setup() are executed just once when Processing starts up, while draw() goes round and round until you stop the program. We can use setup() to set the values of variables that are not going to change - or the initial values of variables that will change over time.

You also need to remember what they said about the *scope* of variables if you want to set a value in setup() and refer to it or change it in draw(). If this is what you want to do (I want to do it all the time) then the variables have to be declared *outside* either setup() or draw() because that gives them *global* scope - they can be seen by any part of the program. If you declare something *within* either setup() or draw() then that variable is seen only within the *local* scope of that function.

Now the real work begins with Element 3: devising and programming an algorithm to produce the image we wish to create.

Designing Our Program

There are two distinct aspects to the design of relatively small programs. (Very large systems raise much broader questions - but these are the province of professional software engineers.)

- Firstly, we need to think about the information that the program must manage.
- Secondly, we must devise algorithms to manipulate that data.

Novice programmers tend to focus on the algorithms - the programming instructions that you write - while experienced professionals know that the key to good program design is usually working out what data you will need, how to get it into the program, how to organise it within the program, then how to transform it (the algorithm) and finally how to move the result out into the world (in our case how to draw on the screen and perhaps how to save a copy of our image to a JPEG file on disk).

So let us think about what we will need to tell the program. I my first thoughts are along these lines:

- We always need to tell Process our desired drawing area size (the number of pixels along horizontal and vertical axes). We usually do this, as the tutorials show, in the setup() function. Furthermore - a subtlety - we normally do this by putting explicit integers into the size() function. These two numbers (the width and the height) will determine the entire scale of what we wish to do with our drawing. Fortunately, once we have called size() the dimensions that we will work within are held in global variables width and height that are accessible anywhere within the program.
- Every dimension we use from now on, such as the length of the lines we draw, will be scaled by one of these drawing area dimensions. (So, we might say that our initial line length will be 90% of the smaller of the two drawing area dimensions, width or height, for example.) We will calculate the initial line length in setup(). We ought to define this as a global variable (that is declared outside setup() and draw() so its value can be seen anywhere in the program).
- I want my line to rotate around as I draw it on each cycle of draw(), so I will need to define the rotation speed. Speed with respect to what? It is useful to know that by default Processing tries to draw a new frame in the drawing area about 60 times per second. (This can be altered in setup() by calling the frameRate() function, and you can always find the current rate in the global variable frameRate, the latter variable will turn out to be useful.) It is useful to know that there is a frameCount predefined global variable which tells you how many times draw() has been called since the program started running. Hence, a measure of the number of seconds since we started running the program is simply:

$$time = frameCount/frameRate$$

- This *time* value, which you should calculate every time you go into draw(), turns out to be useful in just about every Processing program that I create. It should become part of a stereotypical bit of code that you use in all your programs. We will be able to use it in several ways:
    - We can use it to calculate the angle of the line that we draw using an expression something like: *theta = 360 * time/rotationPeriod;* where rotationPeriod is the time it would take the line to rotate through a full circle. Even better, for reasons that become clear below, would be: *theta = TWO_PI * time/rotationPeriod;* which gives us or angle in *radians* rather than degrees - this is usually much more convenient in programming that involves trigonometry expressions. (Note that TWO_PI is a built-in *global constant* in Processing, along with some other useful values, because we need to use the value of 2π so frequently.)
    - We can use it to determine the coordinates of the starting point from which we draw our line. We will want this position to move around the screen as time advanced. A simple way of doing this might well be two expressions for the starting point of the line:

      $$x0 = xAmp * sin(TWO_PI * time/xPeriod);$$
      $$y0 = yAmp * sin(TWO_PI * time/yPeriod);$$

      In these expressions, x0 and y0 describe the horizontal and vertical coordinates of the point from which we start drawing our line. They use the built-in sin() trigonometry function to describe side-to-side and up-and-down oscillations of the starting point. If you recall the way sin() works you will see that it comes back to the same place every time the function argument - inside the brackets - is a multiple of 2π, which happens in the horizontal direction when *time* is a multiple of *xPeriod* and in the y direction when time is a multiple of *yPeriod*. This is often known as a *harmonic* motion because sine waves are quite good for describing musical sounds.. The size of the side to side movement is given by *xAmp* and the size of the y movement by *yAmp*, so we have also identified four more configuration variables which we will need to declare and to which we will need to give initial values. These are *xAmp, yAmp, xPeriod, yPeriod*. It is a good idea to specify *xAmp* and *yAmp* in terms of a fraction of the size of the drawing area, so that if you choose to make a bigger drawing area everything will expand accordingly.
    - We will also use time to change to line length and probably also the motion amplitudes. A good way to do this would be to define an expression for a value that decreases over time, which we will use to multiple to motion amplitudes and the initial line length before using them to draw. Something like:

      $$decayFact = exp(-time/decayRate);$$

      The 'exponential' function exp() starts with a value 1, when the quantity inside the brackets is 0 - i.e. time=0, and then for each period of *decayRate* it decreases by a factor of about 1.713… and goes on forever, never quite getting to zero. We do not need to know the good mathematics concerning why it behaves like this but it is just useful behaviour. The important point is that for each period of decayRate, we will notice some change in behaviour, nether too fast nor too slow. You will also see

that we have again introduced the need for another global variable *decayRate* which has to be given an initial value.

    - We may also use the *time* value to specify changes in the colour of the line we draw. This will be an exercise for the student.

- Although I started by thinking about the data we would need, we have already started to stray into the area of algorithms by deciding that we will probably want to use built-in functions such as *sin()* and *exp()*. You can never quite separate the two, of course, because algorithms imply a need for configuration data, and the need to turn input data into output data (in our case the position, length orientation and colour of a line) implies the algorithm. I do, however, want to reinforce the viewpoint that I recommend: what data do we want to start with, what data do we need to end up with, derive the need for the particular algorithm from that. This nearly always works best for relatively simple programs, and in the design of complex software it is still important: we just tend to be somewhat more formal about the way we analyse our information needs and the way it interacts will algorithms.

There are just a couple of points remaining that are strictly in the area of algorithm design. You would trip over both almost immediately if you tried to start programming. Firstly, we have talked about specifying the angle at which we draw our line. Just how do we do this. There are two ways, both of which work equally well in this case, and both of which might turn out to be a better solution for particular more complicated design problems.

- The Processing line() function (you will also hear it referred to as a "graphics primitive" because it is a basic drawing operation) needs start and end (x,y) coordinates. We know how to get the start coordinates - the have explored that above. How do we get the end coordinates. Here we can just use basic trigonometry (look up the tutorial if you need to) with *xEnd = lineLength*cos(theta)* and *yEnd = lineLength*sin(theta). (*Remember that here, *lineLength*, is the initial line length multiplied by our decay factor.) There is one potential pitfall here: you may, in spite of my advice above, have specified the relation between line angle and line rotation period so that the angle *theta* comes out in degrees. In all programming languages, however, the built-in trigonometry functions expect to get angle in *radians*. (This is not just a design to be arcane and inconvenient, there are compelling mathematical reasons why this is exactly the right thing to do.) So, before putting an angle specified in degrees into a trig function you must multiply it by a conversion factor $\pi/180$. This is needed so often that I always define a global variable *degToRad* and set its value as above in *setup().*

- The second option is to avoid the trig calculations entirely by rotating our entire coordinate system such that the x-axis now lies in the direction we want to draw our line. For this we use the rotate() function. See the 2D Transformations tutorial. We can now specify our line by just drawing on the x-axis - a trivial matter. This is in a number of ways the more elegant approach that is often useful in more complicated situations.

- The `transformation' approach also lets us tack another issue that you would soon notice if you immediately tried to follow the instructions above, without thinking too deeply. Our expressions that define the starting point of our line drawing use sin() function that take both positive and negative values as time advances. However, the default coordinate system in processing puts the zero of the coordinates at the top left hand corner of the drawing area. It is a good idea therefore to use the translate() function to move our origin to the centre of the drawing area.

There is just one more thing you may need: how will you save your finished art work? You will probably need to use the saveFrame() function, but you will have to choose whether you invoke this at a fixed time after starting, or, say, whenever you press and release the “s” key (my preference). I will you to read up how to capture “keyboard events” - but look up keyReleased() event-capturing function that is called whenever a key is released. (Use the related event-trapping function keyPressed() with care, because if you keep the key pressed down it is called each time draw() goes around it loop and is executed multiple time. This can be very useful - but not here.)

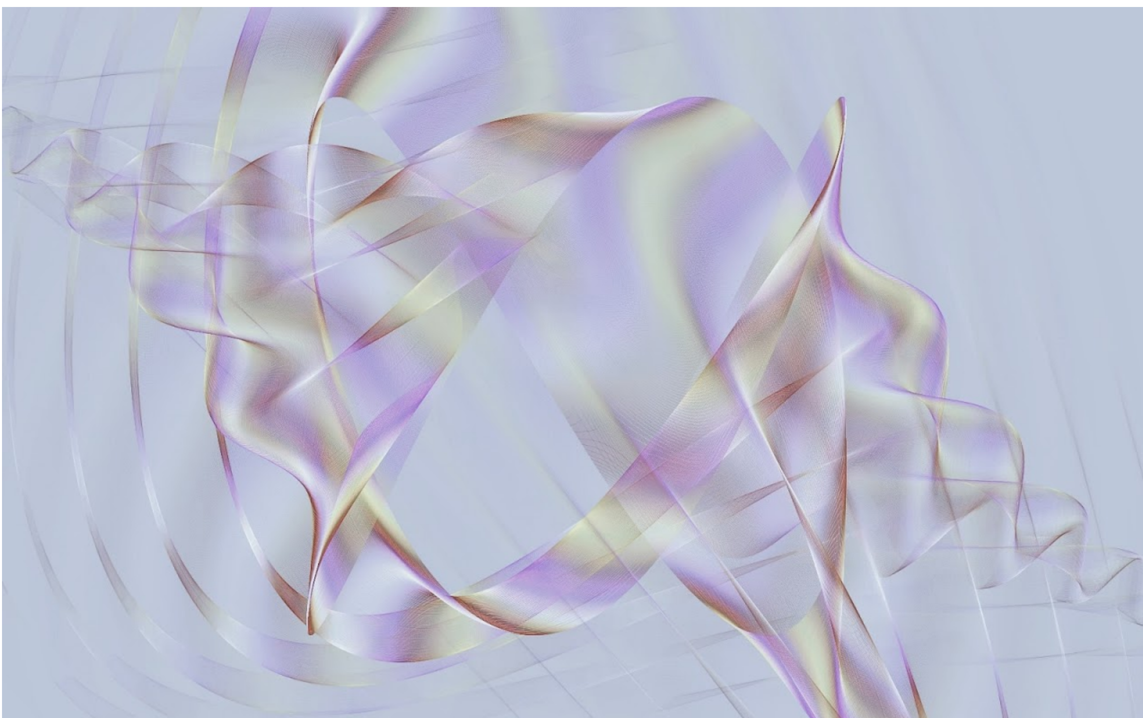
Take it From Here!

You will soon find that you can produce a large variety of images by making slight variations in the configuration parameters. Here are a few things to explore:

- If the periods of the side-to-side and up-and-down motions of the line start are related in some simple ratio (e.g. 2:1 or 3:4 and so on) the trajectory is a *Lissajous Curve* (look it up) and it will draw over itself eventually (apart from the overall scale decay).
- You can now play with the period of rotation and see what happens when it does (or does not) relate to one of the motion periods.
- The colour of the line can be determined by RGB colour values fed to the stroke() function. You can make each of the red, green, blue components of colour vary harmonically around some central value, with periods that may (or may not!) relate to other periods in the image. See my Lissajous Curves gallery for examples.
- You might prefer to use the alternative HSB colour mode, where harmonic variations of hue, saturation and brightness will produce potentially quite different visual effects.

There are many variations that can now be played on the central theme.

- You might work out how to produce a line that fades to nothing along its length, or changes colour along its length.
- You might choose not to use a simple line, but a wavy shape. See for example the image to the below of one of my Generative Waves image galleries.



Further Explorations

See my own website Artful Computing for some ideas and work that I have myself produced. In particular, see the Links page, and the Tools and Resources collection where you will find a page about Books and some pointers to mathematical art websites. (I have tried to keep to web sites that have some longevity - but interesting websites come and go quite regularly. You can also find material on *Instagram* by searching for tags such as **#generativeart, #algorithmicart, #mathematicalart, #processing** and so on, while there are a number of *Facebook* groups dealing with computer generated art using both Processing and other tools, such as those specialising in 3D animation.