CGRA 151 – Worksheet 1 — introduction to basic Processing and getting intuition about randomness

Objectives: (1) To prepare you with sufficient knowledge of the Processing programming language that you can attempt the assignments. (2) To give you some insight into the behaviour of random numbers

Get Processing running

On one of the ECS machines, type

```
processing
```

at the command prompt. Hit enter and the Processing environment should start up with an empty "sketch". Processing programs are called "sketches" to remind us that the language is designed to be used as a programming sketch-pad, where you can write short fragments of programs on the way to writing a full program, the same way an artist or designer would use a physical sketch-pad to sketch out ideas before starting work on their final piece of art.
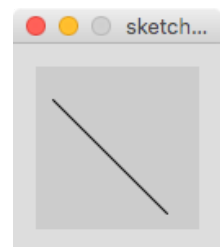
Draw a single line

In the sketch type the line:

```
line(10,20,80,90);
```

Don't forget to put the semicolon at the end.

Now press the run button: ▶

You will get a new window, with a grey background and a single black line.

Draw a single point

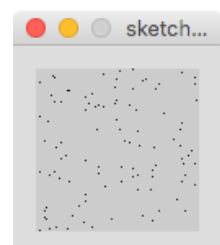Delete the line command and replace it with a point command:

```
point(50,50);
```

Press the run button again. Processing will stop the current sketch and run the new sketch. This time drawing just a single point in the middle of the window.

Draw a hundred random points

The `random` command has two parameters, `random(a,b)`. It produces a random floating point number between `a` and `b`. To generate 100 random points, we use a simple for loop. Delete the point command from your sketch and type in the following in its place:

```
for (int i=0; i<100; i++) {
  point(random(0, 100), random(0, 100));
}
```

Press the run button again. Processing will stop the current sketch and run the new sketch. This time drawing 100 points randomly distributed across the window.

Background and foreground colours

Processing's default background is a pale grey. Let's now change the
sketch to draw white points on a black background.

Add the following commands to the start of the sketch:

```
background(0);
stroke(255);
```
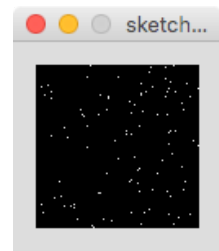
If you specify a single parameter, Processing treats it as a grey value. Grey
values range from 0 (black) to 255 (white). Processing uses the stroke colour for drawing
lines and ponts.

You can also specify colours using RGB (red, green, blue) triples. For example, red points on
a light yellow background:

```
background(255,255,192);
stroke(255,0,0);
```

Now set the sketch up to draw bright yellow dots (255,255,0) on a black background.

```
background(0);
stroke(255,255,0);
```

Setting window size

Processing's default drawing window is 100x100 pixels in size. We asked it to draw points
with coordinates ranging from 0 to 100, so that all of the points lie inside the window.

Now try changing the size of the window to 400x300 by adding this command at the start of
the sketch:

```
size(400,300);
```

Run the sketch again.

You should get a window of size 400x300, with 100 random points drawn in the top left
100x100 region of the window. We haven't changed the parameters of the random number
generator, so our sketch only draws points where it was told to: in the top left 100x100
region of the window.

Using built-in parameters: width and height

Processing has a range of built-in variable names that allow you to access things like the
width and height of the window. If we want random dots across the whole window, we can
use these built-in variables and then we do not need to worry if we later decide to change
the size of the window.

Change the sketch so that it uses `width` and `height`. It should now look like this:

```
size(400, 300);
background(0);
stroke(255, 255, 0);
for (int i=0; i<100; i++) {
  point(random(0, width), random(0, height));
}
```

You should notice that Processing helps you out by highlighting various built-in commands, variable names, and function names in different colours. This helps you to remember that they are built-in, so that you do not accidentally use one of the built-in names as one of your own variable or function names.

Drawing 100 random lines

Drawing 100 random points is not particularly exciting. Change the point command so that it draws a line instead.

```
line(0, 0, random(0, width),
        random(0, height));
```

This draws 100 random lines, all starting at (0,0),which is the top left hand corner of the window. Press the run button a few times to generate a few different random sets of random lines so you get some idea of how similar (or different) these different sets of random numbers look.
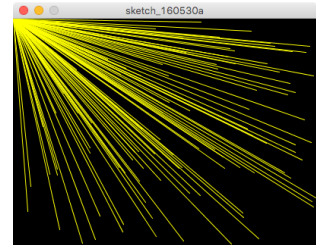
Lines radiating from the top left corner are OK but perhaps we want lines that start and end at random positions in the window. This is easy to achieve. Replace the current line command by:

```
line(random(0, width), random(0, height),
    random(0, width), random(0, height));
```

Notice that Processing ignores line breaks and extra spaces, which allows you to format your sketches neatly. If your sketch is looking a bit messy, you can automatically tidy it up by using the **Auto Format** command on the **Edit** menu. Try that now. If you use this, it applies to the whole sketch and you are essentially forced to accept the formatting that Processing's authors think is appropriate, which is usually OK.

Experimenting with randomness

Change the loop so that it draws 1000 lines and re-run. You should notice that the centre of the window is now almost completely yellow, while the outer edges of the window still have a lot of black (see example at right).

Try 10000 lines. The centre of the window will now be completely yellow, but the outer edge will still have random bits of black.
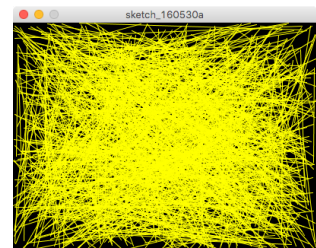
Why does generating 10000 random lines *not* fill the entire window with yellow? Why is the centre solid yellow but the outer boundary not? We used a random number generator to generate the two end points, but the visual result is not uniformly random. Why is this? Think about it before reading on.

As you've probably worked out, the result is because pixels near the centre of the window are much more likely to be between two random end points than pixels at the edge. For a pixel at the edge to be on a line, the random number generator had to generate almost the same x-value (or almost the same y-value) for both ends of the line, which is a lot less likely than generating values that are not almost the same.

This illustrates that you need to be careful when using random numbers: just because the `random` function produces numbers that are uniformly random does not mean that you will get uniformly-random graphical outcomes when you combine several random numbers.

Saving your work

You will need to save your sketches (i.e., your programs) so that you can submit them for marking. Try this now. Change the current sketch to draw 1000 lines and then save it as "Lines1000".

To do this, go to the **File** menu and select **Save**. In the dialog box that pops up type in the name "Lines1000" and click on the **Save** button.
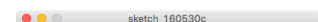
Processing stores all of its sketches in sub-directories of one master directory. It will create a sub-directory, "Lines1000", and in that directory store a file "Lines1000.pde". "pde" is the abbreviation for "Processing Development Environment". There can be multiple "pde" files for a sketch, which will all be stored in the same directory, but the principal "pde" file of the sketch must have the same name as the directory.

When you submit a processing sketch for marking, you need to submit the whole directory. The way we handle this in the ECS submission system is for you to zip, into a single zip file, all of the directories that contain your sketches for that assignment. You submit this single zip file. The marker will then unzip the file into their marking directory.

Drawing a rectangle

Create a new sketch, using the **New** command on the **File** menu. Draw a pale orange rectangle on a white background, by typing in this code:

```
size(400, 300);
background(255);
fill(255, 192, 128);
rect(90, 110, 220, 80);
```

Hit the run button. This draws a rectangle of size 220x80, with its top-left corner at point (90,100).

Notice that the rectangle has a black outline. By default, all shapes have an outline in the current stroke colour. The default stroke colour is black. You can turn off the stroke by using the function:

```
noStroke();
```

Add this to the sketch, before the `rect()` function. Re-run and see the same rectangle without a black border.

Drawing 100 random rectangles

Let's now try some randomness. Replace the single `rect()` command by:

```
for (int i=0; i<100; i++) {
  rect(random(0, width), random(0, height), 40, 40);
}
```

This draws a bunch of small orange squares. Maybe we do not like this shade of orange, so let's make the colours random, by putting the fill command inside the loop and setting the red, green and blue parameters at random, like so:

```
for (int i=0; i<100; i++) {
  fill(random(0, 255), random(0, 255), random(0, 255) );
  rect(random(0, width), random(0, height), 40, 40);
}
```

Notice that the colour parameters (red, green, blue) take values between 0 and 255. Zero represents none of that component of the colour; 255 represents the maximum amount of that component of the colour. (0,0,0) is black, (255,255,255) is white, (255,0,0) is pure red, (255,128,0) is a bright orange. (255,192,128) is fully red, ¾ green, ½ blue, which makes a pale orange: halfway between bright orange and pure white.

Run that sketch a few times, to see different random patterns. You'll notice that some of the rectangles go off the right and bottom of the window, but none go off the left and top. This is because the top-left corner is generated by the random number generator and can lie anywhere in the window. To get all the boxes to lie inside the window, try this:

```
rect(random(0, width-40), random(0, height-40), 40, 40);
```

Random rectangles of random sizes

If the rectangles are now generated in random sizes, then, in order to make sure that no rectangle goes outside the window, we have to calculate a size first and then choose a position at which to place that rectangle. We therefore define two parameters:

```
float rWidth = random(10, 50);
float rHeight = random(10, 50);
rect(random(0, width-rWidth), random(0, height-rHeight),
     rWidth, rHeight);
```

Try this code and see whether it produces the effect you expect.

Debugging

When programming, you often need to work out why the code is not producing the result you expected. In Processing, you can output text to the console window (you type your code into the white region of the Processing window; the output console is the black region just below the white region). For example, if you wanted to check the values of rWidth and rHeight you could add this to the code:

```
println( "rWidth =", rWidth, "   rHeight =", rHeight) ;
```

`println()` prints whatever parameters it is given, then adds a new line. `print()` prints whatever parameters it is given, but does not add a new line.

If you want to know what any Processing function does, or see whether a particular function exists, go to the Processing reference manual. Use the "Reference" item on the "Help" menu, or visit the reference manual here:

https://processing.org/reference/

Making attractive random rectangles

Play with the random parameters (in both size of rectangle and fill colour) and with the number of rectangles until you get something that is artistically pleasing. Show your neighbour. *Save your work.*

Ellipses

Replace the `rect()` command with an `ellipse()` command.

```
size(400, 300);
background(255);
noStroke();
for (int i=0; i<100; i++) {
  fill(random(0, 255), random(0, 255), random(0, 255) );
  ellipse(random(0, width), random(0, height), 40, 40);
}
```
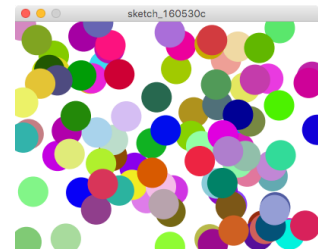
By default, ellipses are centred at the point defined by their first two parameters, so here we see ellipses that go off all four sides of the window. This is in comparison to rectangles, which are specified by their top-left corner, rather than their centre.

Using a similar approach to what we did with rectangles, change the sketch to draw random ellipses of various sizes that all lie inside the window.
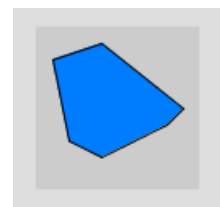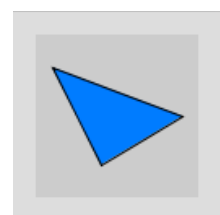
Other primitive objects

A triangle can be generated using three points (i.e., three pairs of (x,y) coordinates, so six values in total). For example,

```
triangle(10, 20, 90, 50, 40, 80);
```

A polygon is created using a `beginShape()`, `endShape()` pair surrounding a list of vertices.

```
beginShape();
vertex(10, 20);
vertex(40, 10);
vertex(90, 50);
vertex(80, 60);
vertex(40, 80);
vertex(20, 70);
endShape(CLOSE);
```

The `CLOSE` parameter in the `endShape()` function joins the last vertex specified, to the first vertex, completing the shape.

Interaction with the mouse

Processing has several variables that allow you to interact quickly with the mouse. They are the floats `mouseX` and `mouseY`, which give you the current position of the mouse; `pmouseX` and `pmouseY`, which give you the previous position of the mouse; and the boolean `mouseButton`, which tells you whether a mouse button is pressed or not.

To use mouse interaction, you need to move to using Processing in its active mode. So far, you have been writing sketches in *static* mode, where Processing draws the image on the window once. In *active* mode, Processing redraws the window every sixtieth of a second. In active mode we use the functions `setup()` and `draw()`. `setup()` is called once, when the sketch starts. `draw()` is called every sixtieth of a second after that.

Create a new sketch and type in the following:

```
void setup() {
  size(400, 300);
}
void draw() {
  rect(mouseX, mouseY, 40, 40);
}
```
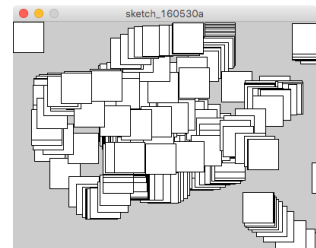
You now have a rectangle that follows the mouse. It leaves a trail because you did not tell Processing to clear the window before it draws the next rect. The `background()` function fills the entire window with the background colour. Try adding

```
background(0);
```

to the code just before the `rect()` function.

You can change the number of times a second that Processing calls the `draw()` function by using the `frameRate()` function. This is usually put inside the `setup()` function. Try adding, just after the `size()` function, the command:

```
frameRate(5);
```

The rectangle will now be drawn only five times a second. What happens when you move the mouse? Experiment: What frame rate do you need to make it feel as if the movement is continuous.

Finally, write a Processing sketch that has `setup()` and `draw()` functions. The `draw()` function should draw a circle centred on the current mouse position. The circle should be red if a mouse button is pressed and blue if it is not pressed. Use the built-in variables `mouseX`, `mouseY`, and `mousePressed`.

Congratulations

You now should have sufficient knowledge to complete successfully Assignment 1.