SDES3107
DESIGN AND COMPUTERS 4

INTRODUCTION TO PROCESSING

WEEK 2
MONDAY
D103

LAST WEEK

CODE ELEMENTS Comments, functions, expressions, statements, console

COORDINATES, PRIMITIVES Coordinates, primitive shapes, drawing, grey values

VARIABLES Data types, variables, Processing variables

ARITHMETIC, FUNCTIONS Arithmetic, operators, grouping, shortcuts, constraining

DECISIONS Relational expressions, conditionals, logical operators

REPETITION Iteration, nested iteration, formatting code blocks

WEEK 2

INTRO TO GENERATIVE GRAPHICS

ASSESSMENT

TOPICS:

VERTICES Vertex, points, lines, shapes, curves

COLOUR BY NUMBERS Setting colours, colour data, RGB, HSB, hexadecimal

DISPLAY, TINT Display, image colour, transparency

CURVES Exponents, roots, normalising, mapping, simple curves

DON'T FORGET!

WE ARE WORKING THROUGH THE TEXT BOOK: YOU SHOULD BE READING EACH CHAPTER THROUGH TO GET MORE DETAIL.

ALSO: OPENPROCESSING.ORG

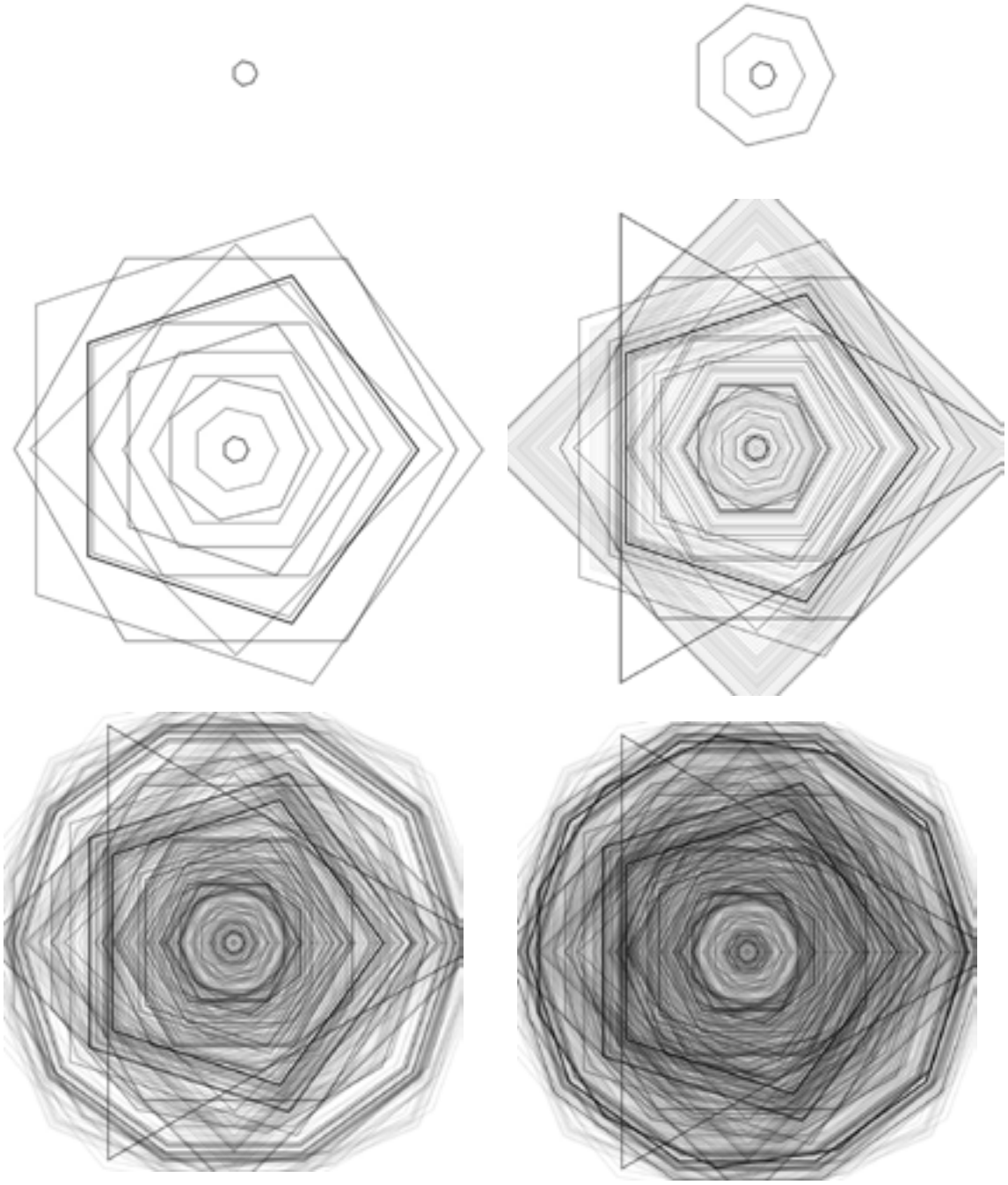
GENERATIVE DESIGN

GENERATIVE GESTALTUNG

Images can be generated by code. An image is no longer created “by hand” but the fact that a visual idea is translated into a set of rules and then implemented in a programming language in the form of source code. The result is that such a program can not only produce a single image, but that by changing parameters whole imagery arise.

http://www.generative-gestaltung.de/code

Also check out http://abandonedart.org/



ASSESSMENT:
PROCESSING BRIEF

Worth **20%** of total course mark

Using the supplied *Generative Code* Package 2 files and using Processing 2.07b, with included extra libraries, select from any two programs from available chapters P_1_0_01.pde (colour), P_2_0_01.pde (form), P_3_0_01.pde (type) or P_4_3_04.pde (image).

10 digital images (PNGs) from code experiments and include a short text explanation of each (i.e. how you changed the code and what it does).

Digital submission only (submit to Omnium website)

Due **end of week 7**

REVIEW OF REPETITION AND DECISIONS

DECISIONS (p.51)

So far, we have been writing code that simply runs top to bottom.

To add further control, we can use **relational expressions** to decide **if** a section of a program should be run.

DECISIONS (p.52)

Operators used in relational expressions allow us to test if something is `true` or `false` by **comparing values**.

e.g. `println(3 > 5)` prints false

`println(3 < 5)` prints true

`println(3 >= 5)` prints false

`println(3 <= 5)` prints true

`println(3 == 5)` prints false

`println(3 != 5)` prints false

>
greater than

<
less than

>=
greater than or equal to

<=
less than or equal to

==
equal to

!=
not equal to

IF STRUCTURES (p.53)

When using relational expressions to make a decision in a program, we use what is called a **conditional** - an **if structure**.

If the conditional evaluates to true, the code inside the brackets will run. If it is false, the program ignores it.

```
if (test) {
   statements
}
```

e.g.

```
if(3 < 5) {
  ellipse(50,50,10,10);
}
```

EXTENSIONS (p.55-56)

We can extend an if structure by adding an **else**. The code inside the **else brackets** will only run if the conditional is **false**.

We can add layers of conditionals by nesting them within each other.

```
if(test) {
    if(test) {
      code;
    }
}
```

We can also add extra control to an else by adding an if.

e.g.

```
if(3 < 5)
ellipse(50,50,10,10);
} else {
  rect(50,50,10,10);
}
```

LOGICAL OPERATORS (p.57-59)

Sometimes we need to combine two or more relational expressions. To do this, we need logical operators.

OR (||) makes the whole expression return true if only **one** part is true.

AND (&&) makes the whole expression return true only if **both** parts are true.

NOT (!) inverts the logical value and can only be applied to boolean variables. It is less used than OR or AND.

e.g.

```
if((3 < 5)&&(3 > 2)) {
  ellipse(50,50,10,10);
}
```

Both parts are true, so the code will run.

EXERCISE

5.2. Create a composition with a series of lines and ellipses. Use an if structure to select which lines of code to run and which to skip.

(p.59)

REPETITION (p.61)

Often we find ourselves writing the same code over and over. For example, to draw lots of ellipses.

To help cut down on lengthy code, we can use **iterative structures**, or **for structures**, that automatically repeat lines of code for us.

Like if structures, we can **nest** for structures inside each other to create complex patterns.

```
for (init; test; update) {
   statements
}
```

e.g.

```
for (int x = 20; x < 80; x += 5)
{
   line(20,x,80,x+15);
}
```

EXERCISE

6.1. Draw a regular pattern with five lines. Rewrite the code using a for structure.

6.2. Draw a dense pattern by embedding two for structures.

(p.68)

SHAPES: VERTICES (p.69)

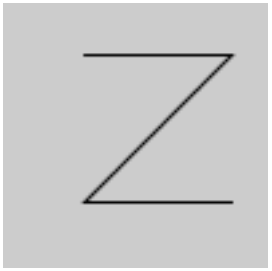
We can use a series of functions to move beyond primitive shapes and make more complex forms.

To do this, we use the following structure:

```
beginShape();
vertex(x,y);
endShape();
```

We can add however many vertices we want to make a shape.

e.g.



```
noFill();
smooth();
beginShape();
vertex(30,20);
vertex(85,20);
vertex(30,75);
vertex(85,75);
endShape();
```

CURVE VERTICES (p.74-77)

Vertices are great for straight lines, but what about curves?

Instead of `vertex(x,y);` we can use `curveVertex(x,y).`

e.g.

```
smooth();
noFill();
beginShape();
curveVertex(20,80);
curveVertex(20,40);
curveVertex(30,30);
curveVertex(40,80);
curveVertex(80,80);
endShape();
```

EXERCISE

7.1. Use beginShape() to draw a shape of your own design.

(p.77)

IMAGES (p.96)

We can load and display images in Processing.

To put an image in our sketch, we must **create an image variable**, **load** it, and then **call** it:

```
PImage img;
img = loadImage(“image.jpg”);
image(img,0,0);
```

↑

coordinates for position of image

The image must be in a folder called “data” in your sketch folder.

TINTING (p.74-77)

Once you've loaded and displayed an image, you can tint it with a particular grey value or a colour.

```
tint(grey value);
tint(value1, value2, value3,
alpha);
```

Set the tint before you call the image with `image(img, 0,0);`

`noTint()` disables the tint.

e.g.

```
PImage img;
img = loadImage
("arch.jpg");
tint(102);
image(img, 0, 0);
noTint();
image(img, 50, 0);
```

EXERCISE

10.1. Draw two images in the display window.

(p.99)

EXPONENTS, ROOTS (p.79-81)

Processing uses several functions that can perform various mathematical tasks:

`sq(value)` = a number squared (e.g. x^2)

`sqrt(value)` = square root of a number

`pow(num,exponent)` = num raised to an exponent

number to multiply **how many times to multiply num by itself**

NORMALISING, MAPPING (p.80-81)

Sometimes we need to convert numbers to a smaller range: 0.0 - 1.0.

This is called normalising: `norm(value, low, high)`

number to normalise — **min value of range** — **max value of range**

```
e.g. float x = norm(102.0,0.0,255.0);
```

Numbers can also be mapped directly to new ranges:

```
map(value, low1, high1, low2, high2);
```

number to re-map

min value of current range

max value of current range

min value of new range

max value of new range

CURVES (p.83-84)

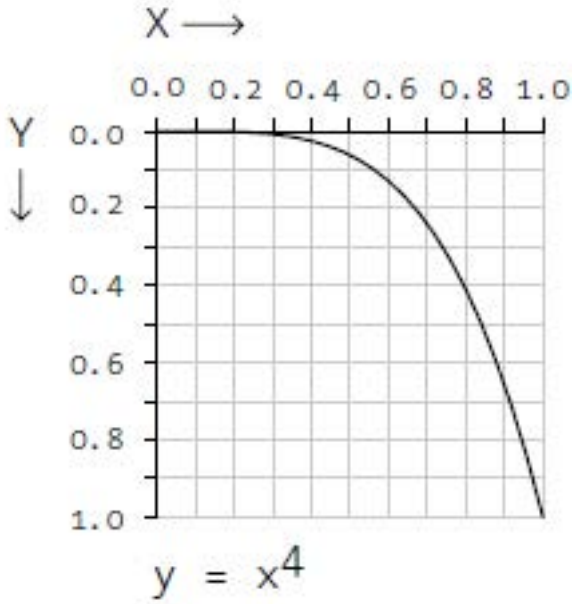
We can use exponential functions to create simple curves.

These come in the form: $y = x^n$

As x gets multiplied by itself over and over, y increases exponentially, creating a curve.

We use a for structure to cycle through numbers, normalise the current number, create the equation, multiply it by the maximum value of the range, then draw a point at the co-ordinates produced. This makes a curve.

X	Y
0.0	0.0
0.2	0.0016
0.4	0.0256
0.6	0.1296
0.8	0.4096
1.0	1.0




```
for (int x = 0; x < 100; x++) {
  float n = norm(x, 0.0, 100.0);
  float y = pow(n, 4);
  y *= 100;
  point(x, y);
}
```

FOR NEXT WEEK

EXERCISES: 7.1, 7.2, 8.1, 8.2, 9.1, 9.2, 10.1, 10.2, 10.3, 11.1, 11.2

READINGS: This week's topics: SHAPE 2, MATHS 2, COLOR 1, IMAGE 1, DATA 2, INTERVIEWS 1

Next week's topics: have a flick through TYPOGRAPHY 1, MATH 3, MATH 4, TRANSFORM 1, TRANSFORM 2

Don't forget to upload your best/interesting exercise to OpenProcessing.

FOR NEXT WEEK

ASSESSMENT PROGRESS: Do some research on generative graphics. Start thinking of designers or artists you can write about. Bring three potential case studies to class next week.

Explore the Generative Code Package and find one that you would like to work with for the assessment - one that relates to your artist or designer. Try looking at the code (but don’t worry if it doesn’t entirely make sense). Play around with the controls and bring the results of your experiments next week in the form of PNG files.