SDES3107

DESIGN AND COMPUTERS 4

INTRODUCTION TO PROCESSING

WEEK 1
MONDAY
D103

DR JOSH HARLE

josh.harle@gmail.com

COURSE COORDINATOR: BRAD MILLER

Room: E104
Phone: x50622
brad.miller@unsw.edu.au

Consultation times: Wednesday 10-12pm

HOUSEKEEPING

OMNIUM

PROCESSING TEXTBOOK

COURSE OUTLINE

- WEEKLY SCHEDULE
- ASSIGNMENTS

GITHUB

VERSION CONTROL

WEEK 1

INTRO TO PROCESSING

CREATIVE APPLICATIONS

USEFUL RESOURCES

TOPICS COVERED:

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

WHAT IS PROCESSING?

An open source programming language

An extension of Java, made for visual designers and artists

Focusing on visual, instant ‘sketch’ programming (with capabilities for sound and animation)

This course uses Processing 2.07b

WHAT IS POSSIBLE WITH PROCESSING?

(how is it relevant to design?)

DIGITAL NATIVES

Matthew Plummer-Fernandez

Everyday items such as toys and a watering can are **3D scanned using a digital camera** and **subjected to algorithms that distort, abstract and taint them into new primordial vessel forms**. In some cases only close inspection reveals traces inherited from their physical predecessors. **These are then 3D printed on a z-corp printer.**

www.plummerfernandez.com/Digital-Natives

CATALINA

Luis Clara Gomes

Music video for Catalina, a track off the Chocolat EP by Moullinex. Done with **Kinect + Processing + Cinema 4D + After Effects.** The 3D footage was captured using a a Microsoft Kinect, using Processing to store the sequence files and Cinema 4D R12, with some built-from-scratch python tag scripts to read these sequence files to Cinema 4D in real-time.

moullinex.tumblr.com/post/3180520798/catalina-music-video

MUSIC VIDEOS

WAVE BRACELET

Nervous Systems Studio

“Inspired by the complex forms of radiolarians, where intricate pattern is integral to structure, these shapes derive from a simulation of spring meshes. **We created interactive software to morph, twist, and subdivide each design**, transforming a simple mesh to a complex patterned structure.”

n-e-r-v-o-u-s.com/shop/

vimeo.com/10074976

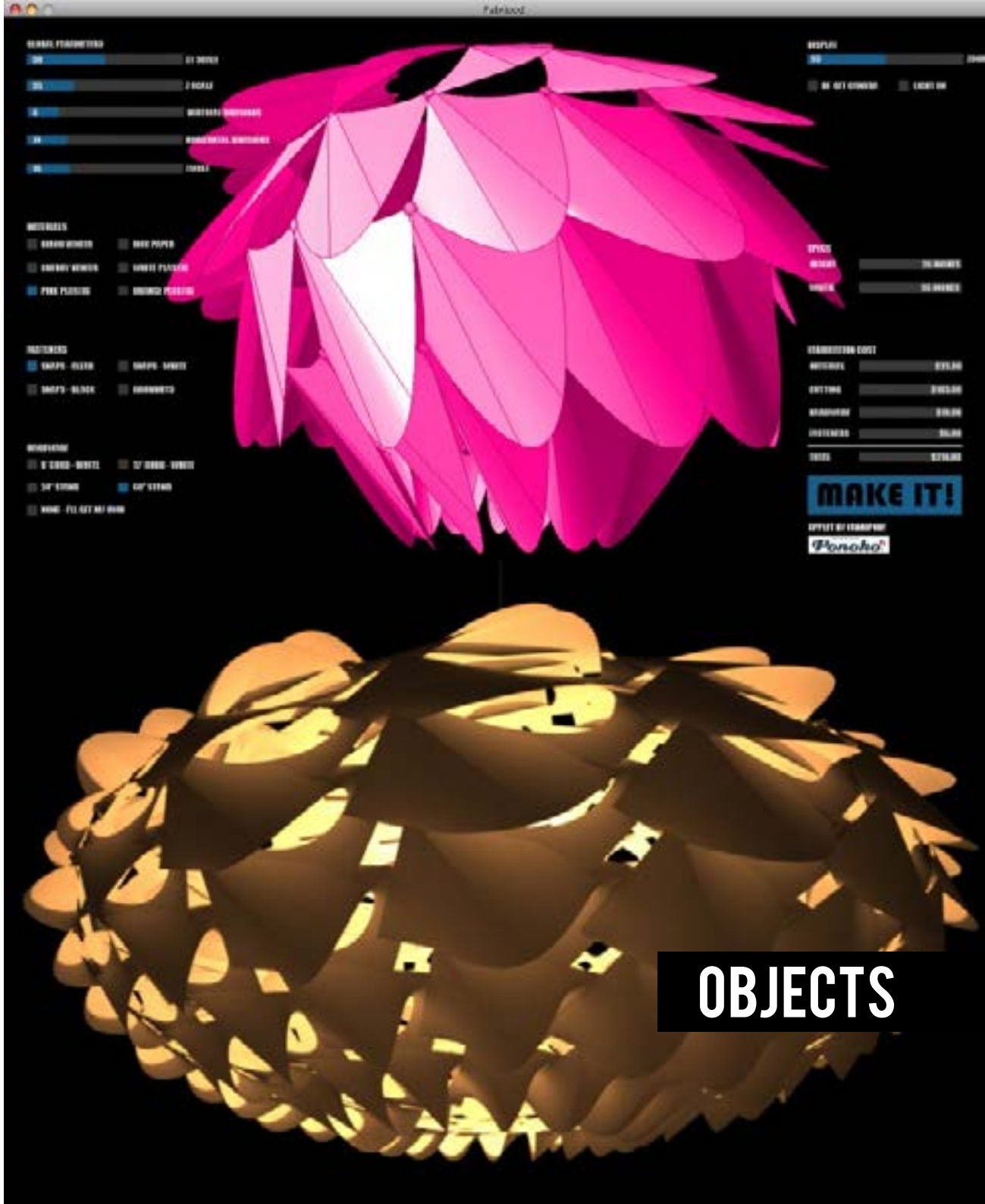
JEWELLERY

CHRYSALIS

Bios Design Collective

Chrysalis is a tool for translating designs made in Grasshopper (an associative algorithm editor used with Rhino 3D) into Processing (an open-source, web-based 2D/3D visualization programming language) sketches that can be used as web apps for digital making. Chrysalis will make it possible for you set up a web storefront that lets people customize a design and then export it to another service (like Ponoko or Shapeways) for making.

www.kickstarter.com/projects/1419987702/chrysalis-grasshopper-processing-for-online-making

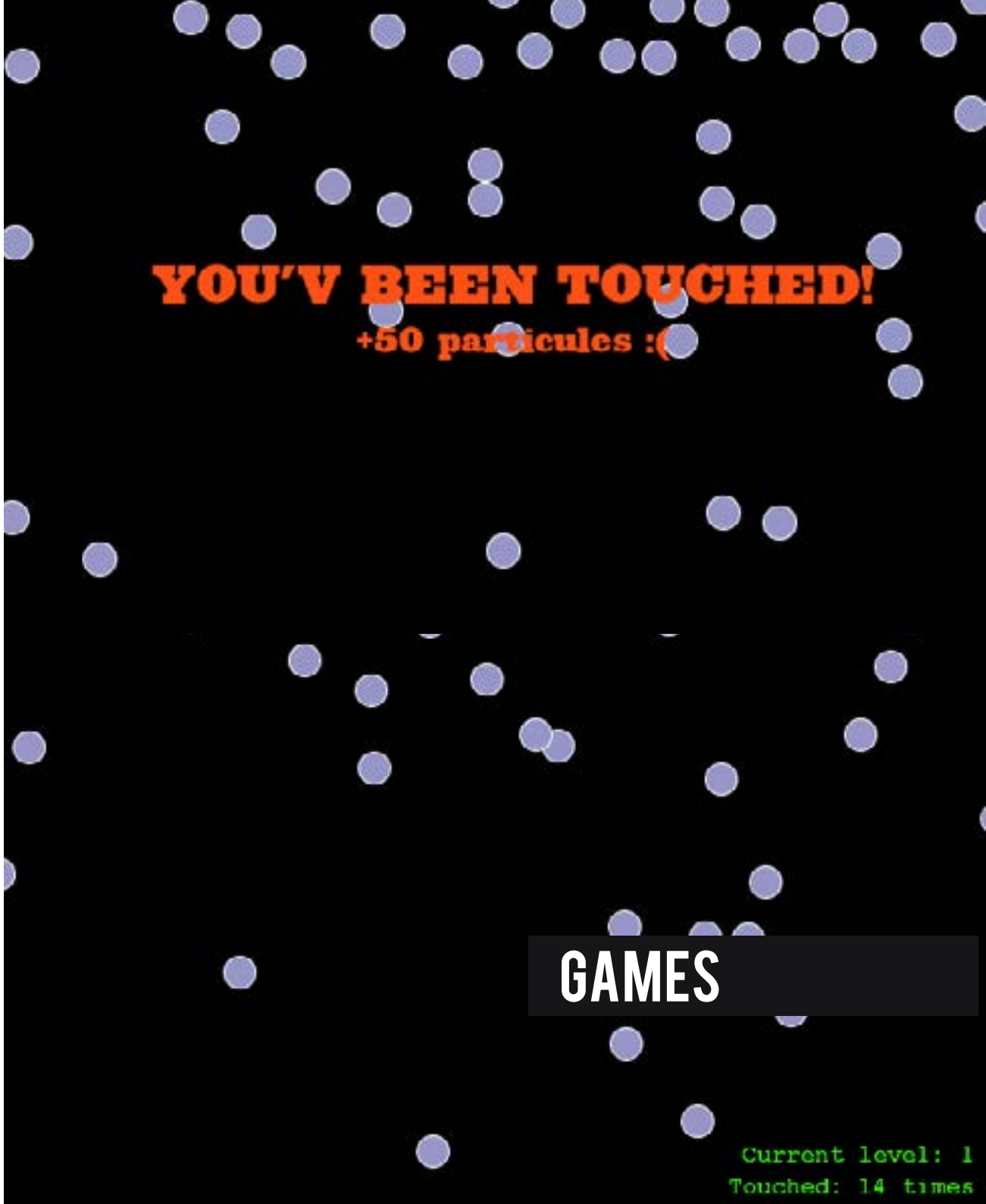


AVOIDANCE

Jonathan Rioux

Avoid the particles with your cursor and reach the highest level.

www.openprocessing.org/sketch/8787



THE ABYSS

CCTC

The Abyss is a collective work made during a one week workshop with Andreas Gysin around ‘Processing’.
The goal was to **build a deep-sea environment by creating some algorithmic creatures**.
At the end of the week, all the creatures were gathered together to create the final environment.

www.c-c-t-c.com/theabyss

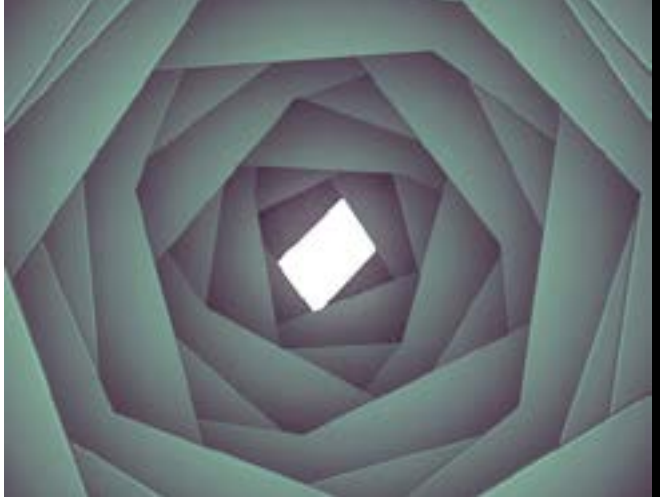
INTERACTIVE ENVIRONMENTS

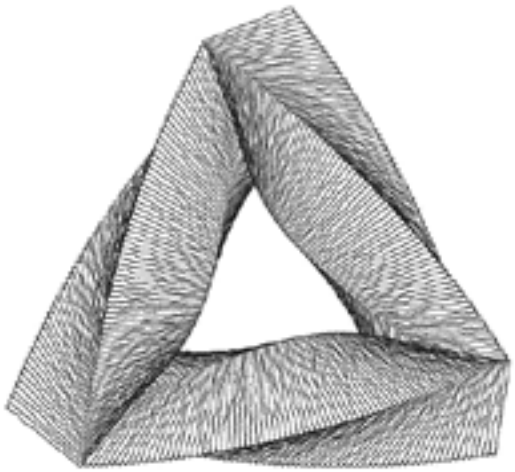
GIF ANIMATIONS

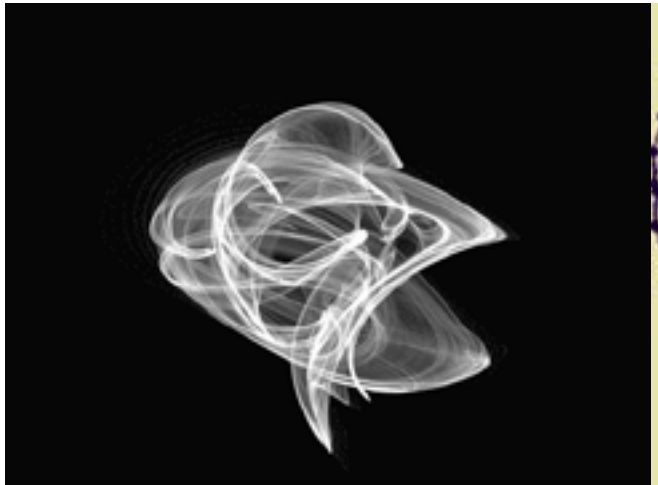
Paolo Ceric

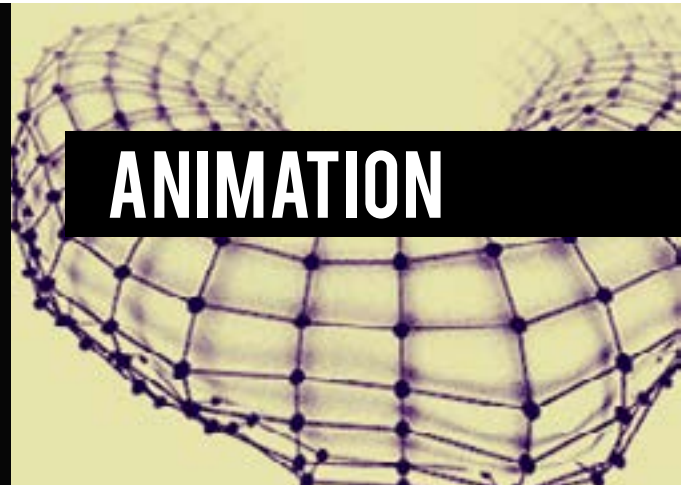
Digital artist Paolo Čerić's experiments with processing and digital art have resulted in a steady stream of fascinating animations. Lately he has truly developed his own personal style that varies from pulsating geometric patterns to glitch art and everything in between.

patakk.tumblr.com/









USEFUL RESOURCES

PROCESSING A Programming Handbook for Visual Designers and Artists
by Casey Reas and Ben Fry

GENERATIVE ART A Practical Guide Using Processing by Matt Pearson

PROCESSING.ORG/REFERENCE Your coding cheatsheet

PROCESSING.ORG/EXHIBITION Various creative projects made with Processing

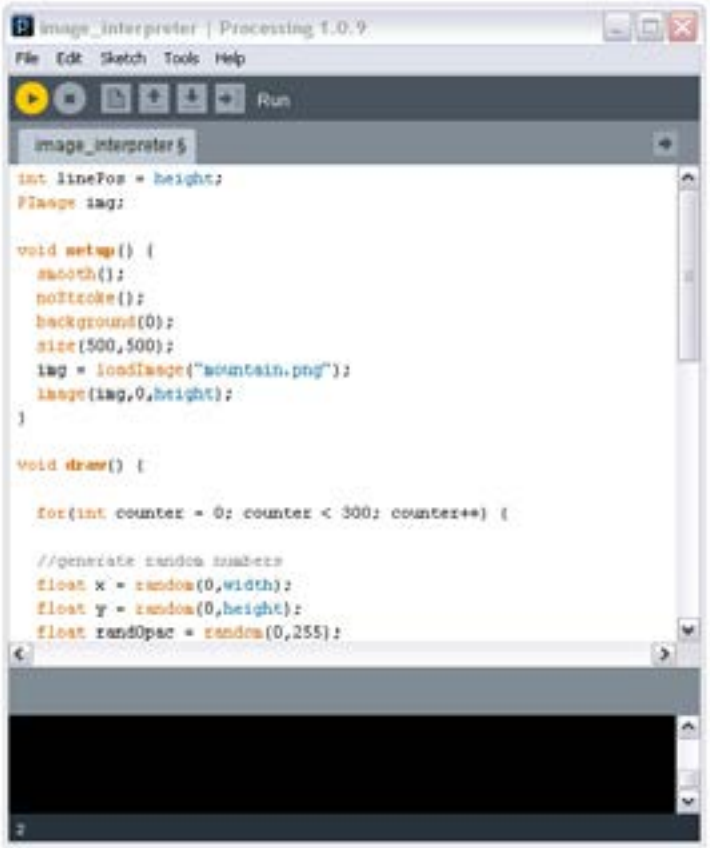
OPENPROCESSING.ORG Processing community site for learning and sharing

THE PROCESSING IDE
(INTERACTIVE DEVELOPMENT ENVIRONMENT)

controls

coding window

console



sketch window

WHAT IS A PROGRAM?

A procedural set of instructions that tell the computer what to do

Often likened to a recipe

Instructions are always carried out top to bottom

Method:

1. Process the biscuits in the bowl of a food processor until crumbs form.
2. Add the butter and process until combined.
3. Press the crumbs evenly over the base and sides of a 20cm springform pan using the back of a metal spoon.
4. Chill in the refrigerator for 20 minutes.
5. Meanwhile, preheat oven to 160°C.
6. Clean the food processor bowl.....

EXAMPLE PROGRAM

```
//This program draws a square 100 pixel window,
//gives it a black background, draws two
//ellipses, and prints words to the console.

size(100,100);
background(0);
ellipse(0,0,60,60);
ellipse(50,50,30,30);
print(“This is some test text!”);
println(“This is some more test text!”);
println(“This is even more test text!”);
```

EXAMPLE PROGRAM

comment

```
//This program draws a square 100 pixel window,
//gives it a black background, draws two
//ellipses, and prints words to the console.

size(100,100);
background(0);
ellipse(0,0,60,60);
ellipse(50,50,30,30);
print(“This is some test text!”);
println(“This is some more test text!”);
println(“This is even more test text!”);
```

COMMENTS

p.17

```
//This program draws a square 100 pixel window,
//gives it a black background, draws two
//ellipses, and prints words to the console.
```

Comments are ignored by the computer. They let you write notes to yourself and to others who read your programs.

Two forward slashes // are used to denote a comment.

A forward slash followed by an asterisk /* allows the comment to continue until the opposite */

EXAMPLE PROGRAM

comment

```
//This program draws a square 100 pixel window,
//gives it a black background, draws two
//ellipses, and prints words to the console.
size(100,100);
background(0);
ellipse(0,0,60,60);
ellipse(50,50,30,30);
print("This is some test text!");
println("This is some more test text!");
println("This is even more test text!");
```

function

parameters

statement terminator

FUNCTIONS

p.18

parameters

function → `size(100,100);` ← **statement terminator**

Functions allow you to execute many types of actions. A function's name is usually a lowercase word followed by parentheses.

The comma-separated elements between the parentheses are called parameters, and they affect the way the function works.

Statement terminators are semicolons. They tell the computer where the end of a statement is (like a full stop in a sentence).

EXAMPLE PROGRAM

comment

```
//This program draws a square 100 pixel window,
//gives it a black background, draws two
//ellipses, and prints words to the console.
size(100,100);
background(0);
ellipse(0,0,60,60);
ellipse(50,50,30,30);
print("This is some test text!");
println("This is some more test text!");
println("This is even more test text!");
```

function

parameters

statement terminator

prints to console

PRINTING TO THE CONSOLE

p.20

```
print(“This is some test text!”);
println(“This is some more test text!”);
```

← **prints to console**

↑ **prints to console, then moves to new line**

The functions print() and println() can be used to display data while a program is running.

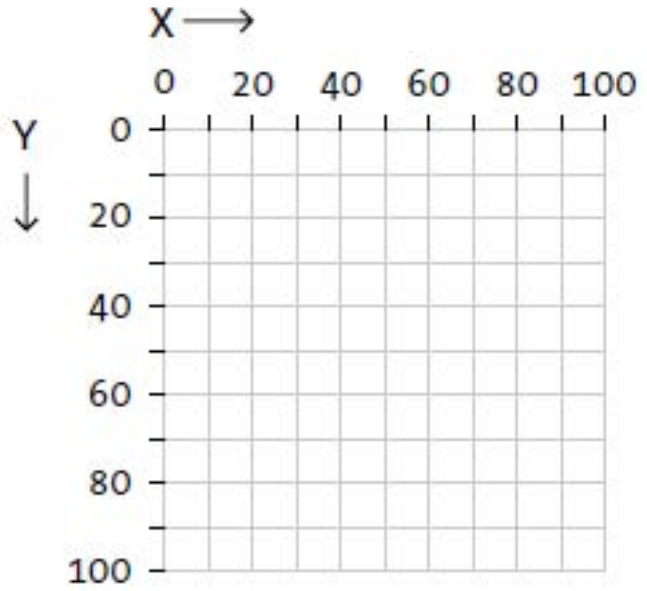
Place the desired output in quotes.

COORDINATES (p.23)

Most drawing elements in Processing use a coordinate system to define their position. This is measured in pixels.

e.g. `size(100,100);`

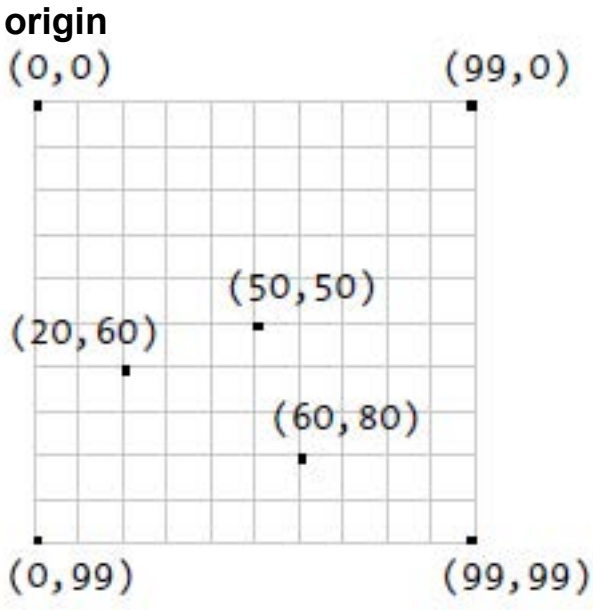
x(width) y(height)



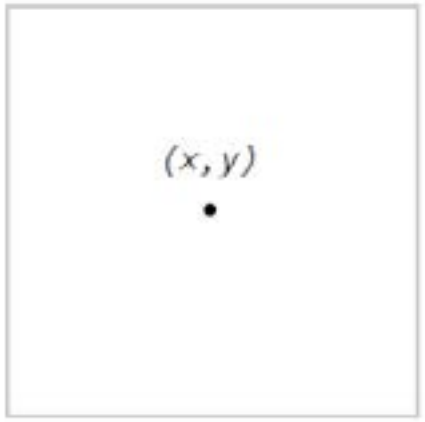
x(horizontal position) y(vertical position)

e.g. `rect(10,10,50,50);`

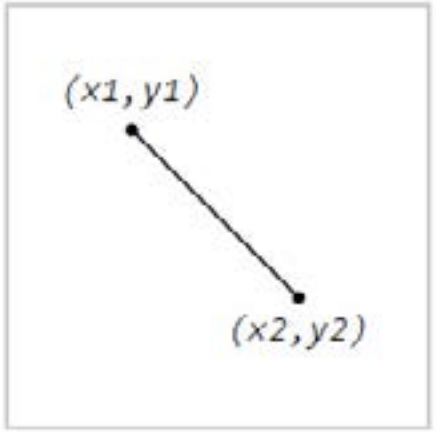
x(width) y(length)



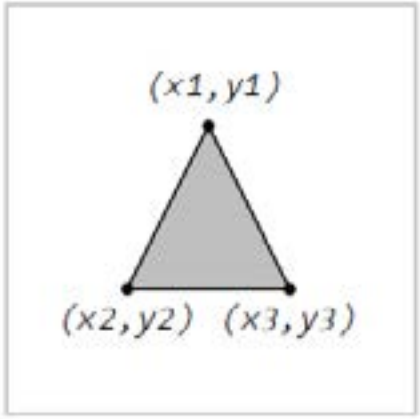
PRIMITIVE SHAPES (p.25)



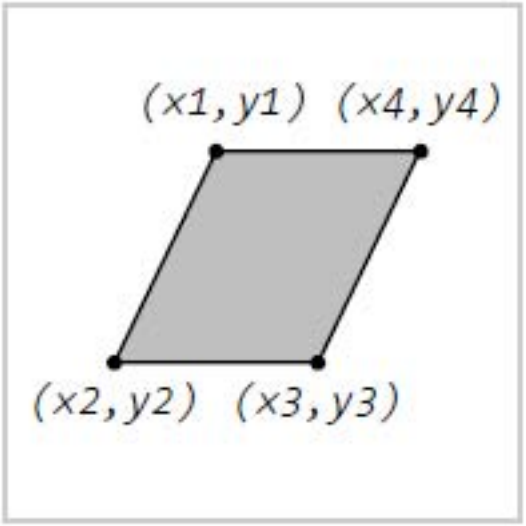

```
point(x, y)
```



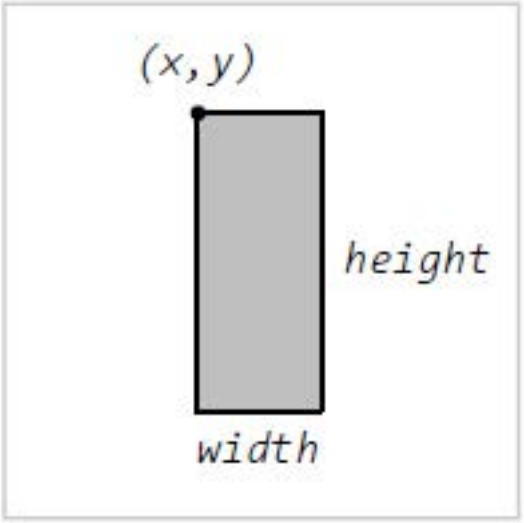

```
line(x1, y1, x2, y2)
```



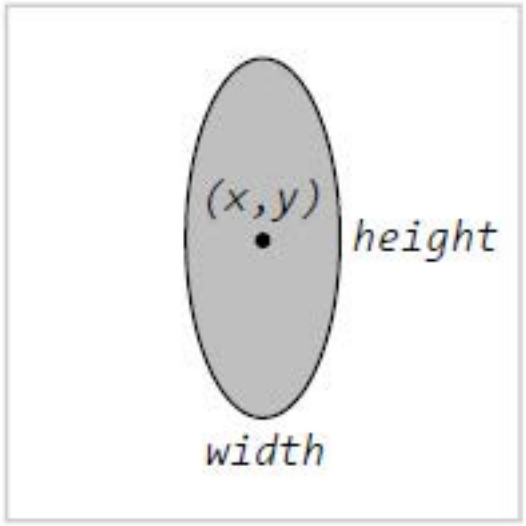

```
triangle(x1, y1, x2, y2, x3, y3)
```



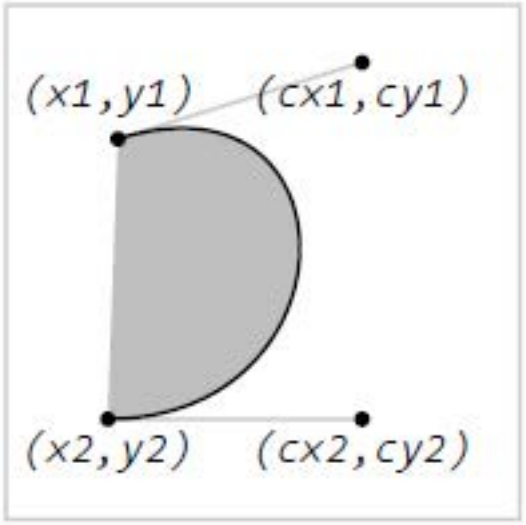

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```




```
rect(x, y, width, height)
```




```
ellipse(x, y, width, height)
```




```
bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)
```

EXERCISE

2.1. Create a composition by carefully positioning one line and one ellipse.

(p.35)

STROKE, FILL, GREY VALUES (p.31)

The fill of a drawn element can be set to a grey value between 0 (black) and 255 (white).

The same applies to background().

The stroke of an element may also be set to a grey value.

Once a fill or stroke value is defined, it applies to all shapes drawn afterward until the fill() or stroke() function is used again.

Try noStroke() and noFill().

```
background(0);
```

```
background(124);
```



```
background(230);
```

EXTRA CONTROL

For more control over these elements, such as transparency, stroke weight, and other features, see pages 32-35 in the Processing handbook.

```
background(0);
fill(255, 220);
rect(15, 15, 50, 50);
rect(35, 35, 50, 50);
```

EXERCISE

2.2. Modify the code for exercise 1 to change the fill, stroke, and background values.

(p.35)

DATA TYPES (p.37)

When writing a program, it is often necessary to store various types of data. Processing has specific ways of referring to that data.

The most basic types of data are called **primitive** types.

`boolean`
true or false

`int`
a whole number (integer)
e.g. 5

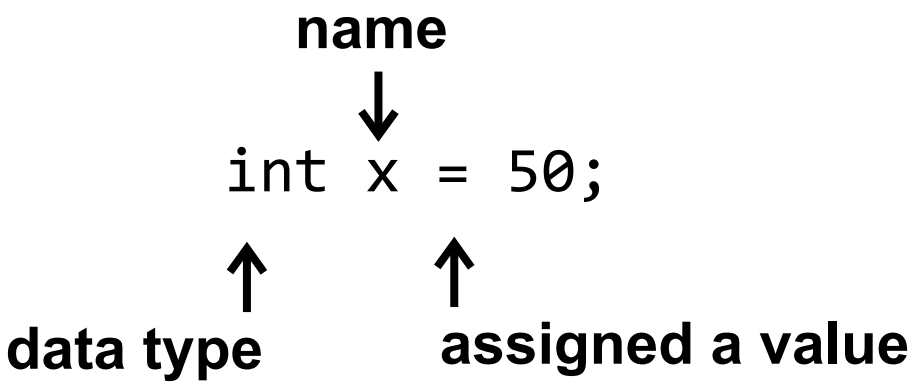
`float`
a decimal (floating point) number
e.g. 5.3

`char`
a single letter (character)
e.g. a

VARIABLES (p.38)

Variables are containers for storing data.

They must be given a data type, a name, and assigned a value:



You can now use the name of the variable to refer to that piece of data.

e.g.
```
int x = 50;
println(x);
```

WHY VARIABLES?

Variables make programs easier to read and write.

Make sure your variable names make sense!

```
ellipse(10,10,10,30);
ellipse(40,50,10,30);
ellipse(30,20,10,30);
```

vs

```
int ellipseWidth = 10;
int ellipseHeight = 30;

ellipse(10,10,
ellipseWidth,ellipseHeight);
ellipse(40,50,
ellipseWidth,ellipseHeight);
ellipse(30,20,
ellipseWidth,ellipseHeight);
```

PRE-DEFINED VARIABLES

Pre-defined variables are designated variable names that are already defined by the Processing core library.

These variables are contextual:

mouseX is a variable that is dependent upon the current x position of the mouse according to the coordinate system

width is a variable that is dependent upon the parameter placed in the size() function

```
size(100,100);
```

(width) **(height)**

EXERCISES

3.2. Make a few int and float variables. Try assigning them in different ways. Write the values to the console with println().

3.3. Create a composition that scales proportionally with different window sizes. Put different values into size() to test.

(p.41)

ARITHMETIC (p.43-47)

Programs can be controlled mathematically using operators (right) in expressions.

e.g. `int a = 1;`
`a = a + 10;`
(a is now 11)

Remember your order of operations?

`+ (add)`

`- (subtract)`

`* (multiply)`

`/ (divide)`

`% (modulus - read page 45)`

EXAMPLES

```
int a = 30;
line(a, 0, a, height);
a = a + 40;  // Assign 70 to a
strokeWeight(4);
line(a, 0, a, height);
```

```
int a = 30;
int b = 40;
line(a, 0, a, height);
line(b, 0, b, height);
strokeWeight(4);
// A calculation can be used as an input to a function
line(b-a, 0, b-a, height);
```

SHORTCUTS (p.48)

Some expressions are used so frequently that Processing includes shortcuts you can use to save time.

```
x += 5;
```
is the same as
```
x = x + 5;
```

```
x -= 5;
```
```
x = x - 5;
```

```
x++;
```
```
x = x + 1;
```

```
x *= 5;
```
```
x = x * 5;
```

```
x--;
```
```
x = x - 1;
```

```
x /= 5;
```
```
x = x / 5;
```

```
x = -x;
```
```
x = x * -1;
```

EXERCISES

4.1. Use one variable to set the position and size for three ellipses.

4.2. Use multiplication to create a series of lines with increasing space between each.

(p.50)

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)

NEXT WEEK:

ASSESSMENT BRIEF HANDED OUT & DISCUSSED

RESPONSIVE ENVIRONMENTS

TOPICS:

VERTICES, COLOUR, IMAGES, CURVES, DATA

FOR NEXT WEEK:

READINGS: SHAPE 2, MATHS 2, COLOUR 1, IMAGE 1, DATA 2

If you haven't done this week's readings, you should probably start with **STRUCTURE 1, SHAPE 1, DATA 1, MATH 1**

FOR NEXT WEEK:

EXERCISES: 1.1, 1.2, 1.3, 2.1, 2.2, 3.2, 3.3, 4.1, 4.2, 4.3, 5.1, 5.2, 5.3, 6.1

Each of your exercises should be called _1_1, _1_2, _1_3, etc.

Use GitHub to sync your changes as you make them.

Pick ONE exercise (Most experimental? Discovered something cool?) and submit to the classroom on openprocessing.org before next week's class (instructions to follow on Omnium tutorial group post)