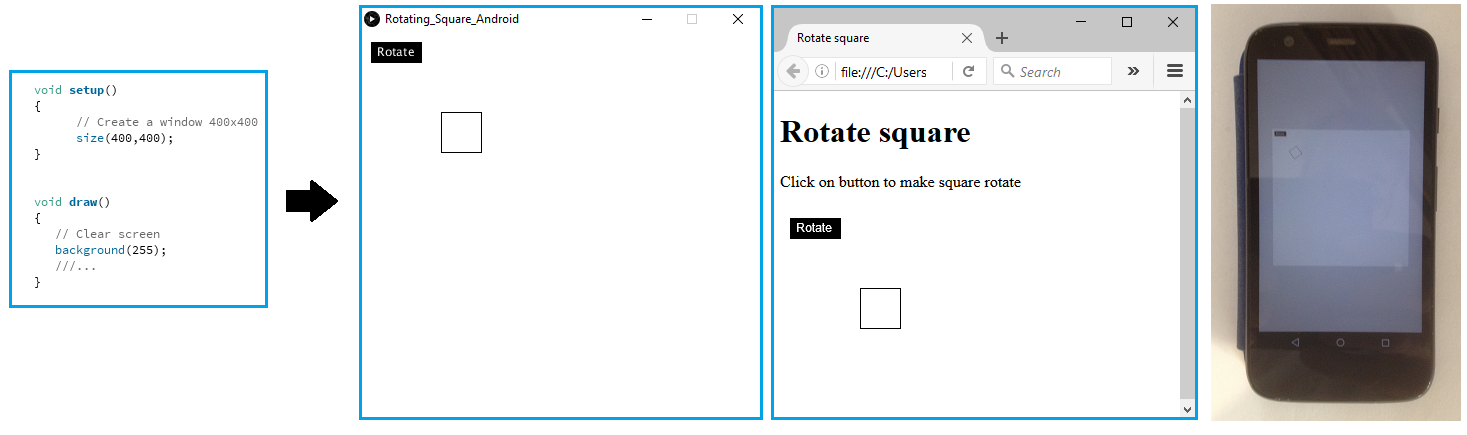
**Department of Computer Science**

**CS171 – Introduction to Computer Systems I**

**Laboratory 1: Introduction to Processing**

**Introduction:** Processing is a programming language originally developed to teach the fundamentals of programming using computer graphics as the program output rather than console/text output. More recently, it has become an effective development package used by professionals to create practical applications. The Processing language extends the Java programming language (Java code will run in the Processing environment). Processing is straightforward to install and it allows the addition of libraries to increase functionality (e.g. computer vision with cameras and 3D graphics).

Another nice feature of Processing is that it allows you develop code that runs on the PC as an executable file (e.g. an icon on desktop that you click), within a webpage on your browser (or remotely via a server) or on a smartphone. In this module, we will show you how to do all these things, but that is for another day.



Source code Runs on a PC Runs in a web browser Runs on a Smartphone

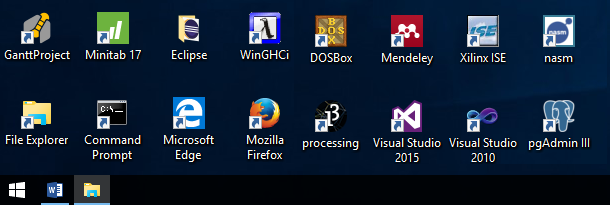
**Figure 1:** A Processing program running as a stand-alone executable on a PC, within a web page and as an Android app.

**Starting here:** Log into Windows, start the Firefox browser and login to Moodle 2018 and choose module CS171[A]. You can use cut and paste from the Word document to speed things up (although this may fail with pdf versions of the document as they can contain rogue (and invisible) characters).

**Part1: Hello world**

Whenever you embrace a new programming language the first program you write is typically “Hello world”. This program provides a very simple confidence test that the software tools and your understanding of the syntax of the language is correct.

Start the Processing IDE (Integrated Development Environment) by click on the Processing icon found on the Windows desktop.



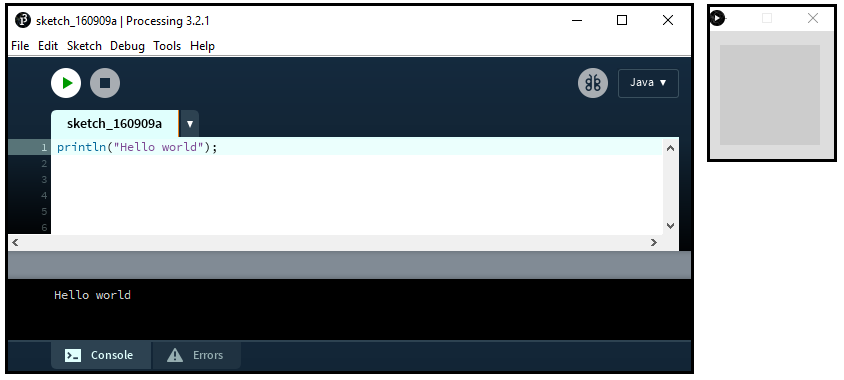
1

**Figure 2:** Launch the Processing IDE from the Windows desktop

Once the Processing IDE has loaded, you will be presented with a window into which you can type your source code. Enter the following single line into the window.

println("Hello world");

Aside: Don’t worry too much about how the code works for now. For those who want to know, the println()is an instruction to Processing to create runnable code to print text to the console screen, it is known as a “method” or “function”. In maths (and computing), a function takes one or more inputs and delivers an output. In this case, we are giving the function the input string of characters “Hello world”. When run the string of characters is passed to the println() method that then prints the string to the console. The semicolon is used to identify the end of a statement (or line of code). The println() method can be used again and again in the same program to display different messages, this software engineering method, known as code-reuse, makes for efficient programs.



Applet

Window

4

3

2

**Figure 3:** Hello world confidence test, output appears in console rather than applet window for now.

Run the code by pressing the  button (this compiles and then runs). You should see the phrase “Hello world” echoed in the console window. Well done you have written and run your first Processing program and at the same time commissioned the IDE.

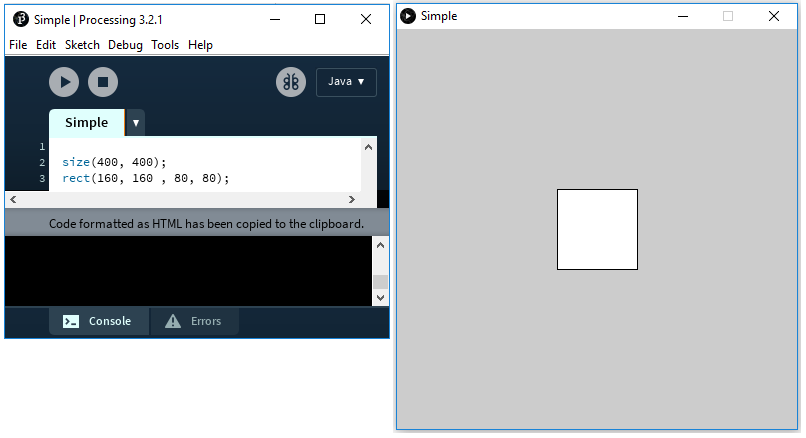
**Part 2: Adding some graphics**

We could keep going with console/text based programming; however, the aim of Processing is to allow you to learn programming within a multimedia environment (graphics and sound). Just keep editing the existing program rather saving each new development. To create a window in which to draw use the *size(width, height)* instruction. The numbers refer to the width and height of the window measured in pixels (picture elements). To draw a rectangle use the *rect(x,y,width,height)* command. The origin (0,0) for the Processing window is the top left hand corner of the screen, x increases across the window and y increases down the window.

Enter and then run the following program that creates a window 400x400 pixel in size with a rectangle located at pixel location (160,160) with a width of 80 pixels and height of 80 pixels.

 size(400, 400);

 rect(160, 160 , 80, 80);



80

80

(160,160) 0,0

(0,399) 0,0

(399,0) 0,0

(399,399) 0,0

(0,0) 0,0

**Figure 4:** Two line program to create a 400x400 window and display a rectangle in the centre.

**Part 3: Adding some colour**

You can tell Processing how to fill the rectangle when it is being drawn, if not set it defaults to solid white. You can change the colour used to fill shapes using the *fill(red, green, blue)* instruction.

**Aside:** Our eyes have three types of colour sensor cells (known as LMS or RGB cones). Any colour can be simulated by using the correct mix of red, green and blue light. For example, the effect on the eye of yellow light can be simulated by using a 50:50 mix of red and green light. Two different mixes of light that have the same effect on the eye are known as metamers.

Consider the following commands and then modify the code to produce a dark yellow (brown) square on the screen.

 fill(255, 0, 0); // Red

 fill(0, 255, 0); // Green

 fill(0, 0, 255); // Blue

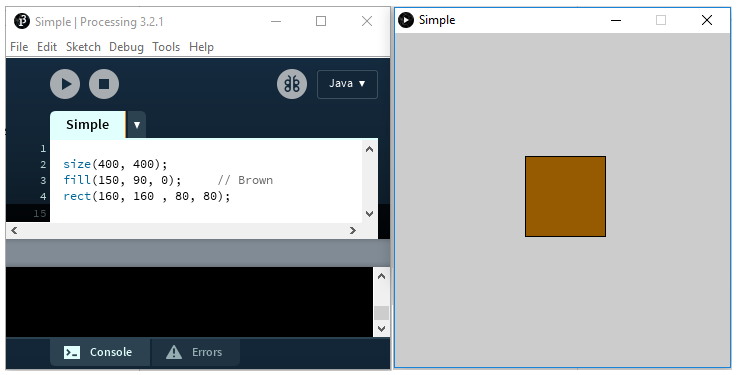
 fill(128,128,128); // Grey

 fill(0,0,0); // Black

 fill(255,255,0); // Red+Green=Yellow

The colour values are contained in 8 bit numbers, the highest value they can contain is (28-1)=255, so you can consider 255 as full power and 0 is off.

The code contains comments // which just help us remember what it does (the compiler ignores everything after a comment).



**Figure 5:** Three-line program to display a brown rectangle in the centre a small window.

**Aside:** Integer numbers are stored in a format known as binary inside the computer. The numbers used to store the colour values use 8 bits (binary digits). The 8 bits are known collectively as a byte. In decimal notation we use units, tens, hundreds, thousands to represent a number. A decimal number written as 346 means (3x100)+(4x10)+(6x1) or (3x102)+(4x101)+(6x100). Inside a computer it is much easier to build a circuit that can be in one of two states (on or off) rather than ten possible states (1 volt, 2 volts, 3 volts …, 10 volts), for this reason we use binary. Instead of units, tens, hundreds we use ones, twos, fours, eights. This approach is known as base 2 (binary) rather than base 10 (decimal). For example, the binary number 00100101 has the decimal value 37,

` 0010,0101 binary = (0x27)+(0x26)+ (1x25)+(0x24)+(0x23)+(1x22)+(0x21)+(1x20)

= (0x128)+(0x64)+ (1x32)+(0x16)+(0x8)+(1x4)+(0x2)+(1x1)

= (1x32)+(1x4)+(1x1)= 37 decimal

The biggest binary number you can create is 1111,1111 which is 255

1111,1111= (1x128)+ (1x64)+ (1x32)+(1x16)+(1x8)+(1x4)+(1x2)+(1x1) =255 decimal

The smallest binary number is 0,

0000,0000= (0x128)+(0x64)+ (0x32)+(0x16)+(0x8)+(0x4)+(0x2)+(0x1) =0 decimal

The values passed to the fill command are single bytes which limits the range of values to [0,255] in decimal.

**Part 4: Drawing primitives**

Processing can display a number of different shapes on the window. These shapes are called “drawing primitives”. Shapes that are more complex can be built up using the primitives.

Enter each of the following lines of code and see the shape that it produces, run each line separately as they are designed to be on the same centre.

**Aside:** You can cut and paste source code from a word document. The pdf document won’t work as you may paste invisible characters into the editor that Processing won’t be able to compile.

 size(400, 400);

 fill(255, 0,255);

Triangle

 triangle(100,300,300,300,200,100);

Rectangle

 rect(120,120,160,160);

Quadrilateral

 quad(120,270,340,280,320,80,90,140);

Ellipse

 ellipse(200,200,200,100);

Arc

 arc(200,200,200,100,PI/2,TWO\_PI);

 strokeWeight(10); // Width

Line

 stroke(128,0,255); // Colour, RGB

 line(100, 100, 300, 300);

 strokeWeight(20); // Width

Point

 stroke(0,0,0); // Colour, RGB

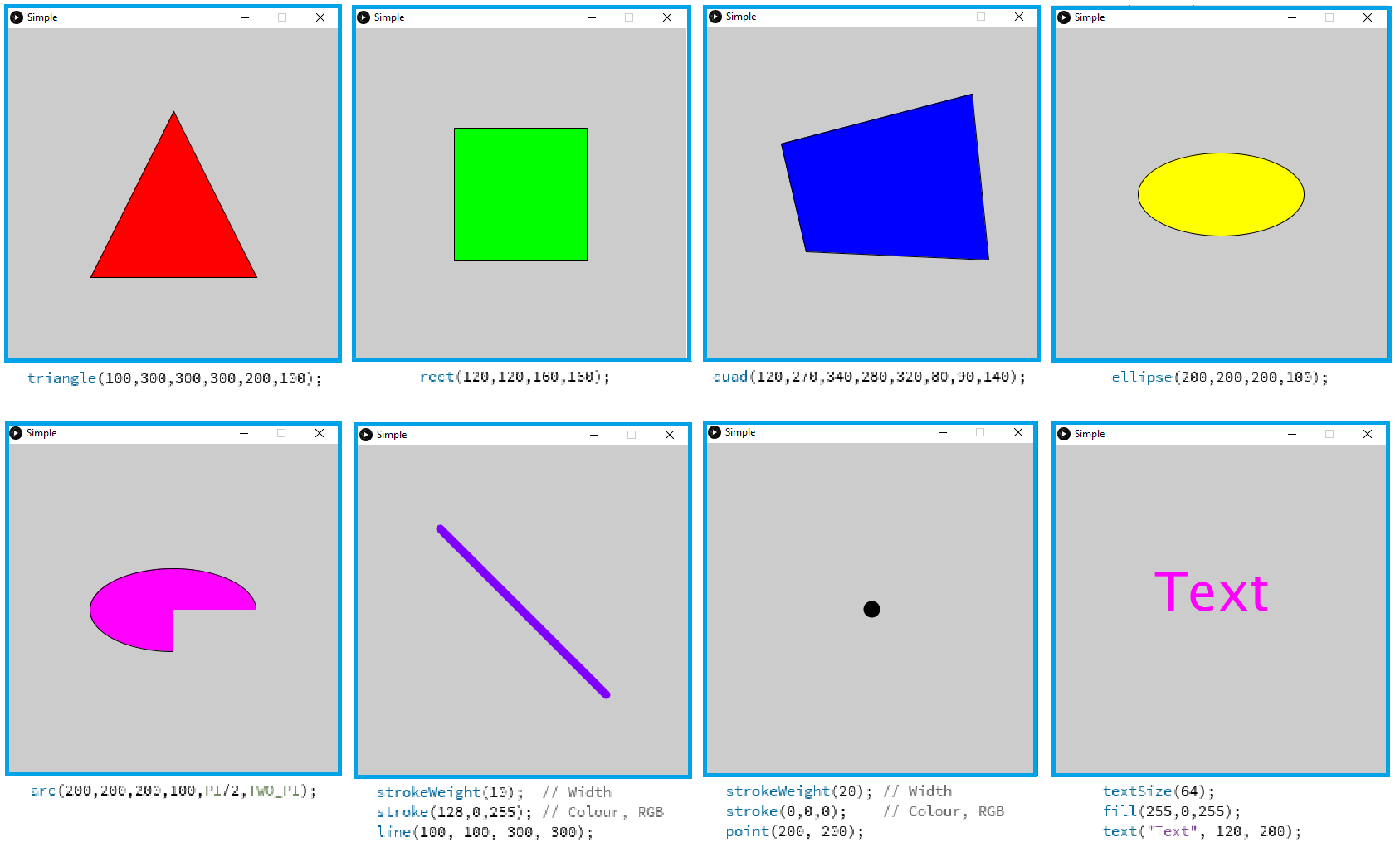
 point(200, 200);

 textSize(64);

Text

 fill(255,0,255);

 text("Text", 120, 200);



**Figure 6:** Drawing primitives, triangle, square, quadrilateral, ellipse, arc, line, point and text.

**Part 5: Animation**

So far, we have used processing to create a single image. To animate a scene you need to create a sequence of images. In processing there are two methods that work together to achieve this, *setup()* and *draw().* *Setup()* runs once at the start of the execution of your program, *draw()* runs repeatedly after this. Each time that *draw()* is run an internal variable (a place with a name that stores a number) called *frameCount* increases by one. Normally you have to declare a variable (set it up for use in your program), however this has already been done for you with *frameCount* as it is part of Processing.

Enter the following program,

void **setup**()

{

Run once at start

 size(400, 400); // Create window

}

void **draw**()

Keep repeating (60 times per second) after setup() has run once.

{

  background(255,255,255); // Clear the screen

  fill(255,0,0); // Red

  rect(frameCount,180,40,40); // Draw rectangle

}

When the program is run, a red square can be seen to cross the screen.

**Aside:** if you replaced the rectangle line above with the following line then the rectangle moves across the screen repeatedly, the % symbol means remainder after integer division (modulus). Taking the modulus of a number that is always increasing returns a number that cycles through a range of values. The framecounter increases 60 times per second, ten seconds after the program has started its value will be 600. Taking modulus 360 of the frame counter returns a value in the range [0,359].

rect(frameCount%360,180,40,40); // Draw rectangle

See if you can change the program to make the square move up the screen repeatedly.

**Part 6: Mouse input**

The mouse cursor position can be obtained using the Processing variables, mouseX and mouseY. Enter the following program and reassure yourself of how it works.

void **setup**()

{

 size(400, 400); // Create window

}

void **draw**()

{

  background(255,255,255); // Clear the screen

  fill(255,0,0); // Red

  rect(mouseX-10,mouseY-10,20,20); // Draw rectangle

}

**Aside:** The value -10 is taken away from the mouse co-ordinate before being passed to the rect() method so that the centre of the square follows the mouse pointer. Without the -10 offsets the top left corner of the square would follow the mouse pointer.

**Part 7: Random numbers**

Another method that may be of use is random(), this returns a random number between 0 and the number you pass to it. Run the following program and justify its operation.

void **setup**()

{

 size(400, 400); // Create window

}

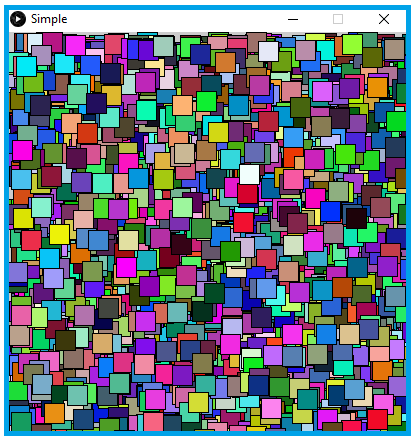
void **draw**()

{

  fill(random(255),random(255),random(255)); // Random colour

  rect(random(400),random(400),20,20); // Draw rectangle

}



**Figure 7:** Drawing random coloured squares at random positions.

**Part 8: Your turn….**

Write a program that animates a small object of your own design, (e.g. a bee, boat, car, plane etc). The object should be made from different coloured primitives. The object should move, either randomly, or using the mouse input, or using the frame counter (your choice). The completed program should be saved on your X:\ drive to keep it as a permanent record.

You should submit the code you created using the Moodle entry “Lab Submission 1”.

Complete the quiz.

Both quiz and code must be completed before next Monday at 10:00AM. They will be graded.

**Part 9: Submitting your work.**

On Moodle, click on the assignment Lab submission 1

Submit your program by dragging the .pde file from the windows file explorer window to the moodle submission panel.

SB/2020