Diploma

DCR 2284- Creative Computing

UOW MALAYSIA KDU

PART OF THE UNIVERSITY OF WOLLONGONG AUSTRALIA GLOBAL NETWORK

Diploma in Computer Studies September 2021

Welcome to Creative Computing

DCR2284

Learning Objectives

- At the end of the course, students will be able to:
  - CO1: Describe the creative concepts in mathematics and computing.
  - CO2: Explain the importance origins of geometry to develop motion, images and sound.
  - CO3: Build the Processing application to construct shapes and objects.
  - CO4: Write the coordinate transformations for motions using Processing..

Course Operation

Ground Rules

- Please keep up with the class content.
  - Check the Teaching Plan and Moodle.
- Adhere to due dates (hardcopy and softcopy)
- No Copying. Honesty is the best policy.
  - Plagiarism is an academic offence!
  - Plagiarised work will be zero-rated.

Ground Rules

- Let's make sure learning is possible! :-)
- Punctuality and Attendance:
  - Up till 9:05pm = PRESENT
  - Up till 9:20pm = LATE
  - After 9:20pm = **ABSENT**
  - (Remember the college attendance requirements and exam barring policy! Do not mention about Parking issues)
- Courtesy to your classmates:
  - No eating in class. Drinking water is allowed.
  - Put your mobile phones on silent mode.
  - If there is too much noise going on, lecture will be stopped until the situation is calmed and conducive for class.

Lecture Contents

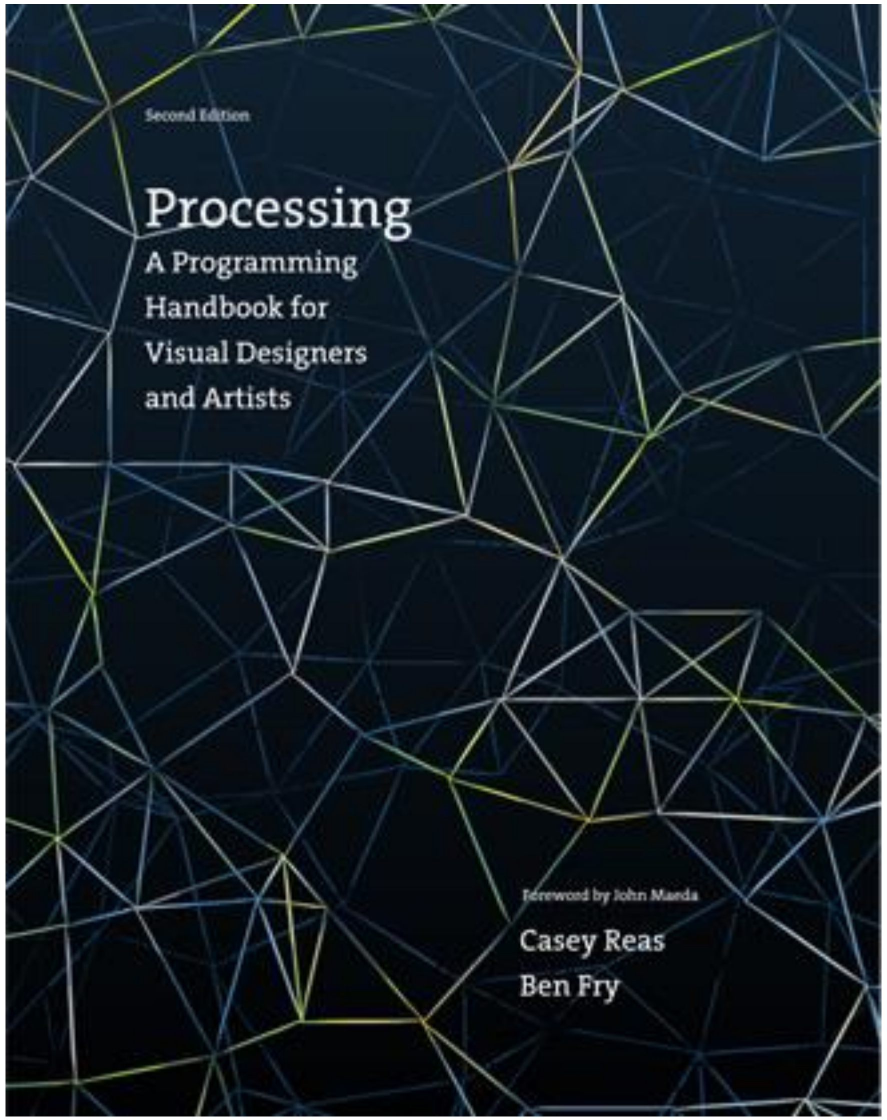
- I won't go over everything in the slides during lecture!
- You need to read the given academic research articles to understand further the topics!
- Focus is on understanding and practice during lectures and labs
  - Please participate actively in the exercises – I guarantee you it will be useful. :-)
- Lecture slides contains Research article links, examples, video's which you are encouraged to try out yourself outside lecture hours.

Assessments

- Assignment: 1
  - ❑CO1, CO2
  - ❑Individual Work [Single Task]
  - ❑Weighted 10%
- Assignment: 2
  - ❑CO3, CO4
  - ❑Group Work [Multiple Tasks]
  - ❑Weighted 20%
- Midterm Examination :
  - ❑ CO1, CO2, CO3
  - ❑Weighted 20%
- Final Examination:
  - ❑Weighted 50%

Computational Tools

https://processing.org/download/ (FREE)


Second Edition

Processing

A Programming Handbook for Visual Designers and Artists

Foreword by John Maeda

Casey Reas

Ben Fry

First lecture!

Learning Objectives

- At the end of the course, students will be able to:
  - CO1: Describe the creative concepts in mathematics and computing.
  - CO2: Explain the importance origins of geometry to develop motion, images and sound.
  - CO3: Build the Processing application to construct shapes and objects.
  - CO4: Write the coordinate transformations for motions using Processing..

Overview

- Introducing Creative Computation
- Introducing ourselves
- Introducing Processing
- Drawing
- Living code

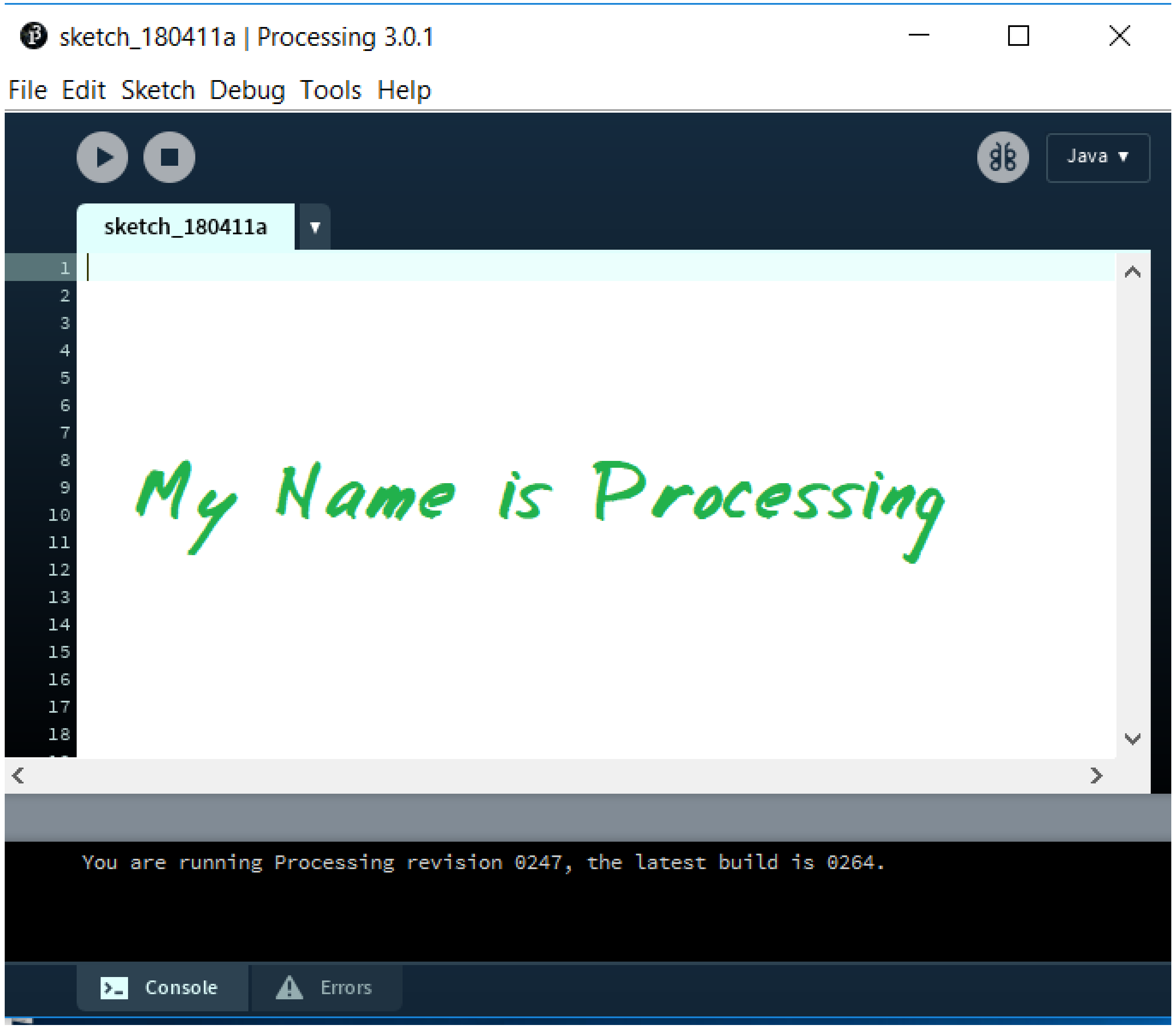
Introducing ourselves

- Office: 3rd Floor
- Website: http://cri.kdupg.edu.my/research-committees/
- Email: jjoshua@kdupg.edu.my/ +6012 578 0009 [Office Hours]
- Consultation Hours :

  Team Channels: Wednesday 2:00PM to 4:00PM

- Note: CHARGE YOUR LAPTOP'S FOR THE CLASS ROOM SESSIONS.

Introducing Processing


sketch_180411a | Processing 3.0.1
File Edit Sketch Debug Tools Help
Java
sketch_180411a
My Name is Processing
Console
Errors

```
You are running Processing revision 0247, the latest build is 0264.
```

Say "Hello"

- Type the following text into the text area:
```
println("Hello, World!");
```
- Press the play button at the top left of the window.
- Look at the black area below the text area.
- Feel the sheer power of programming.

The interface

- We'll learn the Processing interface as we go, but for now notice:
  - The play and stop buttons
  - The console and errors tabs
  - The Help menu
  - The File > Examples menu

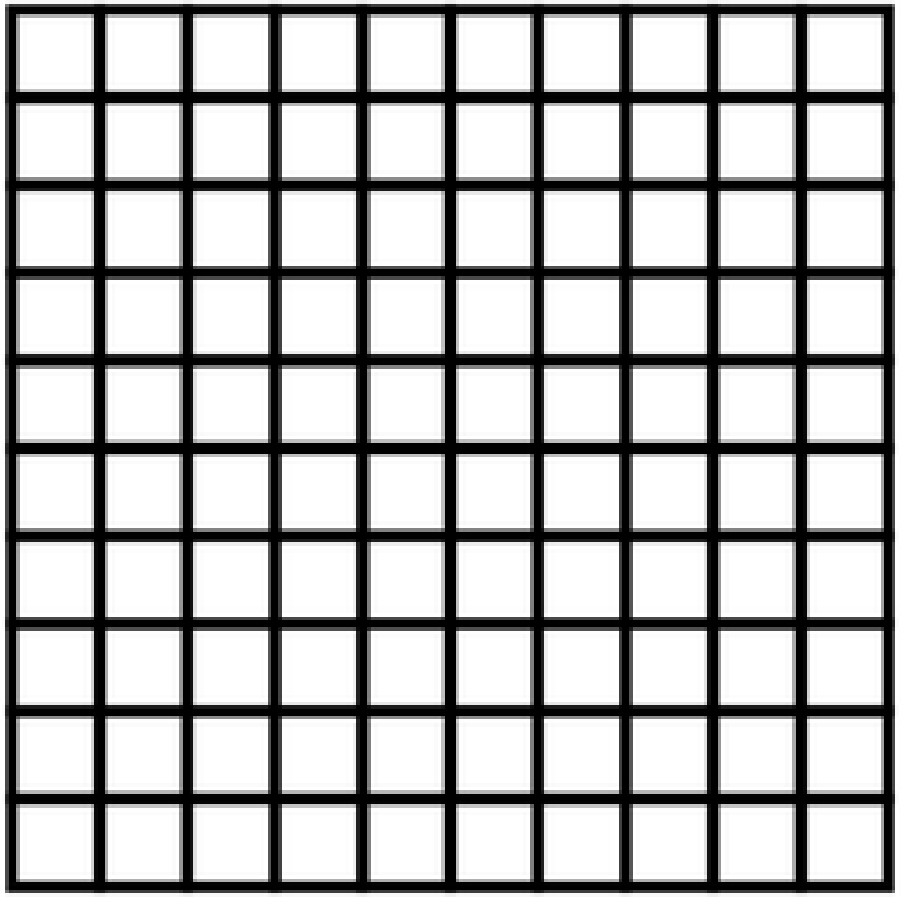
Why Processing?

- It is a programming environment made specifically for people to learn how to program from a creative perspective
- It is actually a more friendly window into Java, which is a Real Serious Programming Language™
- It is free, open-source, and has an excellent community around it

Drawing

The grid

- Computer screens are divided up into *pixels*
- Everything we see on the screen involves setting the colours of the pixels
- By changing the colours of pixels over time, things appear to move!
- If you can zoom your screen, you can even see the individual pixels!

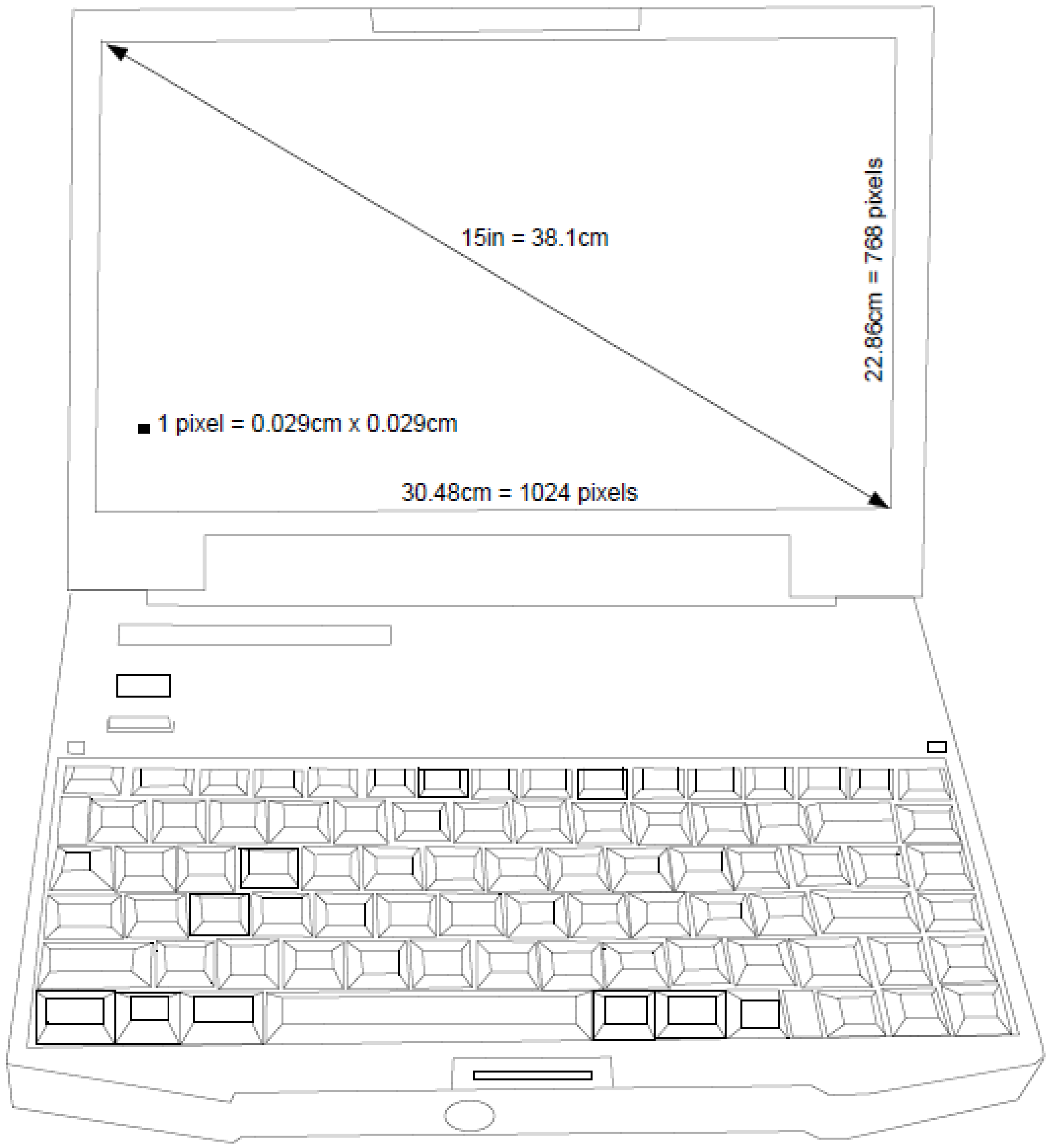


Where on the grid?

- In a graphics application you specify what pixel you want to change the colour of by just clicking on it with a pencil or paintbrush...
- But if you're programming you're working in text...
- ... so how would you refer to a specific pixel?

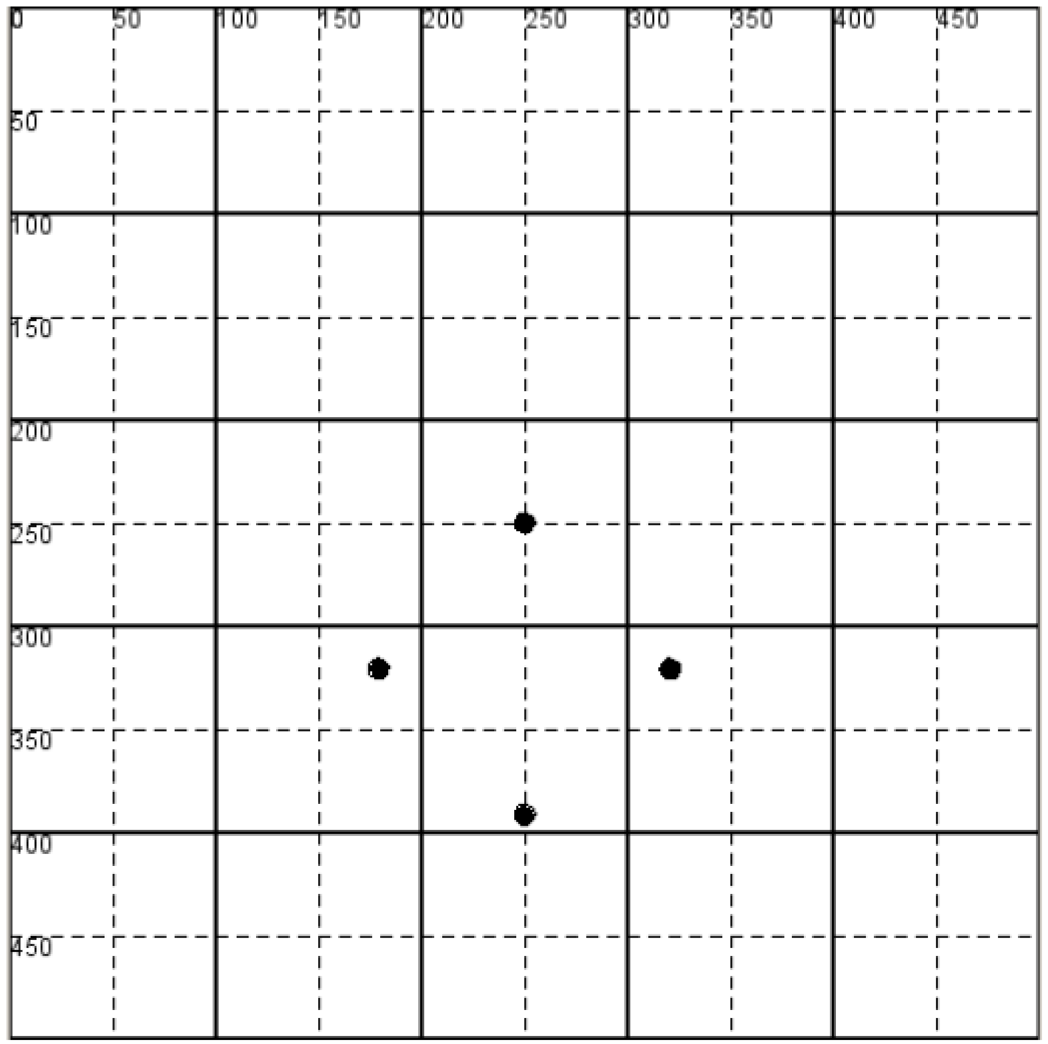
Where on the grid?

- In a graphics application you specify what pixel you want to change the colour of by just clicking on it with a pencil or paintbrush...
- But if you're programming you're working in text...
- ... so how would you refer to a specific pixel?
- That's right! It's our friend *coordinates*!



$$height = \sqrt{38.1^2/(1+(4/3)^2)} = 22.86cm$$

$$width = 4/3 \times 22.86cm = 30.48cm$$



Screen Coordinates

Origin in Processing is the point in the top left corner, that is point (0,0). The grid is made by placing lines at a regular interval in both the horizontal and vertical directions.

Life on the grid

Just like on a graph we have an *x-axis* and a *y-axis*

Life on the grid

- Just like on a graph we have an *x-axis* and a *y-axis*
- The x-axis starts at the *left* of the screen (or window)

Life on the grid

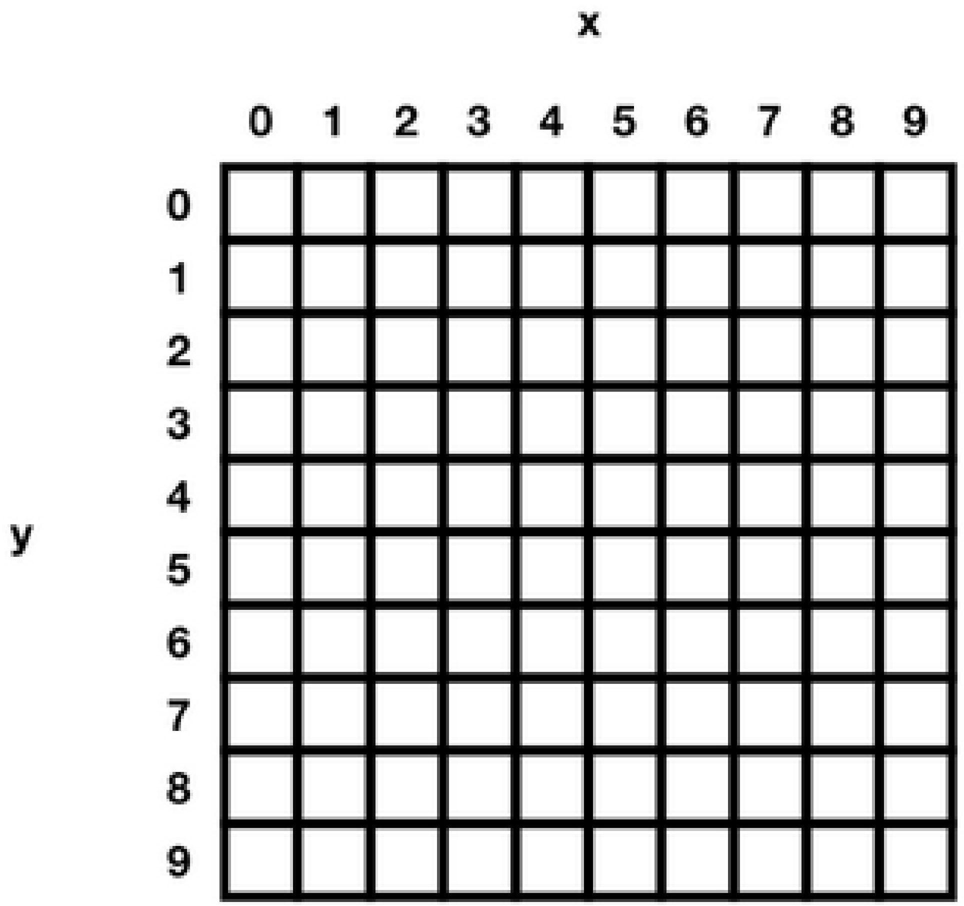
- Just like on a graph we have an *x-axis* and a *y-axis*
- The x-axis starts at the *left* of the screen (or window)
- The y-axis starts at the...

Life on the grid

- Just like on a graph we have an *x-axis* and a *y-axis*
- The x-axis starts at the *left* of the screen (or window)
- The y-axis starts at the... *top* of the screen (or window)

Life on the grid

- Just like on a graph we have an *x-axis* and a *y-axis*
- The x-axis starts at the *left* of the screen (or window)
- The y-axis starts at the... *top* of the screen (or window)
- And we count pixels along the axes from *zero* up


x
0
1
2
3
4
5
6
7
8
9
y
0
1
2
3
4
5
6
7
8
9

There on the grid!

- We specify the location of a pixel on the screen (or in our window) by giving its coordinates
- Like in maths etc. we specify coordinates as *(x,y)*

There on the grid!

- We specify the location of a pixel on the screen (or in our window) by giving its coordinates
- Like in maths etc. we specify coordinates as *(x,y)*
- So where would (5,2) be?

There on the grid!

- We specify the location of a pixel on the screen (or in our window) by giving its coordinates
- Like in maths etc. we specify coordinates as *(x,y)*
- So where would (5,2) be? *Exactly*. *6* pixels to the right from the left edge and *3* pixels down from the top edge

There on the grid!

- We specify the location of a pixel on the screen (or in our window) by giving its coordinates
- Like in maths etc. we specify coordinates as *(x,y)*
- So where would (5,2) be? *Exactly*. *6* pixels to the right from the left edge and *3* pixels down from the top edge
- And where would (525,100) be?

There on the grid!

- We specify the location of a pixel on the screen (or in our window) by giving its coordinates
- Like in maths etc. we specify coordinates as *(x,y)*
- So where would (5,2) be? *Exactly*. *6* pixels to the right from the left edge and *3* pixels down from the top edge
- And where would (525,100) be? *Uh-huh*. *526* pixels to the right from the left edge and *101* pixels down from the top edge.

Getting to the **`point()`**

- We saw just now that we can tell Processing to draw a point using the command `point()`

Getting to the **`point()`**

- We saw just now that we can tell Processing to draw a point using the command `point()`
- In programming we call that *calling a function*
- The function is called `point()` and we *call* it to ask it to draw a point (make sense?)

`point (x,y);`

- First we have the *name* of the function, which is point
- In a perfect world the name of the function tells us what it does!
- The name of a function is *case sensitive* and *spelling sensitive*
- Welcome to programming, where the tiniest inaccuracy will break everything!

point (x,y);

- Next we have an *opening parenthesis* (or *bracket* if you prefer)
- This tells Processing we are about to give it the information it needs to carry out the function.
- In the case of our point that will be the *coordinates* of the point

- There are a *lot* of parentheses in programming and if you miss one, things break. Sorry!

point (x,y);

- Now we give the function the first piece of information it needs to draw the point, known as a *parameter*
- Which is the *x coordinate*
- We would need to replace this ‘’x’’ with an actual number, like 10
- I'm writing it as “x” here because it better shows us what the nature of this parameter is.

point (x,y);

- Because we have *multiple parameters* we need a *comma* next
- The parameters of a function are always separated by commas like this

point (x,y);

- Now we give the function the final piece of information it needs to draw the point, the second *parameter*
- Which is the *y coordinate*
- Again, we would replace with this an actual number like 55

point (x,y);

- Now we've finished giving all the parameters the function needs
- So we *close* the parentheses we *opened* earlier to say "that's the end of the parameters!"
- Making sure your parentheses *match* is really important
- And yet we will all, many times, forget to do so
- Because we are humans
- Right?

point (x,y);

- Now we meet an aspect of coding that nobody really enjoys
- The semicolon ;
- In programming, we put a semicolon at the end of instructions we give the computer
- It basically means "I'm finished telling you what I want you to do for this line!
- It's so, so, *so* easy to forget semicolons

point (x,y);

- Now we meet an aspect of coding that nobody really enjoys
- The semicolon ;
- In programming, we put a semicolon at the end of instructions we give the computer
- It basically means "I'm finished telling you what I want you to do for this line!
- It's so, so, *so* easy to forget semicolons
- But please don't

point (x,y);

- That was kind of exhausting, but that's how we call a function in Processing
- We give its name, specify the parameters, and end with a semicolon
- So now we can read something like point(10,15); and know what it means…

`size ()` *matters*

- So far we've been drawing our points on the default size window for Processing, which is 100x100
- That's kind of small

`size ()` matters

- So far we've been drawing our points on the default size window for Processing, which is 100x100
- That's kind of small
- Fortunately we can specify the size of window we want with a command called...

`size ()` *matters*

- So far we've been drawing our points on the default size window for Processing, which is 100x100
- That's kind of small
- Fortunately we can specify the size of window we want with a command called... `size ();`

`size ()` *matters*

- So far we've been drawing our points on the default size window for Processing, which is 100x100
- That's Fortunately we can specify the size of window we want with a command called... `size ();`
- We use like this:

```
size (w,h);
```

- Where w is the width of the window we want, and h is the height of the window we want.

Shapes

So you want to draw a line…

- What do you think the function would be called?

So you want to draw a line…

- What do you think the function would be called?

```
line()
```

So you want to draw a line…

- What do you think the function would be called?

```
line()
```

- And what parameters does it need?

So you want to draw a line…

- What do you think the function would be called?

```
line()
```

- And what parameters does it need?

- It needs the starting coordinates and the ending coordinates!

- So how will it look with parameters?

So you want to draw a line…

- What do you think the function would be called?

```
line()
```

- And what parameters does it need?

- It needs the starting coordinates and the ending coordinates!
  - So how will it look with parameters?

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

So you want to draw a line…

- What do you think the function would be called?

```
line()
```

- **And what parameters does it need?**

- It needs the starting coordinates and the ending coordinates!

  - So how will it look with parameters?

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

line(0,0,100,100);
```

So you would also like to draw a rectangle…

- What would the function be called?

So you would also like to draw a rectangle…

- What would the function be called?

```
rect ()
```

So you would also like to draw a rectangle…

- What would the function be called?

```
rect ()
```

- And what would its parameters be?

So you would also like to draw a rectangle…

- What would the function be called?

```
rect ()
```

- And what would its parameters be?
- There are a few different and reasonable options here, but the one Processing uses by default is

```
rect(x,y,w,h);
```

That is, the (x,y) coordinates of the *top left corner* of the rectangle, and then its *width* and *height*

What about a circle?

- What would the function be called?

What about a circle?

- What would the function be called?

```
ellipse()
```

- And what would its parameters be?
- Again, there are different options, but the default in Processing is

```
ellipse(x,y,w,h);
```

- The (x,y) coordinates of the *centre* of the ellipse, and then its *width* and *height*

And so on and on!

```
point(x,y);
line(x1,y1,x2,y2);
rect(x,y,w,h);
ellipse(x,y,w,h);
triangle(x1,y1,x2,y2,x3,y3); // New!
quad(x1,y1,x2,y2,x3,y3,x4,y4); // New!
```

However...

- A question arises: "How the heck am I supposed to *know* the names and parameters and effects of all this stuff???"

However...

- A question arises: "How the heck am I supposed to *know* the names and parameters and effects of all this stuff???“
- It's a good question, and the answer for us is *Processing reference*
- In fact, *no* programmer "just knows" all the possible functions and their parameters
- Getting used to reading the reference is a crucial skill

Different ways to draw the same shapes

- As we saw a few times, there are different ways to specify something like a rectangle
- You might give the top-left corner and width and height
- But you might also give the center and width and height
- Or you might even give the top-left corner and the bottom-right corner
- And you might even have a preference!
- Processing gives us functions for us to *tell* it which style we like...

`rectMode()`

rectMode (CORNER);

The *default* we're used to (top-left corner and dimensions)

rectMode(CENTER);

Instead of the top-leftr corner the x and y in rect(x,y,w,h) now specify the *center* of the rectangle.

rectMode(CORNERS); Now you write it rect(x1,y1, x2, y2) where the first coordinates are the top-left corner and the second coordinates are the bottom-right corner.

ellipseMode ()

- We can do all the same things with the same results to set different ways of drawing an ellipse() as well:

ellipseMode(CENTER);
The default we're used to (draw from the center)

ellipseMode(CORNER);
Specify the top-left 'corner' of the ellipse

ellipseMode(CORNERS);
Specify the top-left and bottom-right 'corners' of the ellipse.

Grey is a default colour?

- All the points, lines, and shapes we've drawn have had a black outline and a white fill on a grey background
- But we can specify what shade of grey Processing will use for these things with more functions!

`background(shade);` sets the background shade

`fill(shade);` sets the fill shade

`stroke(shade);` sets the line shade (includes the lines around the edges of shapes)

256 shades of grey

- Shades of grey in Processing are just numbers
- Specifically you can use numbers between 0 and 255
- 0 means black
- 255 means white
- 127 means a mid-grey

And so...

- We can put our numbers meaning different greys into our functions for setting background, fill, and stroke!
- `background (0);` means set the background to black
- `fill(220);` means set the fill to very light grey
- `stroke(50);` means set the stroke to a kind of charcoal grey

- Let's try!

Colourful Dreamy Nights!

- We're not actually limited to making arthouse movie-style things in black and white
- If we use *three* numbers instead of one, we can set colours instead
- What do you think the three numbers refer to?
- Exactly, the *red*, *green*, and *blue* amounts for our colour
- That is, we're using the *RGB colour model*

So…

- `background (255,0,0);` sets the background to pure red

- (CHINESE NEW YEAR IS COMING "RED")

```
background (0,0,255);  ??
background (85,107,47);  ??
```

So…

- `background (255,0,0);` sets the background to pure red
- (CHINESE NEW YEAR IS COMING “RED”)

```
background (0,0,255);   ??  Pure blue
background (85,107,47);  ?? Olive--green
```

Finding Colours

- In Processing in the **Tools** menu you'll find a **Color Selector** that you can use to pick a colour and see its RGB values
- You can do similar things in applications like Photoshop, which may also give you an 'eyedropper' to pick colours from your screen
- You can also Google 'RGB colour picker' and it will bring one up
- Plenty of ways to find colours

Losing Colours

- You might want to make things transparent sometimes - e.g. give them *no colour at all*
- You can do this with the following functions:

`noFill ();` means have transparent fill

`noStroke();` means have transparent stroke

Class Activity

- What will we see on our screen with this code? (Try not to cheat by typing it into Processing!)

```
size(640,480);
background(100,0,0);
fill(255);
rect(0,0,250,250);
noStroke();
fill(0,0,250,250);
ellipse(250,250,250,250);
background(0,100,0);
```

Points to Remember!

- We're now able to see "behind the scenes" of software
- One thing we know now is that Processing doesn't really know about "rectangles", it just knows how to set a particular set of pixels to a particular colour
- If we wanted that rectangle to move, *we* would have to write the code to move it

Living Code

DEAD Code

- So far the code we've written just runs once and stops
- Processing carries out our instructions from the top to the bottom and that's that
- But that's not going to lead to especially interesting programming
- And in particular we're not going to be able to make something *interactive* this way
- *So what do we need?*

Three Tips for Interactive Code!

- Generally speaking there are three key elements involved in interactive applications

1. We *set up* the initial conditions of the program
2. We *update* the state of the program over and over to carry out the program's instructions over time
3. We *handle events* like user and other input

Set up

- Before a program really gets going, we often want to set some starting conditions that will help to define how it runs

In a game, for example, we might need to create the player's avatar, put it in front of the door of the dungeon, load the sound effects of the player's footsteps, configure the physics parameters that will be applied during play, etc.

- These are all things we want to do *once* at the *start* of the program

Update

- Once the program is ready, we want it to keep running until it's time to stop
- The way we tend to do this is with an *update loop*, which runs code for making the program work over and over again (once per frame)

Handle events

- While a program is running, we want to be able to react to input either from the person using it, or from other sources
- Sometimes we can do this in the *update loop* and sometimes we have special functions called *event handlers* which run only when a specific kind of event happens (like a mouse click, say)

Set up, update, handle events

- `So one more time`
- We *set up* the conditions for our program
- We *update* the state of our program every frame
- We *handle events* that occur while our program is running
- That's the secret to writing interactive software!
- And Processing has already got specific ways of dealing with these things!

setup()

```
void setup() {
} size(640,480);
fill(255,0,0);
stroke(0,255,0);
```

- This is the processing setup() function
- It's where we write what we want to happen at the *start* of our program
- It's also the first time we're *writing a function* ourselves

Stretch Break!

Living Code

DEAD Code

- So far the code we've written just runs once and stops
- Processing carries out our instructions from the top to the bottom and that's that
- But that's not going to lead to especially interesting programming
- And in particular we're not going to be able to make something *interactive* this way
- *So what do we need?*

Three Tips for Interactive Code!

- Generally speaking there are three key elements involved in interactive applications

1. We *set up* the initial conditions of the program
2. We *update* the state of the program over and over to carry out the program's instructions over time
3. We *handle events* like user and other input

Set up

- Before a program really gets going, we often want to set some starting conditions that will help to define how it runs

In a game, for example, we might need to create the player's avatar, put it in front of the door of the dungeon, load the sound effects of the player's footsteps, configure the physics parameters that will be applied during play, etc.

- These are all things we want to do *once* at the *start* of the program

Update

- Once the program is ready, we want it to keep running until it's time to stop
- The way we tend to do this is with an *update loop*, which runs code for making the program work over and over again (once per frame)

Handle events

- While a program is running, we want to be able to react to input either from the person using it, or from other sources
- Sometimes we can do this in the *update loop* and sometimes we have special functions called *event handlers* which run only when a specific kind of event happens (like a mouse click, say)

Set up, update, handle events

- `So one more time`
- We *set up* the conditions for our program
- We *update* the state of our program every frame
- We *handle events* that occur while our program is running
- That's the secret to writing interactive software!
- And Processing has already got specific ways of dealing with these things!

setup()

```
void setup() {
} size(640,480);
fill(255,0,0);
stroke(0,255,0);
```

- This is the processing setup() function
- It's where we write what we want to happen at the *start* of our program
- It's also the first time we're *writing a function* ourselves

`setup ()`

- First we have `void`
- We're going to avoid talking about it for now, ...
- But no, seriously, we'll talk about it in two weeks time!
- For now, just know it needs to be there

`setup ()`

- Next we have `setup`
- This is the name of the function
- In this case, setup is the name of a function Processing already knows about
- But later we'll learn how to make up our own functions, too
- Notice how the name explains what the function is for

setup ()

-
- Next we have empty parentheses, ()
- We know from earlier that this is where the parameters go when we call a function
- When we're writing a function it's where we say what the parameters need to be
- setup() has no parameters, but we still need these empty parentheses here to tell Processing there are no parameters

`setup ()`

- Next we have an opening curly bracket, {
- Now that we've given the name and parameters of our function, this opening curly bracket means "now I'm going to tell you what to do in this function"
- So in this case the curly bracket means "now I'm going to tell you how to set up the program, do this stuff once at the start“
- Finally, we have the closing curly bracket on its own line, }
- This means "I'm done telling you what to do in this function"

draw ()

- void draw () {

  rect (0,0,100,100);

  }
- This is the Processing draw() function
- It's where we write what we want to happen every frame of our program
- As you can see, it's syntactically the same as the setup() function

  In this case the program will draw a rectangle every frame! Exciting!

Class activity: Let’s try this…

```
void setup() {
size(640,480);

stroke(0,255,0);
}
void draw() {
rect(0,0,100,100);
}
```

- So here is an actual Processing program that runs over time
- What will happen?

What was the other thing?

- We now have setup() for getting our program ready to run
- And we have draw() that runs every frame so that our program works over time
- What was that other thing?

What was the other thing?

- We now have setup() for getting our program ready to run
- And we have draw() that runs every frame so that our program works over time
- What was that other thing?
- Oh yeah, we need to handle events so that our program doesn't just ignore us the whole time.

mouseX and mouseY

- One nice way to react to the user is to know where their mouse is in our window
- And Processing gives an easy way to get that location
- The x coordinate of the user's mouse is called mouseX
- The y coordinate of the user's mouse is called mouseY
- These two things are called variables and we'll talk about them next week
- Let's see this in action..

mouseX and mouseY

```
void setup() {
size(640,640);
fill(255,0,0);
stroke(100,0,0);
background(255,255,255);
}
void draw() {
rect(mouseX,mouseY,25,25);
}
```

mouseX and mouseY

```
void setup() {
size(640,640);
fill(255,0,0);
stroke(100,0,0);
background(255,255,255);
}
void draw() {
rect(mouseX,mouseY,25,25);
}
```

mouseX and mouseY

```
void setup() {
size(640,640);
fill(255,0,0);
stroke(100,0,0);
background(255,255,255);
}
void draw() {

rect(mouseX,mouseY,25,25);
}
```

Event handlers

- Processing also has some special functions like setup() and draw() that let us react to user input
- The easiest ones to understand right now are probably the ones to do with pressing buttons on the computer
- Specifically, Processing can tell when the user presses a key on the keyboard or clicks their mouse button

mousePressed ()

```
void setup() {
size(640,480);
}
void draw() {
}
void mousePressed() {
rect(mouseX,mouseY,10,10);
}
```

What do we think this would do?

mousePressed ()

What do we think this would do?

Exactly. When we click the mouse in our window, a little rectangle gets drawn at the click location

mouseReleased()

```
void setup() {
size(640,480);
}
void draw() {
}
void mouseReleased() {
rect(mouseX,mouseY,10,10);
}
```

•Same thing, but now the rectangle is drawn when we *let go* of the mouse button
•Subtle, but sometimes we want to make the distinction

keyPressed()

```
void setup() {
size(640,480);
}
void draw() {
}
void keyPressed() {
rect(mouseX,mouseY,10,10);
}
```

Same thing, but now the rectangle is drawn when we press a key

keyReleased() exists too and does what you might expect!

Just Look Back What we have Learnt So far!

- We've come a long way!

- We can now write a Processing program that
- Sets up the starting conditions
- Runs over and over each frame
- Reacts to user input
- There's more to come, but that's a lot of the heart of writing code!

Class Activity:

- In Processing go to File > Examples and play around with stuff

  (change numbers, go wild!)

Thank You

UOW MALAYSIA KDU

PART OF THE UNIVERSITY OF WOLLONGONG AUSTRALIA GLOBAL NETWORK

Workshop Lab

Introduction to Processing