UNIVERSITY of GREENWICH

Processing Workshop

Ryan Flynn

Overview

- What is Processing?
- Download/install
- Terminology
- Hello World
- `setup()` and `draw()`
- Basic drawing
- Variables
- Review/recap

What is Processing?

- Processing is a coding environment that uses Java to help teach programming skills, but it is also a very powerful tool in its own right for creating interactive products

- Its strengths are in areas such as video, audio and drawing - whilst it can do GUI these are not great and can be difficult to get working how you want

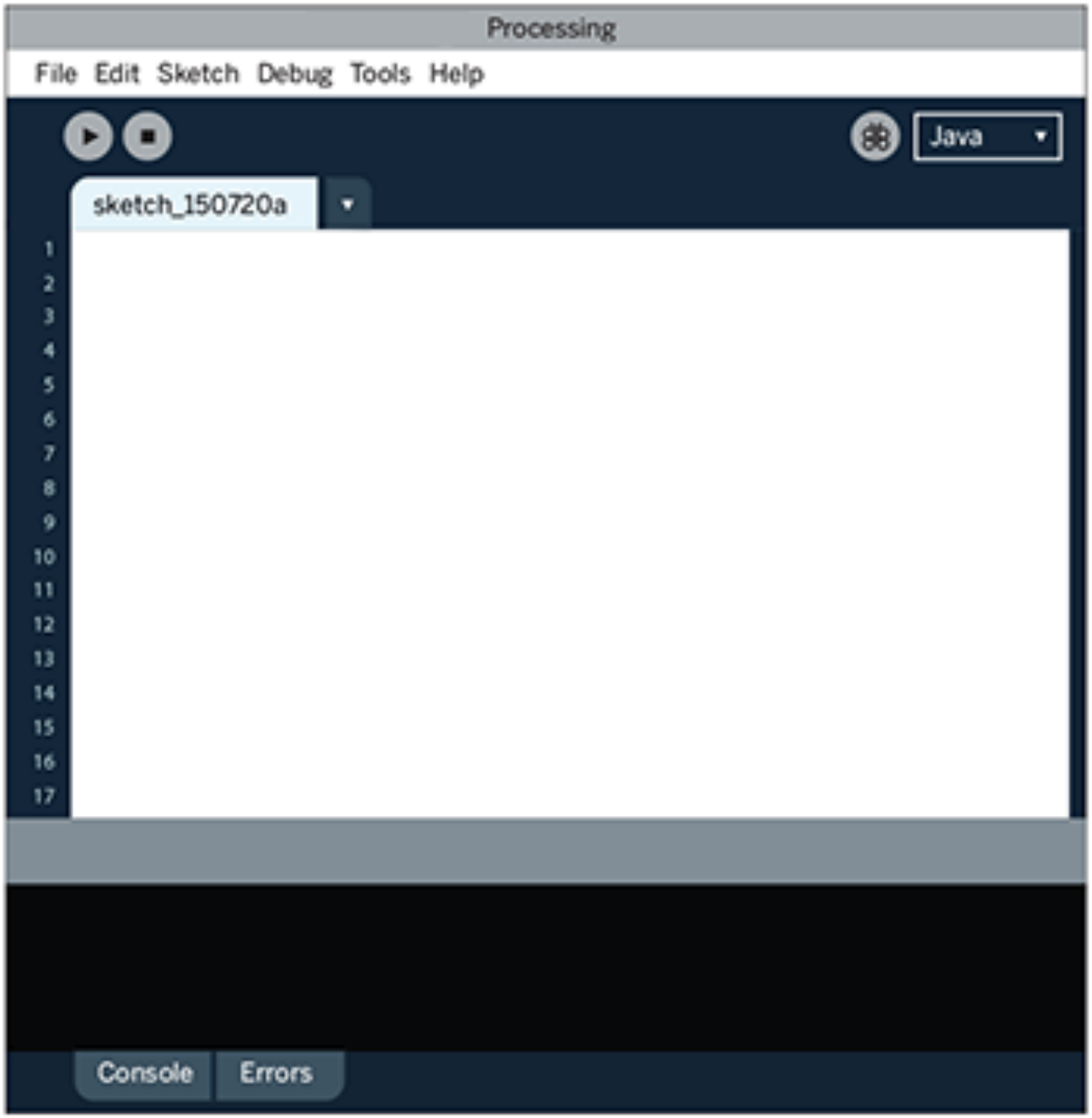
What is Processing? (2)

- As Processing is geared towards creative output it is ideal for making the types of product we are interested in
- It is also extendable and can communicate with other packages using a variety of methods, such as OSC
- Let’s start with a really basic example - Hello World - once we have seen how to install it and the interface

Download/Install

- On our PCs here you should be able to install it using Software Centre
    - Start -> type Software Centre -> search Processing -> install
- On other machines, or if you want to have the newest version, you can download it from https://processing.org/ then extract and run it from the download location
- It *should* be self-contained so you don’t need to download anything else unless you have specific requirements such as an extension or similar you need to install

Display Window



Menu

Toolbar

Tabs

Text Editor

Message Area

Console

Terminology

- In Processing, the file(s) that contain our code is stored in something called a *sketch*
- A sketch has a folder AND a file that ends .pde - both must be present for the sketch to work
    - i.e. if you move a file called mySketch.pde out of the mySketch folder, it will not work

Hello World

- As you may know, a common thing to do with a new programming language is to code something that shows Hello World on the screen

- To do this in Processing we need to understand a couple of things about it's default structure - namely, `setup()` and `draw()`

setup()

- Processing usually needs two default functions to start off a sketch - `setup()` and `draw()`

- `setup()` allows us to run things that we only want to happen once - for instance, setting whether the sketch will run full screen or in a window

draw()

- draw() is a loop. It runs over and over again until the sketch is stopped

- You can have more functions than just setup() and draw() if you want, but almost all Processing sketches have them both

- So let's see an example of how to use them

Open a New Sketch and add the following code…

```
void setup() {

  print(“Hello World”);

}

void draw() {

}
```

Now press Play in Processing

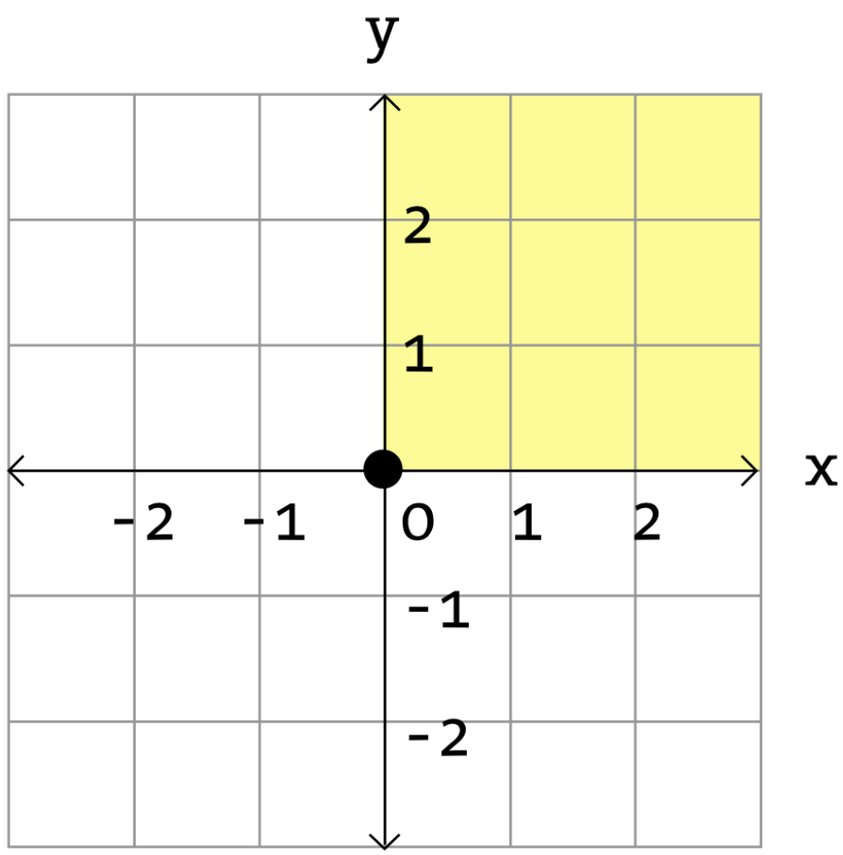
Hello World

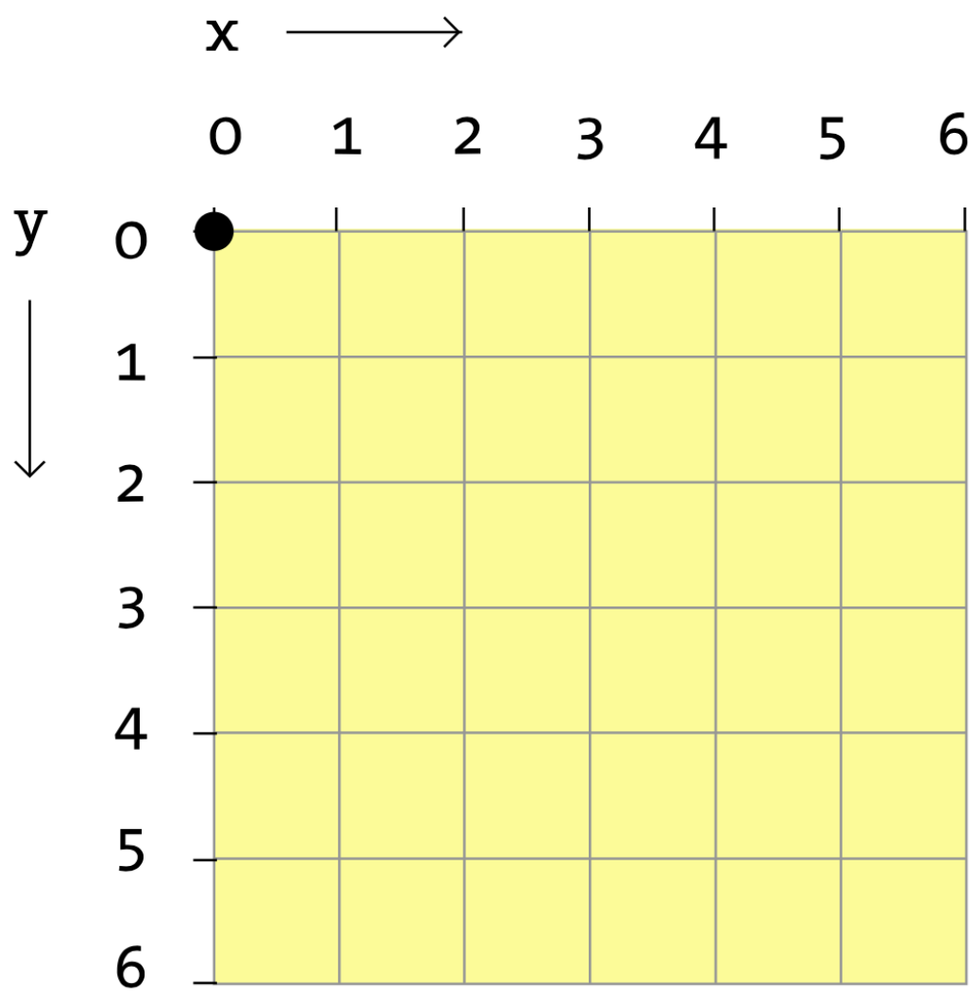
- What do you think would happen if we moved the line `print("Hello World");` to inside the `draw()` function?
  - Try it out…

Basic Drawing

- Processing has four default shapes - point, line, rectangle (`rect()`) and ellipse

- We can draw by viewing the Processing window as a co-ordinate system (or graph), such that the origin (point 0,0) is in the top left corner


y
2
1
-2
-1
0
1
2
-1
-2
x


x
0
1
2
3
4
5
6
y
0
1
2
3
4
5
6

Basic Drawing (2)

- Now we understand how the window works, we can understand how to draw something

- For instance, to add a point we simply use `point(x,y);`

- To draw a line, we use `line(x1, y1, x2, y2);` where x1, y1 refers to the first point and x2, y2 refers to the second

- To draw a rectangle we use `rect(x, y, width, height);` where x, y refers to the top left point of the rectangle and the width and the height correspond to the…well…

- To draw a circle/ellipse we use `ellipse(x, y, width, height);` but here x, y specifies the centre point of the ellipse. Processing fills in the line for us based on the other information

Basic Drawing (3)

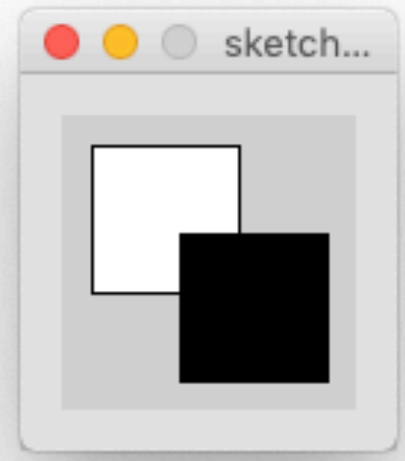
- We can modify the stroke and fill of the shapes we draw using `stroke()` and `fill()`
- Fill takes an argument that relates, by default, to the level of grey in the filled area - 0 is black and 255 is white
- Once you've used one of these commands, it will apply to all subsequent drawing commands. To use a different value, you need to declare it again - example on next slide

sketch_191004a | Processing 3.5.3

Java

sketch_191004a

```
1  void setup() {
2    fill(255);
3    rect(10,10,50,50);
4    fill(0);
5    rect(40,40,50,50);
6  }
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```



Console

Errors

Basic Drawing (4)

- You can also control transparency by adding a second value after the first, e.g. `fill(255, 220);`
- You can also turn off the fill using `noFill();` and the stroke using `noStroke();`

Demo 001

- Let’s see these in action…

Task

- Develop your own stick figure using the tools/techniques you have seen so far

- If it helps, try plotting out the sizes/shapes on paper first

Variables

- As in most programming languages, you need to use variables for some work
- In Processing, you need to declare a variable and its type before using it
  - Otherwise known as being *strongly typed*
- This can catch people out at first, so remember to check your work carefully…

Variables (2)

- For instance, let's say we want to define some colours for use later on in our code

- We need to use the color data type to declare these, so Processing knows what it is we are trying to use

  - e.g. `color beccaPurple = color(102, 51, 153);`

- We can then use that by name, e.g. `background(beccaPurple);`

Variables (3)

- We also need to be aware of variable *scope*
- The scope of a variable tells Processing where it can be used - for instance, if a variable is declared inside `draw()` then it can only be used inside `draw()`
- Again, this can sometimes be confusing when starting out so make sure you check your scope carefully
- If we put a variable outside of any function, it can be called *global* - in other words, it can be used inside any function in the program

Demo 002

- Let’s see these in action…

Variables (4)

- Further work with variables can open up many creative avenues
- For instance, let's imagine that we need something to move around the screen by itself
- We have just seen how to declare a value for the x and y positions of an object, so it stands to reason that we could increment these in order to make an object move…

Demo 003

- Let’s see these in action…

Variables (4a)

- Further work with variables can open up many creative avenues
- We know from previous examples that we can take the x or y value(s) and increment them, but how do we stop the object from going outside the screen? Let’s make a new example to see how this will work…

Open a New Sketch and add the following code…

```
float x = 100, y = 0;
void setup() {
   size(400, 400);
 }
 void draw() {
  background(255);
  fill(0);
  noStroke();
  rectMode(CENTER);
  rect(x, y, 10, 10);
  y++;
}
```

Now press Play in Processing

Variables (5)

- What happened?
- The rectangle moves off the screen, as it doesn’t know that the screen edges are “off limits”
- We can add this to our code by asking Processing to check the current y value for our rectangle and, if it goes beyond a set amount, move it back

Add the following code…

```
float x = 100, y = 0;
void setup() {
    size(400, 400);
  }
  void draw() {
    background(255);
    fill(0);
    noStroke();
    rectMode(CENTER);
    rect(x, y, 10, 10);
    if(y>height) {
      y--;
    } else {
      y++;
    }
}
```

Now press Play in Processing

Variables (6)

- What happened?
- Well, not what we wanted! The rectangle is stuck at the bottom of the screen. Why?
    - Every time the movement occurs when y > height we reverse the direction, but it straight away is back in the same place. So it gets stuck - we need to decouple the movement and the value of y to avoid this
- We can fix this by using a variable to hold our speed, and use that as the movement indicator

Modify your code to read…

```
float x = 100, y = 0, speed = 1.0;
void setup() {
  size(400, 400);
}
void draw() {
  background(255);
  fill(0);
  noStroke();
  rectMode(CENTER);
  rect(x, y, 10, 10);
  if (y>height) {
    speed = speed * -1.0;
  }
    y = y + speed;
}
```

Variables (7)

- What happened?
- That's better. Notice that we removed the else statement so that we are always adding a value to the y direction - but also notice that if the rectangle goes off the bottom of the screen we multiply speed by -1.0
  - Which basically makes it negative
- This approach opens up some cool new paths for us too; let's make our rectangle bounce

Variables (8)

- To make our rectangle bounce, we need to use gravity
- Gravity is a complex physical property, but it can be made “good enough” by simply looking at it as a value that slowly decrements our speed
- So, if we use a value that is less than 1 to multiply our speed, it will lose a little each time it happens
- Let’s see it in action…

Modify your code to read…

```
float x = 100, y = 0, speed = 1.0, gravity = 0.1;
void setup() {
  size(400, 400);
}
void draw() {
  background(255);
  fill(0);
  noStroke();
  rectMode(CENTER);
  rect(x, y, 10, 10);
  y = y + speed;
  speed = speed + gravity;
  if (y>height) {
    speed = speed * -0.95;
  }
}
```

Now press Play in Processing

Variables (9)

- What happens if you change the value for gravity?
  - e.g. try 2.0; try -0.1

Images

- So far, we've seen drawing objects - but what if we wanted to display images?

- One thing that is a little strange about Processing is the way it handles drawing things to the screen

- To understand why this is, we need to see some examples of how to use images and how placement of images can be changed using code in Processing

Images (2)

- The first steps to using images in Processing is:
  - Add an image to the sketch
  - Add a reference to the image using the PImage object

Images (3)

- To add an image to the sketch:
  - Find an image
  - Sketch>Add File…
  - Select the file
  - Note that the file is copied into the sketch directory, inside a new data directory (Processing does this for you!)

Images (4)

- To use PImage:
  - Create a reference to PImage, using the code on the next slide
  - Use the PImage documentation to alter image where necessary: https://processing.org/reference/PImage.html

Demo 004
Add the following code…

```
PImage img;
void setup() {
   size(800, 800);
   img = loadImage(“RF.jpg”);
}
void draw() {
  imageMode(CENTER);
  image(img, width/2, height/2);
}
```

Now press Play in Processing

Images (5)

- Let’s make this interesting. What if we set the x and y position of the image to where the mouse pointer is in our sketch?

- We can get the current x and y value of our mouse by using the commands `mouseX` and `mouseY`

- We simply need to replace the current location values (`width/2, height/2`) with `mouseX` and `mouseY` - see next slide for the updated code

Modify your code to read…

```
PImage img;
void setup() {
   size(800, 800);
   img = loadImage(“RF.jpg”);
}
void draw() {
  imageMode(CENTER);
  image(img, mouseX, mouseY);
}
```

Now press Play in Processing

Images (6)

- If we don’t want the image to leave a trail, we need to “reset” the background each time the screen is refreshed

- Remember, the draw() function updates the screen, so it makes sense to add something there

- We also know that to set the background colour we use the background() command - let’s add the required information into our sketch

Modify your code to read…

```
PImage img;
void setup() {
   size(800, 800);
   img = loadImage(“RF.jpg”);
}
void draw() {
  background(150);
  imageMode(CENTER);
  image(img, mouseX, mouseY);
}
```

Now press Play in Processing

Transformations

- If we want to move objects, or rotate them, or scale them, we call this a transformation

- To understand how transformations work in Processing, you need to have some core concepts under your belt - namely, some basic understanding of coordinates, a basic understanding of angles, and then a slightly more complex consideration of how computers understand how these calculations effect each other

- https://www.youtube.com/watch?v=o9sgjuh-CBM talks through this!

Transformations (2)

- Let’s see how we can use these concepts to create something a bit more interesting and interactive…

Demo 005

- Let’s see these in action…

Image Pixel Array

- When using images, we can also access a pixel level representation of the image - so read individual pixels and do things with them

- We do this by using something called the `pixels[]` array. To do this:

  - Any time we want to access the pixel array, we need to declare `loadPixels()` on the image

  - When we are done, we call `updatePixels()` on the image

Demo 006

- Let’s see these in action…

Image Pixel Array (2)

- Now, a more complex example. Let's loop through all the pixels in the image, one by one, and alter them based on user input

- This involves us using a 2D array - basically, an array within an array - and gets mouse input to change the pixel brightness based on proximity to the mouse location

Demo 007

- Let’s see these in action…

Image Pixel Array (3)

- Of course, you can combine the techniques we have seen so far by using the draw commands to take pixel information and represent it in a different way...

Demo 008

- Let’s see these in action…

Image Pixel Array (4)

- And finally we can extend the image capabilities to also use 3D - it is worth noting that Processing does not really "do" 3D in a manner like, say, Unity or Unreal

- It can push points to 3D space and manipulate them there but more advanced techniques are complex and buggy. It is far easier if you plan to do 3D work to use Unity or Unreal and add on communication via something like OSC to get hardware input; Processing can do it but it requires a lot of coding to get the same type of look you get "out of the box" with Unity/Unreal

Demo 009

- Let’s see these in action…