

Introduction to Computer Graphics

Whakataki ki Whakairoiro Rorohiko

CGRA 151

Prof. Neil Dodgson, Dr Fang-Lue Zhang, Joshua Scott

Fanglue.Zhang@ecs.vuw.ac.nz (course coordinator)

Joshua.Scott@ecs.vuw.ac.nz (teaching fellow)



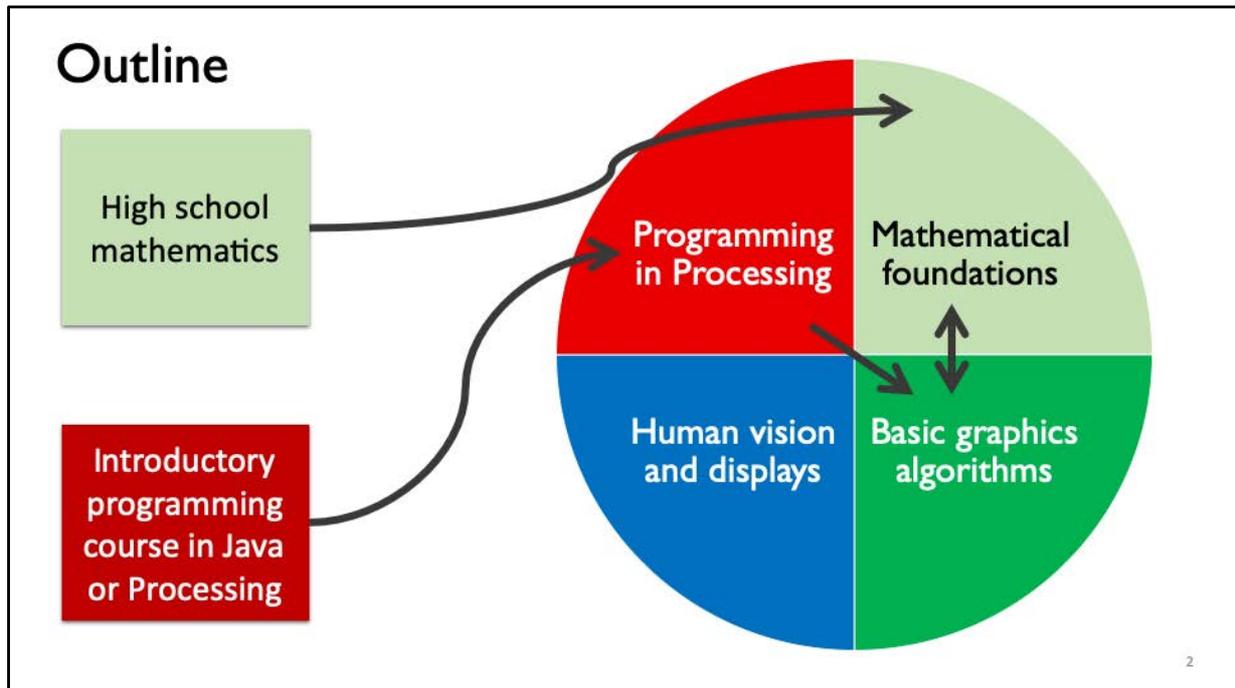
© Disney/Pixar

Welcome to CGRA 151 Introduction to Computer Graphics

The word “graphics” can be interpreted in several ways. Here we mean anything visual drawn by the computer. The course covers practical aspects of making computer graphics along with the background required to understand how a computer generates visual information, how a computer stores visual information and the structures that generate it, and how the human perceptual system works to perceive it.

The Maori “whaka-iro-iro” is the closest translation to “graphics” and adds useful shades of meaning. “whakairoiro” as a noun means simply “carving”. As a verb it is closer to what we are about: “to carve, to ornament with a pattern, decorate”. As a modifier, it conveys “carved, ornamented, ornate, elaborate, decorative”*. This course will teach you much about how we produce ornate, elaborate and decorative computer output.

*Translations taken from the Māori Dictionary
<http://maoridictionary.co.nz/>



We ask that you have done a programming course before taking CGRA151. This can be in Java (COMPI02, COMPI12) or in Processing (DSDN142)

We also ask that you have basic mathematics, either ENGR121, any of the MATH 100-level courses, or 16 NCEA mathematics credits.

We use vectors and matrices in this course. To remind yourself about these, please go check out the Mathematics Workbook on the course [Assignments](#) web page. The workbook is designed to help you revise you all the bits of mathematics that you will need. There is a mathematics assignment due in Week 6 for you to demonstrate your mathematical knowledge. The terms test and the exam will have mathematics questions.

Outline

-  Computer graphics programming in Processing
-  Behind the scenes: fundamental algorithms in computer graphics and the mathematics needed for them
-  Beyond the algorithms: fundamental concepts, including human perception, colour, images, displays



The course has three parts, divided time-wise so that you get the knowledge needed to complete the assignments early on.

Programming: to teach you how to use a Java-like graphics language, Processing, to consolidate what you learnt in COMPI02, COMPI12 or DSDNI42, to teach you something about algorithm design, especially about ways to optimise an algorithm.

Behind the scenes: so that you know what is going on when you tell a computer to draw a line, render a triangle mesh, or run a game.

Fundamental concepts: so that you know the limitations of what we do and why those limitations exist.

Administration

- Course representative
— nominations by Friday*
- “no penalty” withdrawal by 19 July
- Five assignments (35%), Tuesdays at 9a.m.
 - Three programming assignments
 - Mathematics worksheet
 - Project
- One test (10%), ~~15 August~~ **16 August**, in lecture time
- One exam (55%), date to be confirmed

* to Joshua Scott <Joshua.Scott@ecs.vuw.ac.nz>

These dates are correct at the time of printing. They are subject to change.

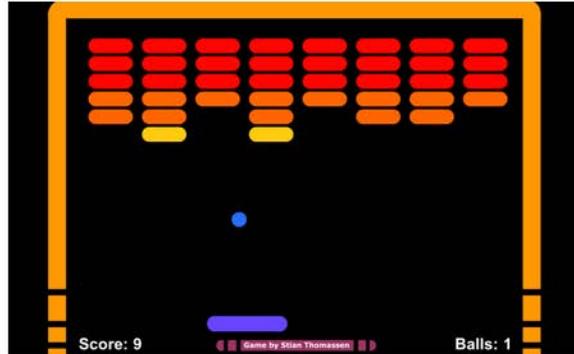
Please check the dates on the course web page:
https://ecs.victoria.ac.nz/Courses/CGRA151_2019T2/

To pass the course you must:

- score at least 50% overall;
- score at least 40% on all five assignments put together;
- score at least 40% on the exam.

The project

- Write either
 - a **2D game** or
 - a **2D interactive art work**
 - Multiple objects
 - Interacting with each other
 - With user interaction
 - Compelling, interesting, challenging, fun
- Mid-trimester break: plan it
- 3 September: submit plan 
- 8 October: submit code 



5

Project management

Plan early.

Get a **basic version working early** and then **repeatedly refine** it into something better.

Always be willing to **reassess, refine, rethink** your plan in response to what goes wrong, what goes right, what feedback you get from others, what other work you need to complete, and what else is going on in your life.

Your timetable for success

Weeks 1–6: Get inspiration, try some bits of experimental coding, think up a plan for what you are going to do.

Mid-trimester break: Write your plan and get some code started.

Week 7: Submit plan.

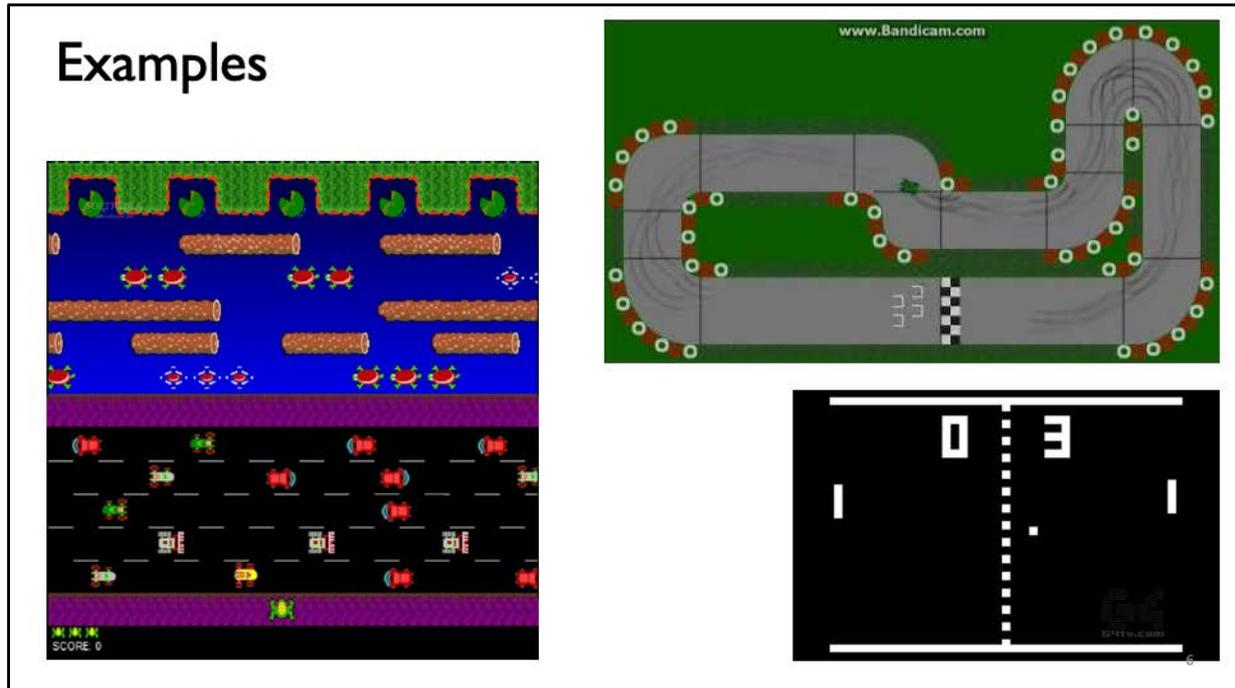
Weeks 7 & 8: Get feedback from a tutor in the planning session.

Week 9: Submit assignment 3 and really get started on your project programming.

Week 10: Have a basic version running. You *could* hand this in if you absolutely had to.

Week 11: Have something working that you would be *happy* to hand in.

Week 12: Have something really good that you *do* hand in.



These are screen shots from:

Frogger

A top-down racing game

Pong

Pong meets the absolute minimum specification for the project but it is too simple to get an A grade mark. Some crazier version of Pong might be interesting.

Laboratories and Lectures

- **Laboratories**
 - CO242 (Stream A)
 - or CO243 (Stream B)
 - Sign up for one session using [myAllocator](#)
 - Attend that session each week
- **Lectures**
 - Tuesday, Thursday, Friday, 9 a.m.
 - MCLT 101
 - Recorded

	Mon	Tue	Wed	Thu	Fri
9		lecture	Red	lecture	lecture
10			Green	Cyan	
11			Blue		
12				Magenta	
13				Yellow	
14			Orange		
15			Violet		

7

Laboratories are named after colours. You attend the same session each week, whether it is for a tutorial, a marking session, a planning session or a help desk. There are no laboratories in Week 6. Stream A has a laboratory (a marking session) in the study week after lectures have finished.

The cyan sessions are timetabled so that Design students (and others who are not normally based on the Kelburn campus) can attend both a Lecture and a Lab in a single two-hour visit to the Kelburn campus.

About the colours

- Red, Orange, Yellow, Green, Blue, Violet are the six main colours that we identify in the spectrum or the rainbow.
- Red, Yellow, Blue are the primary colours used by artists. Orange Green and Purple are the artist's secondary colours, made by mixing two of the primaries. Purple (a mix of Red and Blue) is not quite the same thing as Violet (a colour at the end of the spectrum).
- Red, Green, Blue are the three primary colours used in additive displays (such as LCD panels). Mixing all three primaries produces white in an additive system.
- Yellow, Cyan, Magenta are the three primary colours used in subtractive displays (such as ink jet printers). Mixing all three primaries produces black in a subtractive system.

We will discuss colour in detail in the final part of the course.

Laboratories

- **Tutorial:** working through preparation for an assignment, guided by a tutor
- **Marking:** 10 minutes with a marker
- **Planning:** discussing your plan for your project with a tutor
- **Helpdesk:** a tutor is on hand to help you with your project

Week	Stream A (CO242)	Stream B (CO243)
Week 1	Tutorial 1	Tutorial 1
Week 2	Marking 1	Tutorial 2
Week 3	Tutorial 2	Marking 1
Week 4	Marking 2	Tutorial (maths)
Week 5	Tutorial (maths)	Marking 2
Week 6		
<i>Mid-tri break</i>		
Week 7	Tutorial 3	Planning
Week 8	Planning	Tutorial 3
Week 9	Helpdesk	Marking 3
Week 10	Marking 3	Helpdesk
Week 11	Helpdesk	Helpdesk
Week 12		Marking (project)
<i>Study week</i>	Marking (project)	8

In **Tutorial** sessions you will have a worksheet to work through and tutors available to help you. The worksheets help you prepare yourself to attempt the assignments. You are encouraged to attempt the worksheets before the Tutorial session so that you can bring your problems to the tutors. People who complete the worksheet can work on their assignments during the Tutorial and ask the tutor for help.

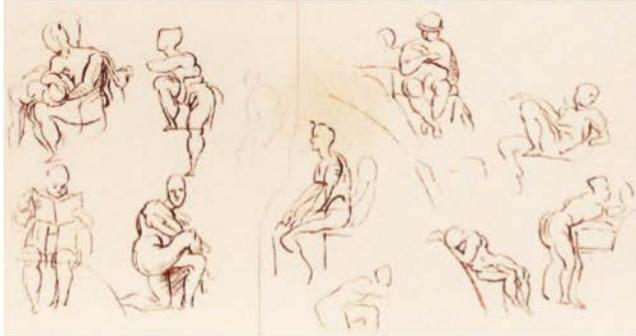
For **Marking** sessions you will have a 10 minute marking slot with a marker. You can spend the rest of the session working towards your next assignment or quietly discussing challenges with your classmates.

The **Planning** session is a 10 minute meeting with a tutor to discuss your plan for your project.

Helpdesk sessions are times where a tutor will be on hand to help with your project.

Why Processing?

- Java-like language
- Quick to get things working
- Sketch-based programming model



9

Java-like: so that you can use the experience you already have from COMPI02 and COMPI12

Quick to get things working: you do not need to remember a load of special steps just to get a graphics window open on the screen. It is easy to get something to show up and easy to get interaction with the user through the mouse.

Sketch-based: the idea is that you make small programs that are “sketches” like rough ideas in an artist’s sketch book. These are later put together (or reworked) as a larger program in Processing.

Processing is on the ECS machines and can be downloaded, free, for your own machine from:

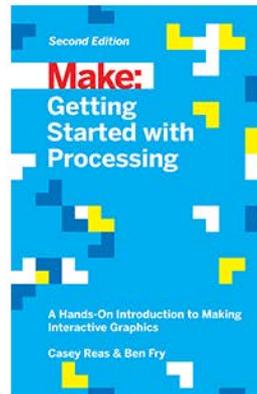
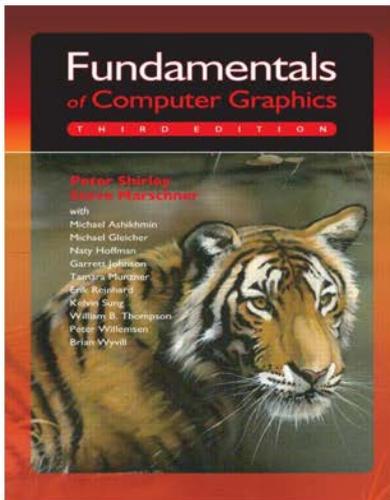
<http://www.processing.org/>

Get Processing running. Go to the Help menu and select Getting Started. Copy and paste the simple drawing example (just below the image of white circles) and see how easy it is to specify a window, interact with the mouse, and draw something simple.

Processing has extensive help available, including a reference manual for every built-in function with example code for each. The reference manual can be accessed from the Help menu in Processing or at this web page:

<http://www.processing.org/reference/>

Course texts



Available as eBooks through Talis (Reading List link on the course web site)

They support the course rather than define the course: read them to get an alternative view

10

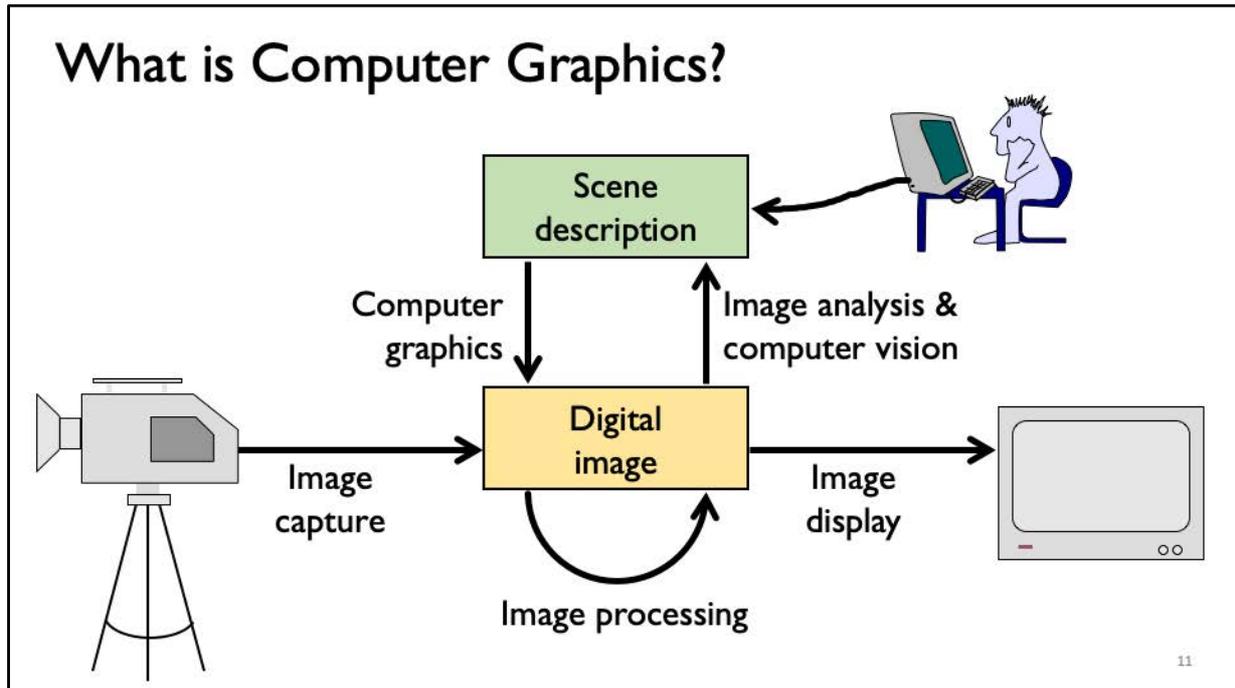
Both books are provided by Victoria University of Wellington to all students on the course as eBooks, which can be read online.

You can access them through the University's Talis system (the [Reading List](#) link on the course website).

In these notes I refer to them as “*Fundamentals of Computer Graphics*” and “*Getting Started with Processing*”.

For information on the philosophy behind Processing, and what Processing can do for you, see *Getting Started with Processing*, Chapter 1 (pages 1–5).

For an opinion on the very wide range of things that could be considered within the remit of “computer graphics” see *Fundamentals of Computer Graphics*, Sections 1.1 and 1.2 (pages 1–4)



Computer graphics is technically the process of taking a **scene description**, that is some way of describing a set of graphical objects, and converting this into a **digital image**. However, “computer graphics” can be taken much more broadly to include any of the processes shown in the diagram.

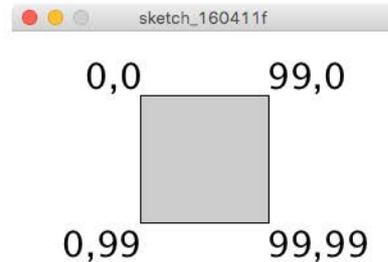
How widely used is computer graphics?

- It is everywhere!
- All visual computer output
 - on screen
 - printed
- Games
- Visual effects
- Post-production
- Books, magazines, newspapers



Coordinate system

- x-axis runs left-to-right
- y-axis runs top-to-bottom
- point (0,0) is at top left
- Processing's default window size is 100x100



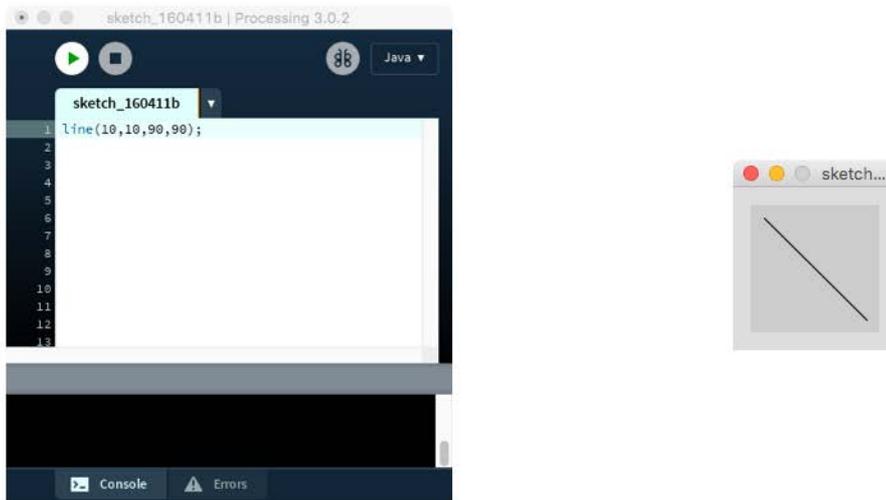
13

The default window is 100x100 pixels.

The top left pixel is always (0,0), no matter what the window size is.

Notice that the pixel numbering starts at zero. This means that, in a 100x100 pixel window, the bottom right pixel will be (99,99).

Getting started with Processing: line()



`line(10,10,90,90)` — draw a line from point (10,10) to point (90,90)

`line` is a **function** name

10,10,90,90 are its four **parameters**

`line(x1,y1,x2,y2)` — draw a line from point (x1,y1) to point (x2,y2)

The textbook, *Getting Started with Processing*, Chapter 2 (pages 7–12) gives more information on writing your first Processing program, including:

- where to download the software for your own computer (www.processing.org)
- how to write a one-line program (they use an ellipse rather than a line)
- some simple mouse interaction (we'll be doing that in a few slides' time)
- how to save your code (there's the usual Save option on the usual File menu)
- where to go to find example code and a reference manual (www.processing.org)

The x-axis runs horizontally left-to-right



15

Here we see three lines:

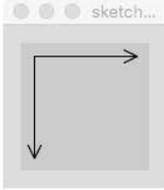
- a long line running from (10,10) to (90,10). This long line is horizontal: the second coordinates, known as the y-coordinates, of the two points are the same.
- two short lines running from (90,10) to either (80,5) or (80,15). These two lines start and end at the same x-coordinate but the y-coordinate is different for the second point of each, so one is above the long line and the other is below the long line.

The symbol “//” introduces a comment. Processing ignores anything that is written between the “//” and the end of the line.

Comments are important for making code understandable to a human: while the code may be clear to *you now*, it will not be clear to *someone else* and it will not be clear to *you in six months' time*.

The y-axis runs vertically top-to-bottom

```
sketch_160411b
1
2 // a horizontal line
3 line(10,10,90,10);
4 // two lines making an arrowhead
5 line(90,10,80,5);
6 line(90,10,80,15);
7
8 // a vertical line
9 line(10,10,10,90);
10 // two lines making an arrowhead
11 line(10,90,5,80);
12 line(10,90,15,80);
```

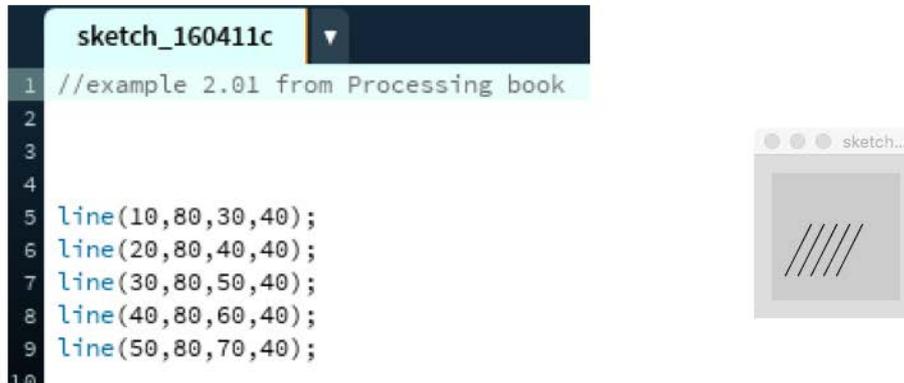


16

To make the vertical arrow we have switched the x and y coordinates of all the points.

Notice that the y-axis goes down. This is consistent with Victoria University of Wellington's Java graphics environment (used in COMPI02) but it is upside down compared to what is normally used in mathematics, where the y-axis goes up. You just have to get used to it.

Five lines



```
sketch_160411c ▾  
1 //example 2.01 from Processing book  
2  
3  
4  
5 line(10,80,30,40);  
6 line(20,80,40,40);  
7 line(30,80,50,40);  
8 line(40,80,60,40);  
9 line(50,80,70,40);  
10
```

The screenshot shows a Processing IDE window titled 'sketch_160411c'. The code editor contains five lines of code, each starting with 'line' and followed by four numerical arguments. The first argument increases by 10 units from 10 to 50, while the second, third, and fourth arguments remain constant at 80, 40, and 40 respectively. To the right of the code editor is a preview window titled 'sketch...' which displays a grey square canvas containing five parallel, slanted lines. The lines are positioned at regular intervals along the x-axis, all starting at the same vertical level (y=80) and ending at the same vertical level (y=40).

17

Draw 5 lines, all with the same y-coordinate (80) at their start point and the same y-coordinate (40) at their end point.

The textbook, *Getting Started with Processing*, Chapter 3 (pages 13–31) takes you through all the material that we are about to cover, including drawing lines, rectangles, ellipses, triangles, polygons, in colour, in greyscale. It will provide you with an alternative introduction to that provided in these slides.

The “Processing book” referenced on the slide itself is yet another introduction to Processing: *Processing: a programming handbook visual designers and artists*, Casey Reas and Ben Fry, MIT Press, Second Edition, 2014. We do not have an eBook version of that, which is why we’re using some of the examples from that book in the lecture slides.

background(), stroke(), strokeWeight()

```
sketch_160411c ▾
1 //example 2.02 from Processing book
2 background(0);
3 stroke(255);
4 strokeWeight(5);
5 line(10,80,30,40);
6 line(20,80,40,40);
7 line(30,80,50,40);
8 line(40,80,60,40);
9 line(50,80,70,40);
10
```



18

Introducing some simple functions:

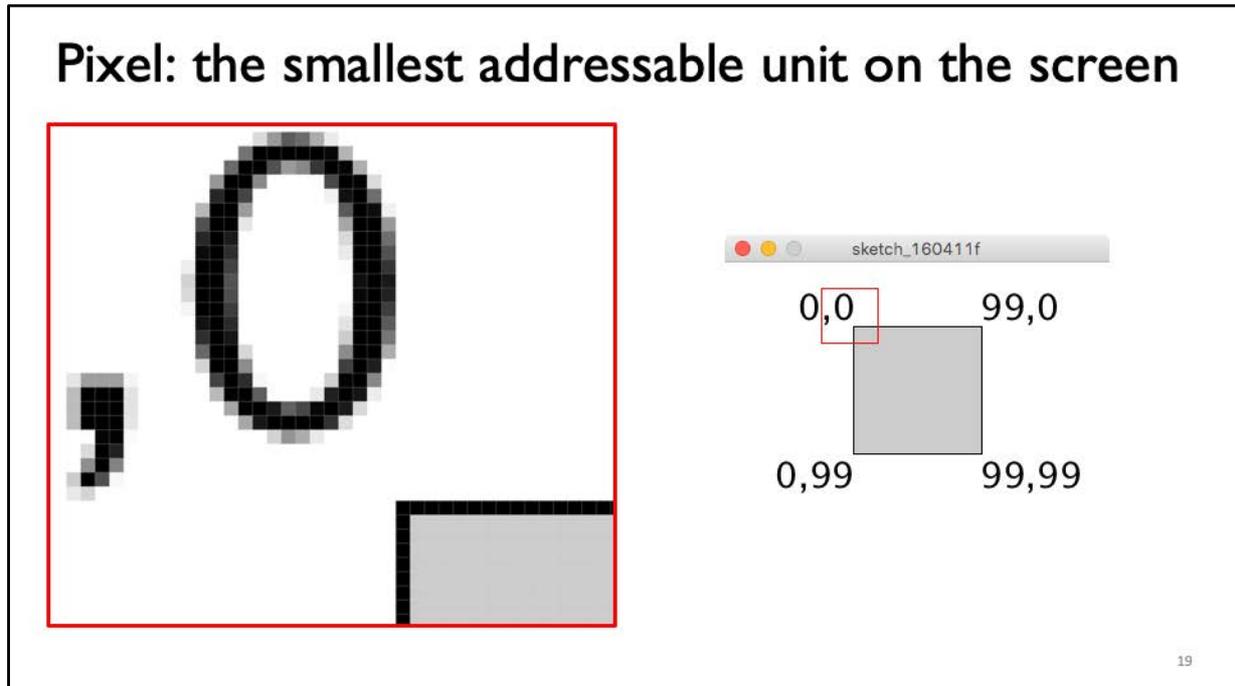
`background(0)` — set the background colour to be greyscale value 0, which is black, and fill the window with that colour

`stroke(255)` — set the colour for drawing **strokes** (lines and edges) to be greyscale value 255, which is white

`strokeWeight(5)` — set the **weight** of strokes to be 5. The weight is the width of the drawn line and it is measured in **pixels**

A note on notation: at the top of the slide is written “background(), stroke(), strokeWeight()” with the parenthesis “()” but without the parameters inside the parenthesis. This is a shorthand to tell you that background, stroke and strokeWeight are the names of **functions**.

In the Processing code window, the names of functions are written in blue, automatically, so that you know that they are system-defined functions. We will later see how Processing uses other colours to identify system-defined variables and commands.



The image on the right was drawn using Processing, showing a 100×100 gray rectangle in a Processing window of size 300×200 .

The black boundary is exactly one pixel wide and is outside the Processing 100×100 drawing frame. The number “0” is drawn as 15 pixels wide and 22 pixels high. On my desktop monitor it is 5mm high, which means my desktop monitor has about 4.4 pixels per millimeter.

Monitor resolutions are measured in **pixels per inch (ppi)**. One inch is 25.4 millimeters. So my desktop monitor has a resolution of about 110 pixels per inch (ppi).

My monitor is marketed as a 27 inch display with 2560×1440 pixels. The 27 inches is measured diagonally from corner to corner. Pixels are usually as tall as they are wide, so the 2560×1440 means that my monitor has an aspect ratio of 16:9. That is, its width is $16/9$ times its height. A 27 inch display therefore can be calculated as having a width of 23.53 inches and a height of 13.24 inches. Thus its resolution is 108.8 pixels per inch.

For many years, monitors had about 100 ppi, just like my current desktop monitor. This is sufficiently coarse that you can see individual pixels if you look closely. Higher resolutions became available from 2010, starting with the iPhone 4. Pixel densities are now regularly over 300 ppi. At that fine resolution, it is hard or impossible for a human eye to see the individual pixels. There is more on this later in the course.

Addressing individual pixels: point()

```
sketch_160411g
1 background(255); // white background
2 stroke(0);      // draw in black
3 point(10,10);  // four points
4 point(90,10);
5 point(10,90);
6 point(90,90);
7
8 point(0,0);    // top left
9 point(1,1);
10 point(98,98);
11 point(99,99); // bottom right
12 point(100,100); // outside the drawing area
```



20

Notice that the bottom right hand pixel is (99,99). The function call `point(100,100)` tries to draw a pixel outside the drawing area, so nothing is drawn.

Later in the course we consider how the graphics system of a computer ensures that each application is only drawing inside its own window, rather than scribbling all over the screen.

What is an image?

- two dimensional function
- value at any point is an intensity or colour
- not digital!

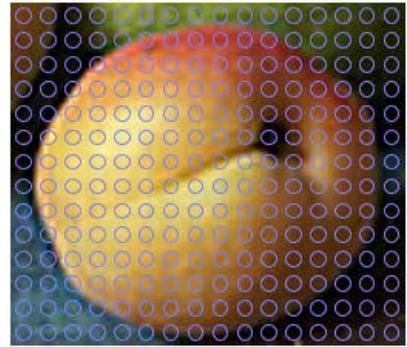
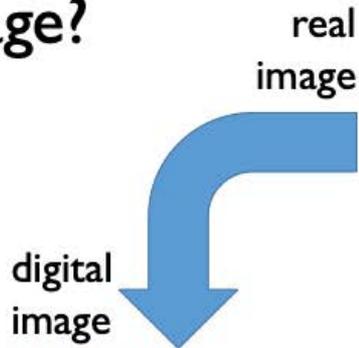


The image that you see on the screen is not a digital image. It is a digital image that has been converted to analogue (physical) form by the display device, whether that device is a computer screen, a projector or a printer.

See *Fundamentals of Computer Graphics*, Section 3.2 (pages 59–64). Also useful to read through the earlier part of Chapter 3.

What is a *digital* image?

- a contradiction in terms
 - if you can see it, it's not digital
 - if it's digital, it's just a collection of numbers
- a **sampled and quantised** version of a real image
- a rectangular array of **intensity or colour** values



```

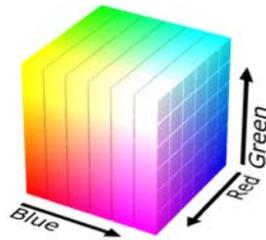
69 14 80 56 12 34 30 1 78 79 21 145 156 52 136 143 65 115 129 41 128 143 50 85
106 11 74 96 14 85 87 23 66 74 23 73 82 29 67 76 21 40 48 7 33 39 9 84 54 19
42 27 6 19 10 3 59 60 28 102 107 41 208 88 63 204 75 54 197 82 63 179 63 46 158 62
46 146 49 40 52 65 21 60 68 11 40 51 17 35 37 0 28 29 0 83 50 15 2 0 1 13 14
8 243 173 161 231 140 69 239 142 89 230 143 90 210 126 79 184 88 48 152 69 35 123 51
27 104 41 23 55 45 9 36 27 0 28 26 2 29 28 7 40 28 16 13 13 1 224 167 112 240
174 80 227 174 78 227 176 87 233 177 94 213 149 78 196 123 57 141 72 31 108 53 22 121
62 22 126 50 24 101 49 35 16 21 1 12 5 0 14 16 11 3 0 0 237 176 83 244 206 123
241 236 144 238 222 147 221 190 108 215 170 77 190 135 52 136 93 38 76 35 7 113 56 26
156 83 38 107 52 21 31 14 7 9 6 0 20 14 12 255 214 112 242 215 108 246 227 133 239
232 152 229 209 123 232 193 98 208 162 64 179 133 47 142 90 32 29 19 27 89 53 21 171
116 49 114 64 29 75 49 24 10 9 5 11 16 9 237 190 82 249 221 122 241 225 129 240 219
126 240 199 93 218 173 69 198 135 33 219 186 79 189 184 93 136 104 65 112 69 37 191 153
80 122 74 28 80 51 19 19 37 47 16 37 32 223 177 83 235 208 105 243 218 125 238 206
103 221 188 83 228 204 98 224 220 123 210 194 109 192 159 62 150 96 40 116 73 28 146 104
46 109 59 24 75 48 18 27 33 33 47 100 118 216 177 98 223 189 91 239 209 111 236 213
117 217 200 108 218 200 100 218 206 104 207 175 76 177 131 54 142 88 41 108 65 22 103
59 22 93 53 18 76 50 17 9 10 2 54 76 74 108 111 102 218 194 108 228 203 102 228 200
100 212 180 79 220 182 85 198 158 62 180 138 54 155 106 37 132 82 33 95 51 14 87 48
15 81 46 14 16 15 0 11 6 0 64 90 91 54 80 93 220 186 97 212 190 105 214 177 86 208
165 71 196 150 64 175 127 42 170 117 49 139 89 30 102 53 12 84 43 13 79 46 15 72 42
14 10 13 4 12 8 0 69 104 110 58 96 109 130 128 115 196 154 82 196 148 66 183 138 70
174 125 56 169 120 54 146 97 41 118 67 24 90 52 16 75 46 16 58 42 19 13 7 9 10 5
0 18 11 3 66 111 116 70 100 102 78 103 99 57 71 82 162 111 66 141 96 37 152 102 51
130 80 31 110 63 21 83 44 11 69 42 12 28 8 0 7 5 10 18 4 0 17 10 2 30 20 10
58 88 96 53 88 94 59 91 102 69 99 110 54 80 79 23 69 85 31 34 25 53 41 25 21 2
0 8 0 0 17 10 4 11 0 0 34 21 13 47 35 23 38 26 14 35 24
    
```

intensity is also known as **greyscale** or, in the world of film and photography, as **black & white**

colour is usually specified by three numbers. Here we are using RGB, but there are many other useful three-dimensional colour spaces. Why three dimensions? That is explained later in the course.

RGB Colour

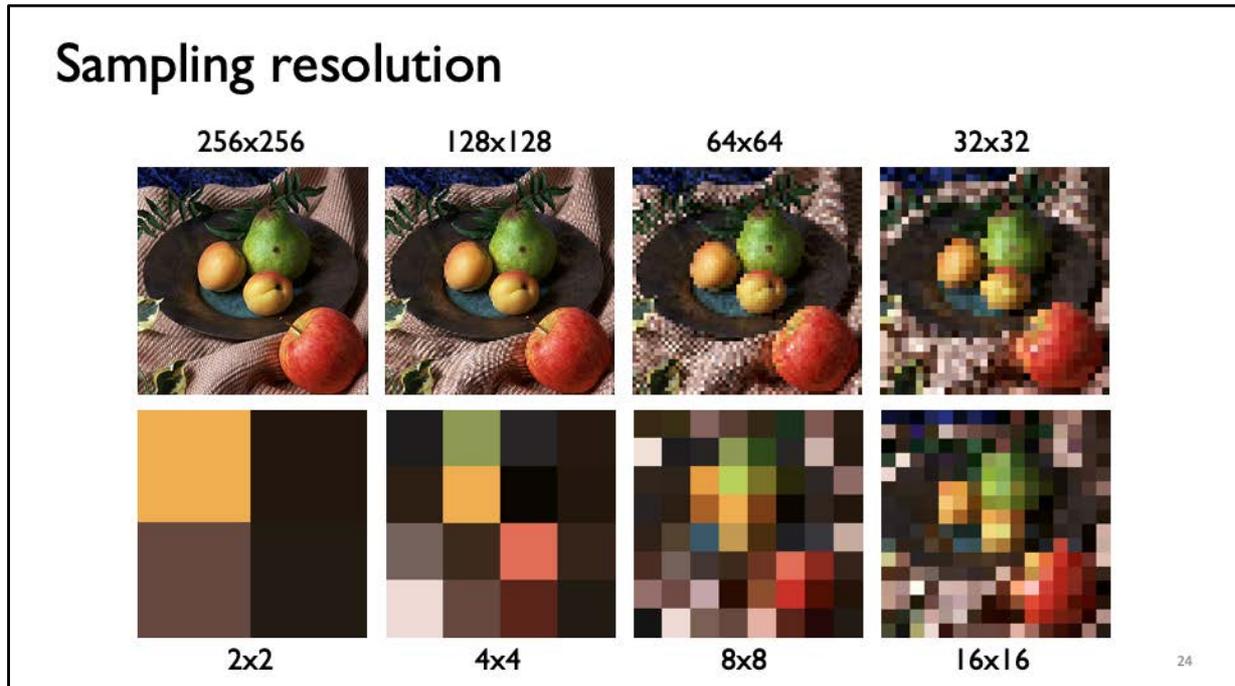
- In computer, we need to represent colour by some set of numbers
 - preferably a small set of numbers which can be quantised to a fairly small number of bits each
- We use three numbers for each point to represent the different intensity of primary colour light
 - Red
 - Green
 - Blue



A representation of additive color mixing. Projection of [primary color](#) lights on a white screen

23

A representation of additive color mixing. Projection of [primary color](#) lights on a white screen shows secondary colors where two overlap; the combination of all three of red, green and blue in equal intensities makes white.



a digital image is a rectangular array of intensity values

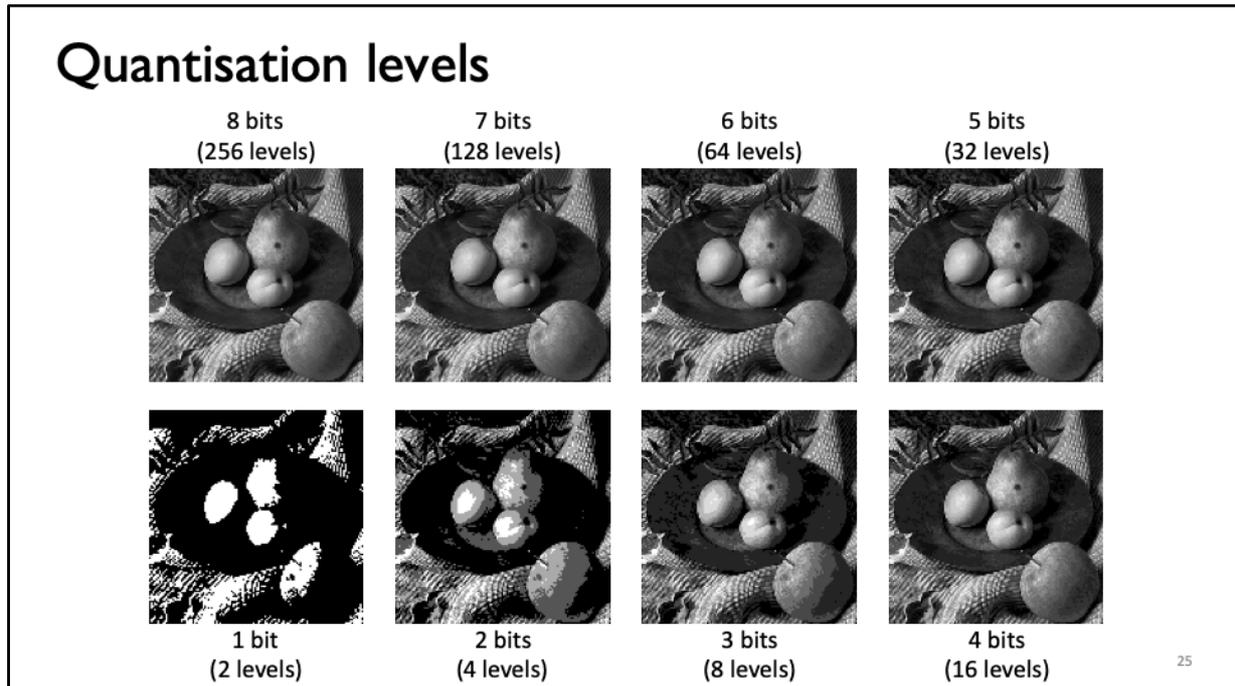
each value is called a pixel (“picture element”)

sampling resolution is normally measured in pixels per inch (ppi) or, equivalently, dots per inch (dpi)

displays have a resolution between 100 and 400 dpi

printers have resolutions of between 600 and 3000 dpi

but printers display only two levels (ink or no ink) while displays show many many gradations of colour



each intensity value is a number

for digital storage the intensity values must be quantised
 limits the number of different intensities that can be stored
 limits the brightest intensity that can be stored

how many intensity levels are needed for human consumption?
 8 bits often sufficient
 some applications use 10 or 12 or 16 bits
 more detail later in the course

colour is stored as a set of numbers
 usually as 3 numbers of 5–16 bits each
 more detail later in the course

printers can generally only show a mark or no mark in each pixel (one bit per pixel)
 how we get (apparent) gradations of grey and how we get (apparent) colour are explained later in the course

Colour

- `background()`, `stroke()`,
`fill()` all set colour values
for drawing
- one number:
a grey value
in the range 0–255
0 = black
255 = white
- examples:
`fill(255);`
`background(200);`
`stroke(0);`
`stroke(125);`

	0
	25
	50
	75
	100
	125
	150
	175
	200
	225
	250

26

The sample greys at right were generated by a simple Processing program which filled each box with the appropriate grey value and wrote the corresponding number next to it. The maximum grey value is 255. The numbers in the examples go from 0 to 250 in steps of 25. The background of the window is white, which is colour 255. You can just tell that 250 is slightly darker than pure white.

See *Getting Started with Processing*, pages 23–27.

Colour

- `background()`, `stroke()`, `fill()`
all set colour values for drawing
- three numbers:
a colour
red, green, blue
each in the range 0–255
- examples:
`background(100,50,0);`
`fill(0,250,0);`
`stroke(250,100,50);`

0,0,0	50,0,0	100,0,0	150,0,0	200,0,0	250,0,0
0,50,0	50,50,0	100,50,0	150,50,0	200,50,0	250,50,0
0,100,0	50,100,0	100,100,0	150,100,0	200,100,0	250,100,0
0,150,0	50,150,0	100,150,0	150,150,0	200,150,0	250,150,0
0,200,0	50,200,0	100,200,0	150,200,0	200,200,0	250,200,0
0,250,0	50,250,0	100,250,0	150,250,0	200,250,0	250,250,0
0,0,0	50,50,0	100,100,0	150,150,0	200,200,0	250,250,0
0,0,50	50,50,50	100,100,50	150,150,50	200,200,50	250,250,50
0,0,100	50,50,100	100,100,100	150,150,100	200,200,100	250,250,100
0,0,150	50,50,150	100,100,150	150,150,150	200,200,150	250,250,150
0,0,200	50,50,200	100,100,200	150,150,200	200,200,200	250,250,200
0,0,250	50,50,250	100,100,250	150,150,250	200,200,250	250,250,250

Notice that the `color()` function is allowed to take either one parameter (a grey value) or three parameters (colour represented by red, green and blue components). Processing has several functions that have different versions that depend on the number of parameters. This means you need to read carefully the function definitions in the Processing reference manual.

The reference manual is available from inside Processing, from the Help menu choose “Reference”. Go have a look at it. In particular look at the definition of `background()`, which is the first item in the third column. Notice that there are several more possibilities for number and type of parameter.

The (red, green, blue) triple used to represent colour is usually abbreviated RGB.

Want an easy way to select colours and get their RGB values: try a utility like <http://www.colorpicker.com>

See *Fundamentals of Computer Graphics*, Section 3.3 (pages 64–65).

We discuss representations of colour in detail later in the course.

Colour in action

```
sketch_160411c ▼  
1 background(0);  
2 strokeWeight(5);  
3 stroke(255,0,0); // a red line  
4 line(10,80,30,40);  
5 stroke(255,255,0); // a yellow line  
6 line(20,80,40,40);  
7 stroke(0,255,0); // a green line  
8 line(30,80,50,40);  
9 stroke(0,0,255); // a blue line  
10 line(40,80,60,40);  
11 stroke(255,0,255); // a magenta line  
12 line(50,80,70,40);
```



28

Some things to note about these RGB colours:

0,255,0 is a very bright green, brighter than you might expect compared to red and yellow.

0,0,255 is vivid but dark blue, though it is clearly blue it's quite hard to distinguish from the black background.

255,0,255 is “magenta”, a shade of purplish-pink. Purples and pinks are colours that are not found in the rainbow. Magenta (255,0,255) is one of the standard inks used in colour printers along with cyan (0,255,255), yellow (255,255,0) and black (0,0,0).

The reasons for all of these observations are explained later in the course, when we discuss human perception and colour representations.

The processing loop: setup() and draw()

```

RotatingLine
1 float angle = 0.0 ;
2
3 void setup(){
4   size(400,400);
5 }
6
7 void draw(){
8   float c, s ;
9   background(255) ;
10  c = cos(angle) ;
11  s = sin(angle) ;
12  strokeWeight(5) ;
13  line( c*50+200, s*50+200, c*150+200, s*150+200 ) ;
14  angle = angle + 0.02 ;
15 }

```

- **setup()**
 - run once when the program starts
 - usually includes `size()`
- **draw()**
 - run repeatedly
 - once every fraction of a second
 - screen is updated at end of `draw()`
- **frameRate()**
 - tells Processing how often to call `draw()`

29

This is for a live demo. The example code is in file `RotatingLine`.

For the live demo, we first see what `RotatingLine` does. Then show how `frameRate()` affects what can be seen. It is instructive to change the angle increment to 0.2 for slow frame rates.

You need to be happy with the cosine and sine functions later in the course. If you are a bit hazy on them, you should refresh your memory before we meet them again. See *Getting Started with Processing*, pages 113–117, for a nice introduction and *Fundamentals of Computer Graphics*, Section 2.3 (pages 18–20) for a summary of facts.

There are three variables in this program, one *global* variable (`angle`) that is available from anywhere in the program, and two *local* variables (`c` and `s`) that are available only in the `draw()` function. If you are a bit hazy on variables, then you need to read *Getting Started with Processing*, Chapter 4, pages 35–40, which is a quick tutorial on defining and using variables.

For more on the `setup()/draw()` approach see *Getting Started with Processing*, Chapter 5, pages 49–53, which gives a worked example for you to copy and learn from.

Now move to a simpler demo, drawing a line...

A simple animation

```

StrobingLine
1 int y=0;
2
3 void setup(){
4   size(200,600);
5   frameRate(60);
6 }
7
8 void draw(){
9   background(255);
10  strokeWeight(5);
11  line( 0, y%height, width, y%height );
12  y=y+1;
13 }

```

- **height** and **width** are pre-defined variables
- **int** declares an integer variable
- **void** means that a function returns no value
- **%** is an operator that returns the remainder after division

30

We can do this as a simple example, building up the various features to the final version shown above. The most basic, but boring, version of the draw function is just:

```

void draw(){
  line( 0, 50, 200, 50 );
}

```

Add to this, one at a time, in the following order and see how each changes the output:

background()

strokeWeight()

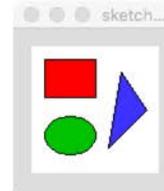
width

parameter *y*

the mod operator: *y%height*

Some more basic shapes: `rect()`, `ellipse()`, `triangle()`

```
sketch_160412b
1 background(255); // white background
2 stroke(0);      // black outlines
3 noSmooth();    // do not smooth outlines
4
5 fill(255,0,0); // red rectangle
6 rect(10,10,40,30);
7
8 fill(0,192,0); // green ellipse
9 ellipse(30,70,40,30);
10
11 fill(64,64,255); // blue triangle
12 triangle(70,20,60,80,90,50);
```



31

You will meet all these in assignment I.

If you have not already, have a look at *Getting Started with Processing*, Chapter 3 (pages 13–31).

How would you specify a rectangle?

- a rectangle needs four parameters
- default is `rect(left,top,width,height);`

The diagram shows four rectangles, each with a different specification method:

- rectMode(CORNER):** A rectangle with a dot at the top-left corner. The text 'left,top' is next to the dot. A horizontal double-headed arrow below the rectangle is labeled 'width', and a vertical double-headed arrow to its right is labeled 'height'.
- rectMode(CORNERS):** A rectangle with a dot at the top-left corner and another dot at the bottom-right corner. The text 'left,top' is next to the top-left dot, and 'right,bottom' is next to the bottom-right dot.
- rectMode(CENTER):** A rectangle with a dot at its center. The text 'centreX,centreY' is next to the dot. A horizontal double-headed arrow below the rectangle is labeled 'width', and a vertical double-headed arrow to its right is labeled 'height'.
- rectMode(RADIUS):** A rectangle with a dot at its center. The text 'centreX,centreY' is next to the dot. A horizontal double-headed arrow below the rectangle is labeled 'radiusX', and a vertical double-headed arrow to its right is labeled 'radiusY'.

This is partly “behind the scenes”, because we need to think about *why* Processing has four different ways to specify rectangles.

An axis-aligned rectangle (that’s a rectangle with edges parallel to the horizontal and vertical axes, rather than one that is rotated) can be specified by exactly four numbers. You cannot specify one with fewer than four parameters and you do not need more than four.

The different ways of thinking about how to place a rectangle reflect that different people really do think about these things in different ways. They also reflect that some methods are more useful in some circumstances than others.

For example, when drawing a rectangle using the mouse, you likely think about the top-right approach. You click the mouse button, which specifies one corner, then drag and release the mouse button at the other corner.

There is at least one more way you could *sensibly* specify a rectangle with four numbers. Think about it.

You can probably understand why someone might want to use the top two methods and maybe the bottom left one, but why the bottom right one? It is because of how we think about ellipses.

How would you specify an ellipse?

- an ellipse needs four parameters

```
ellipse( a, b, c, d );
```

- default is `ellipse(centreX,centreY, width,height);`

The diagram illustrates four modes of ellipse specification:

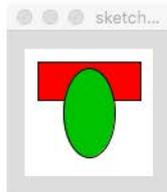
- ellipseMode(CORNER)**: Shows an ellipse with a dot at the top-left corner. A horizontal double-headed arrow below the ellipse is labeled "width", and a vertical double-headed arrow to its right is labeled "height".
- ellipseMode(CORNERS)**: Shows an ellipse with a dot at the top-left corner and another dot at the bottom-right corner. A horizontal double-headed arrow below the ellipse is labeled "width", and a vertical double-headed arrow to its right is labeled "height".
- ellipseMode(CENTER)**: Shows an ellipse with a dot at its center. A horizontal double-headed arrow below the ellipse is labeled "width", and a vertical double-headed arrow to its right is labeled "height".
- ellipseMode(RADIUS)**: Shows an ellipse with a dot at its center. A horizontal double-headed arrow below the ellipse is labeled "radiusX", and a vertical double-headed arrow to its right is labeled "radiusY".

Ellipses have exactly the same four modes of specification.

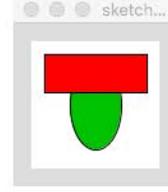
For an ellipse the bottom two methods make more sense than the top two.

Order is important

```
background(255);    // white
fill(255,0,0);     // red
rect(10,10,80,30); // rectangle
fill(0,192,0);    // green
ellipse(50,50,40,70); // ellipse
```



```
background(255);    // white
fill(0,192,0);     // green
ellipse(50,50,40,70); // ellipse
fill(255,0,0);    // red
rect(10,10,80,30); // rectangle
```



34

Processing draws using the “painter’s algorithm”. Whatever is drawn *later* is drawn *on top* of what was drawn earlier.

Simple interaction

```
int a, b, c, d ;
```

```
void setup(){
  size(500,500) ;
}
```

```
void draw(){
  background(255) ;
  rectMode( CORNERS ) ;
  rect( a, b, c, d ) ;
}
```

```
void mousePressed(){
  a = mouseX ;
  b = mouseY ;
  c = a ;
  d = b ;
}
```

```
void mouseDragged(){
  c = mouseX ;
  d = mouseY ;
}
```

- `mousePressed()` is called whenever a mouse button is pressed down
- `mouseDragged()` is called whenever a mouse button is down and the mouse has moved
- `mouseX` and `mouseY` are pre-defined variables that give you the location of the mouse
- what do you need to change to get this to work correctly for the other three types of `rectMode()` ?

35

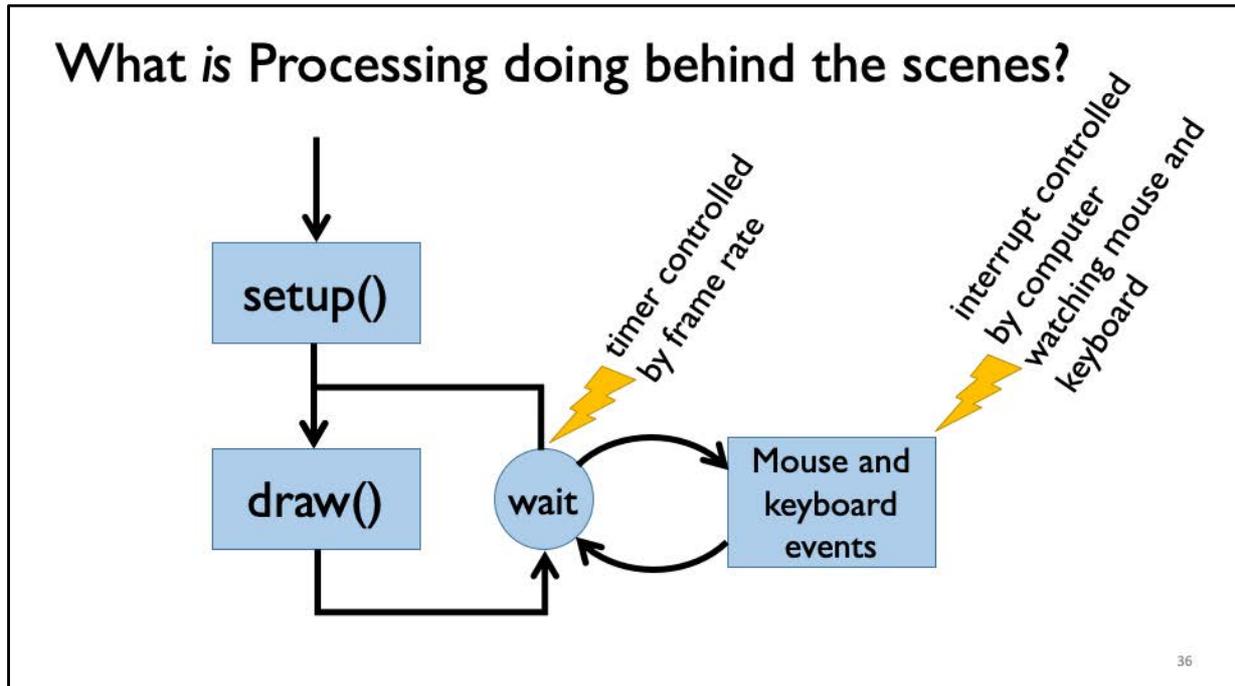
This is example code `DrawSimpleRectangle`

When the mouse is pressed, Processing calls the `mousePressed()` function which, in this case, sets both corners of the rectangle to be at the current mouse location.

So long as the mouse button is pressed down, Processing calls the `mouseDragged()` function whenever the mouse moves from its current location. In the example, this sets the second corner of the rectangle (c,d) to the new mouse position. The other corner (a,b) stays where it is because `mouseDragged()` does not update it.

You might like to switch to drawing ellipses to see how that feels; then go back to rectangles.

Homework: Work out how to get each of the four types of `rectMode()` to work: `CORNERS`, `CORNER`, `RADIUS`, `CENTER`



`setup()` is called exactly once, when the program starts

`draw()` is called repeatedly, after each time `draw()` runs, Processing goes into a wait state until the right length of time has passed to get the frame rate right.

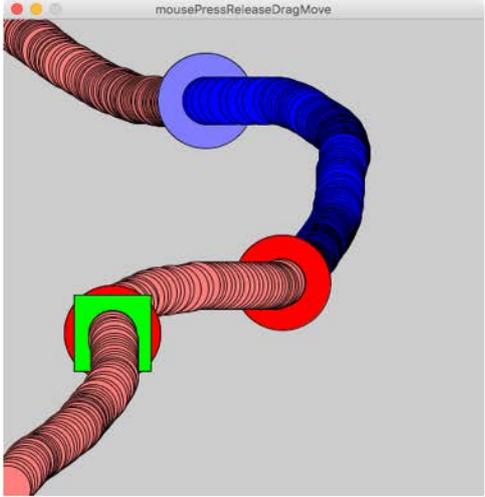
The computer itself keeps a constant watch on the keyboard and the mouse. If anything about them changes then a system **interrupt** occurs. Processing watches out for these interrupts and stores them up. It runs the appropriate callback functions, like `keyPressed()` or `mouseDragged()`, but only during the wait phase. If a key or mouse event happens during `draw()`, then Processing waits until `draw()` has finished before running the appropriate callback function.

The mouse functions

```

void setup(){
  size(500,500);
}
void draw(){
}
void mouseMoved(){
  fill(255,128,128);
  ellipse(mouseX,mouseY,50,50) ;
}
void mouseDragged(){
  fill(0,0,255);
  ellipse(mouseX,mouseY,50,50) ;
}
void mousePressed(){
  fill(128,128,255);
  ellipse(mouseX,mouseY,100,100);
}
void mouseReleased(){
  fill(255,0,0);
  ellipse(mouseX,mouseY,100,100);
}
void mouseClicked(){
  fill(0,255,0);
  rect(mouseX-40, mouseY-40,80,80);
}
}

```



37

What is going on here? [Code: mousePressReleaseDragMove]

We have written code for all five of the mouse callback routines. In each callback, Processing draws a different shape.

Moving the mouse

Small circles show `mouseMoved()` and `mouseDragged()` in pink and dark blue. `mouseMoved()` is called for mouse movement when *no* mouse button is pressed, `mouseDragged()` is called for movement when a mouse button *is* pressed.

Pressing and releasing mouse buttons

Large circles show `mousePressed()` and `mouseReleased()`, light blue for `mousePressed()` and red for `mouseReleased()`.

`mouseClicked()` causes a green square to be drawn.

`mouseClicked()` is an interesting case because a mouse click is defined as a mouse press followed by a mouse release with no mouse movement in between (i.e., the `mouseX` and `mouseY` values do not change). If this happens, Processing will first call `mousePressed()` then `mouseReleased()` and finally `mouseClicked()`. If you run the program you will see a blue circle drawn when you press the mouse. When you let go (having not moved the mouse) it will draw a red circle immediately followed by a green square.

Notice that the `draw()` function contains no code. It is, however, needed: Processing draws everything to an **off-screen buffer**, and copies this to the screen window at the end of the `draw()` function. If you don't have `draw()` then Processing will never draw anything on the screen.

The key functions

- `keyPressed()` — like `mousePressed()`
- `keyReleased()` — like `mouseReleased()`
- variables `key` and `keyCode`
- `keyTyped()` — a light version of `keyPressed()` that ignores `modifier` and `function` keys and doesn't set `keyCode`



38

`keyPressed()` and `keyReleased()` are useful if you are doing control, like in a game — there is a simple example in the program `moveBallUsingKeys`

`keyTyped()` is useful if you want textual input — all you care about then is what key has been typed, not when it was pressed and released

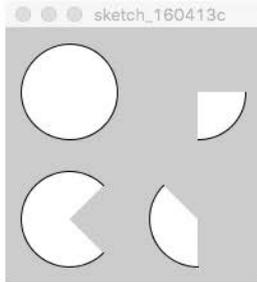
But how much of this stuff about keyboard input do you actually need to remember?

What you *need* to know is that these functions exist and that you can look up the details in the Reference manual if you want to use keyboard input.

Feeling confused by all of the different concepts that have been introduced so far? Now would be a good time to go read *Getting Started with Processing*, Chapters 3–5 (pages 13–74). This will give you a different view of the Processing world, will consolidate what you have learnt, and will fill in some bits that the lectures have not covered.

Bits of ellipses: arc()

```
size(200,200);  
  
ellipse( 50, 50, 75, 75 );  
arc( 150, 50, 75, 75, 0, PI/2 );  
arc( 50, 150, 75, 75, PI/4, 7*PI/4 );  
arc( 150, 150, 75, 75, radians(90), radians(225) );
```



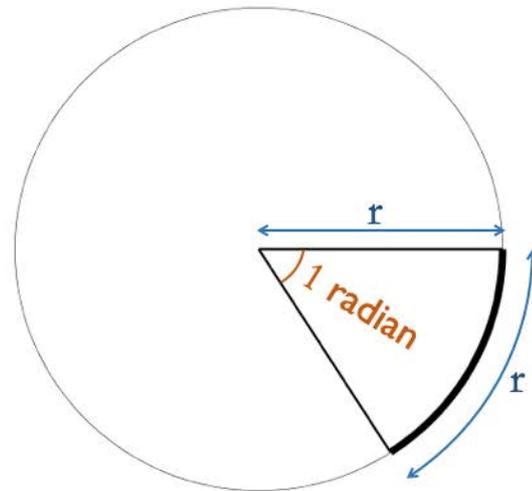
- first four parameters of arc() are same as ellipse()
- fifth and sixth parameters are the start and end angles of the arc
- angles are in **radians**

39

Getting Started with Processing, Examples 3-7 and 3-8 (pages 18–20) explain arcs and radians.

Radians

- one radian produces an arc whose length is equal to a radius
- a full circle has 2π radians
- Processing pre-defines:
 - `TWO_PI`
 - `PI`
 - `HALF_PI`
 - `QUARTER_PI`
- if you like degrees...
 - `radians(d)` converts d° to radians



You need to get comfortable with radians because, although you can always convert degrees to radians using the `radians()` function, you will find many Processing examples written using radians.

See *Fundamentals of Computer Graphics*, Section 2.3.1 (page 18).

`TWO_PI` is a full circle

`PI` is half a circle

`HALF_PI` is quarter a circle

Bouncing ball: a practical programming example

- Draw a ball
- Make it move in a straight line
- Make it bounce off the window edges
- Add gravity
- Add mouse interaction



In this practical programming example, we will go over:

`setup()`, `draw()`, `size()`, `frameRate()`, `background()` — setting up the basics
`fill()`, `stroke()`, `noStroke()`, `noFill()` — setting drawing colours
`ellipse()` — drawing a ball
variables — storage locations for information
position, velocity, acceleration — very basic physics
`if()` — conditional executing of code
conditions — use of greater than and less than, boolean results

Some of the parts of this example are also covered in *Getting Started with Processing* pages 104–107.

Debugging Processing programs:

When you need to find out what is going on inside your Processing program you can print information to the standard output window, which is that black area at the bottom of the Processing sketch window. The functions for doing this are `print()` and `println()`. The only difference is that `println()` adds a new line after it has printed.

Getting Started with Processing, Examples 5-1 and 5-2 (pages 49–50) show a couple of simple examples of using the `println()` function. Example 5-8 (page 54) has an example of a debugging use of `println()`, where the printed output allows you to check that the program is behaving in the way that you want it to.

Position, Velocity

- position (x,y)

- velocity

- change in position per unit time

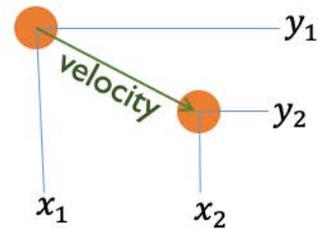
$$x_2 = x_1 + v_x$$

$$y_2 = y_1 + v_y$$

- more correctly:

$$x_2 = x_1 + v_x(t_2 - t_1)$$

$$y_2 = y_1 + v_y(t_2 - t_1)$$



42

Velocity is change in position per unit time. To work out the position after a unit time, simply add the velocity.

To be more general, multiply the velocity by the change in time to get the new position.

Position, Velocity, represented as vectors

- position (x,y) $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$
- velocity $\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$

$$x_2 = x_1 + v_x(t_2 - t_1)$$

$$y_2 = y_1 + v_y(t_2 - t_1)$$

43

It is painful to have to write out two almost identical equations: one for x and one for y .

We bundle these equations together using *vectors*.

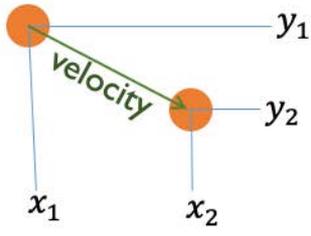
Convention is that a vector is represented by a boldface upright character, while *scalars* are represented by italic characters.

A *scalar* value is a single number. A vector is made up of two or more scalars.

See *Fundamentals of Computer Graphics*, Sections 2.4, 2.4.1, 2.4.2 (pages 21–23).

Position, Velocity, represented as vectors

- position (x,y) $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$
- velocity $\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$


$$\mathbf{x}_2 = \mathbf{x}_1 + \mathbf{v}(t_2 - t_1)$$

44

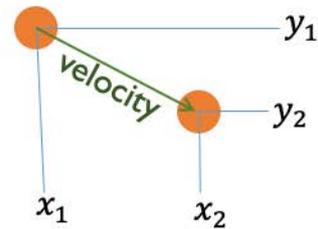
These vectors allow us to write two equations as one.

Notice that we have used the standard convention that position (x,y) is represented by a vector called \mathbf{x} . You need to get used to the idea that we might use the same letter to mean two different things. Of course, you can tell them apart (one is italic, the other is bold), and the context generally lets you know which is meant.

Position, Velocity, represented as vectors

• position (x,y) $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$

• velocity $\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$



$$\mathbf{x}_2 = \mathbf{x}_1 + \mathbf{v}\Delta t$$

45

Velocity is change in position per unit time. To add velocity to position, we need to know what the difference in time is between the two positions. Here we show the standard notation that is used for difference, the delta symbol.

$$\Delta t = t_2 - t_1$$

Position, Velocity, Acceleration

- position (x,y) $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$ $\mathbf{x}_2 = \mathbf{x}_1 + \mathbf{v}\Delta t$
- velocity $\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$
 - change in position per unit time $\mathbf{v}_2 = \mathbf{v}_1 + \mathbf{a}\Delta t$
- acceleration $\mathbf{a} = \begin{bmatrix} a_x \\ a_y \end{bmatrix}$
 - change in velocity per unit time

46

Processing has an object type, PVector, that allows you to store and manipulate vectors.

For example:

```
Pvector pos, vel ;
pos = new PVector( x, y ) ;
vel = new PVector( vx, vy ) ;
pos.add(vel) ; /* this updates pos by adding vel
to pos, similar to saying p += v */
```

```
PVector pos2 = PVector.add( pos, vel ) ; /* this
creates a new vector, the sum of pos and vel,
leaving those two vectors unaltered */
```

if statement

```
• if( condition ) {  
    code to run if condition is true  
}  
  else {  
    code to run if condition is false  
}
```

47

This is simply a reminder of one of the important ways to control the flow of code.

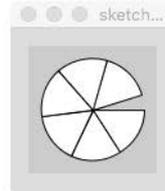
You will remember that *condition* is something that returns a boolean: either true or false.

The else part is optional.

while loops

- `while(condition) {`
 code to run while condition is true
`}`

```
float angle = 1.0 ;  
while( angle < TWO_PI ){  
    arc( 50, 50, 80, 80, angle-1, angle, PIE ) ;  
    angle += 1.0 ;  
}
```



48

The while loop keeps running so long as the *condition* is true.

Therefore there should be something inside the while loop that might cause *condition* to become false, otherwise we run forever

for loops

- `for(initialisation ; condition ; increment)`
`{`
`code to run while condition is true`
`}`

- `for(int i=0 ; i<10 ; i++){`
`println(i) ;`
`}`

Equivalent while loops

- `initialisation ;`
`while(condition){`
`code to run while`
`condition is true`
`increment ;`
`}`

- `int i=0 ;`
`while(i<10){`
`println(i) ;`
`i++ ;`
`}`

A for loop is a neat way to format more neatly a particular type of while loop.

Some early programming languages had for loops that could only handle integers.

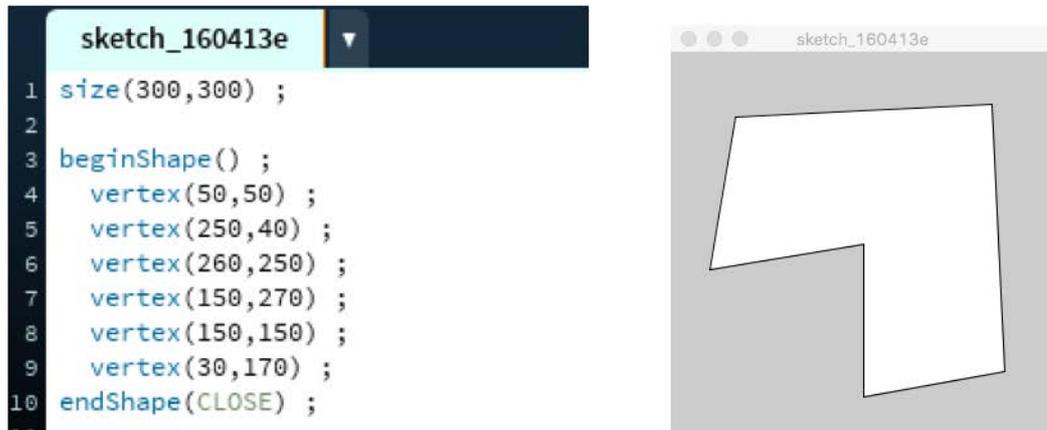
For example, in BASIC:

```
FOR i = 0 TO 9 DO
...
END
```

this would do the same as is shown in the Processing code. The Java and Processing version of for is much more powerful, because you can put (almost) anything you want as the initialisation, condition and increment statements.

For loops are so useful that you can find a whole tutorial on ways to use them in *Getting Started with Processing*, Chapter 4, starting halfway down page 40 and going to the end of the chapter (page 48).

Final primitive (for now): the polygon



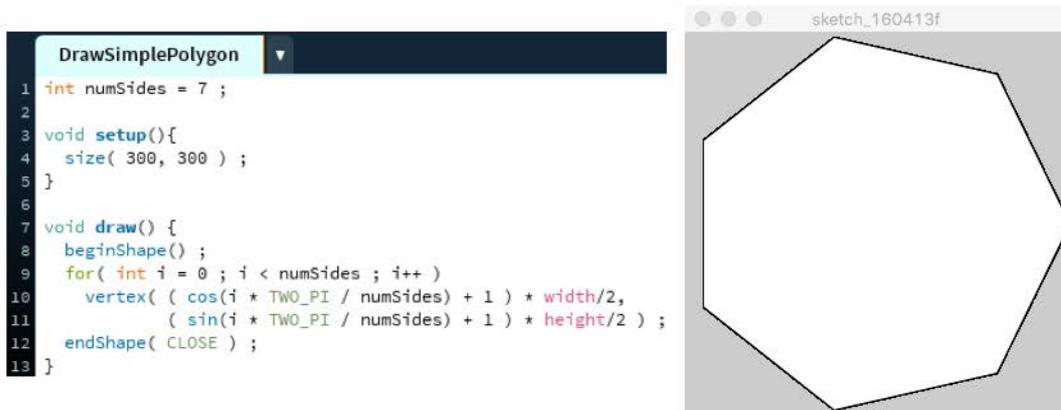
50

You will use the polygon in assignment 1.

`triangle()`, `quad()` and `rect()` all draw particular types of polygon (respectively, 3 sides, 4 sides, and 4 sides aligned with the axes).

The example shows how you can create a polygon with an arbitrary number of sides. The same idea is used in various ways throughout computer graphics. We will meet it later when we look at drawing curves.

for loops creating a series of vertices



51

Notice that, unlike the previous slide, we do not need to have an explicit list of `vertex()` commands between the `beginShape()` and `endShape()` commands. Processing is quite happy to keep track of all vertices given to it between the `beginShape()` and `endShape()` commands, and will draw the appropriate polygon when `endShape()` is called.

Look at that use of cosine and sine. We are saying that $x = \cos(\theta)$ and $y = \sin(\theta)$ for some θ . As we increase θ from 0 to `TWO_PI`, the value of (x,y) goes round a circle. We multiply by `width/2` and `height/2` to make the circle fit exactly in the window. The "+1" in the brackets is because otherwise the circle (and hence the polygon) would be centred on point $(0,0)$, the top left corner of the window. We will come back to how to move the point $(0,0)$ to somewhere more convenient later in the course.

Now consider how to change the number of sides using the x-coordinate of the mouse.

Try this:

```

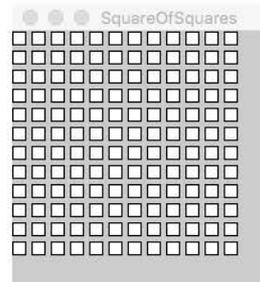
numSides = 12 * mouseX / width + 3 ; /* the +3
ensures that we have at least a triangle */

```

nested loops

- Especially useful for doing things in two dimensions

```
size(200,200) ;  
for( int i=0 ; i<12 ; i++ ){  
  for( int j=0 ; j<12 ; j++ ) {  
    rect( i*15, j*15, 10, 10 ) ;  
  }  
}
```



52

This example can be expanded on in several ways to demonstrate a range of effects.

For example, what if you wanted the squares to fill the window?

First of all, you have to decide what you mean by having the squares fill the window.

Do you want 12 x 12 squares?

If so, what size do you want them and with what spacing?

Or do you want a bunch of squares of size 10x10, spaced 5 units apart (so the squares are placed on a grid spacing of 15x15)?

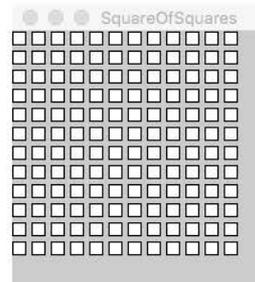
See *Getting Started with Processing*, Examples 4-10 to 4-13 (pages 44–46).

Parameters: make your code easier to read and reuse

```

size(200,200) ;
int numSquares = 12 ;
int squareSize = 10 ;
int squareSpacing = 15 ;
for( int i=0 ; i<numSquares ; i++){
  for( int j=0 ; j<numSquares ; j++ ) {
    rect( i*squareSpacing, j*squareSpacing,
          squareSize, squareSize ) ;
  }
}

```



53

Parameters help improve the readability of your code and make it easier to modify.

Now let us say that we want numSquares across the window, and we want squareSize to stay at 10.

We can calculate squareSpacing as:

```
int squareSpacing = width / numSquares ;
```

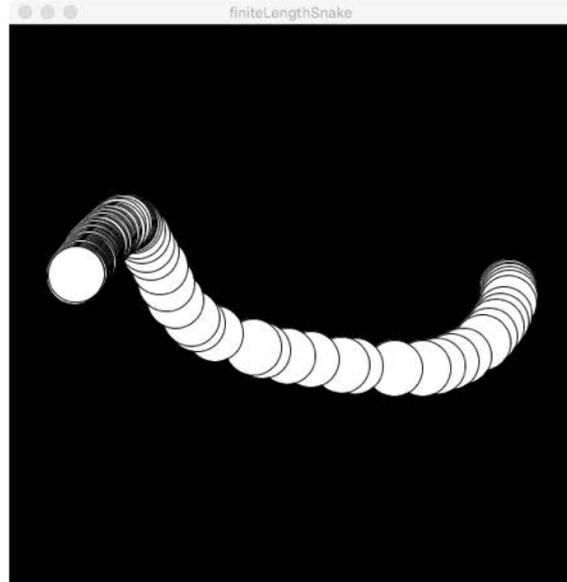
If we want to be able to cope with numSquares not exactly dividing into width then:

```
float squareSpacing = 1.0 * width
                    / numSquares ;
```

Alternatively, we could set squareSpacing to be a constant and calculate numSquares instead.

Snake: a practical programming example

- A “snake” is drawn as 100 circles
- When the mouse is moved, the oldest circle is replaced by a new circle at the mouse position
- Store the (x,y) positions of the circles in arrays



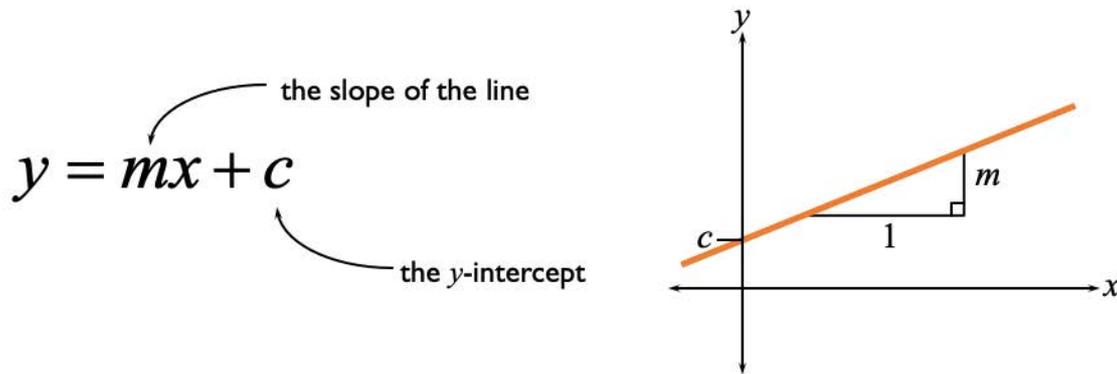
In this practical programming example, we will go over:

1. Storing the coordinates in arrays.
2. Using `mouseMoved()` to update the array.

Making the code more complex in stages:

3. Simply draw each of the 100 circles in the default fill and stroke.
4. Change to draw from back to front.
5. Change to have `noStroke` and fade out of colour as we go from front to back.

Behind the scenes: how do you draw a line?



55

We now move to considering one of the core algorithms in computer graphics: drawing a straight line. We will look at two line drawing algorithms, as worked examples of how to do good algorithm design. The algorithms are a digital differential analyser (DDA) algorithm based on the explicit line equation $y=mx+c$ and the Midpoint algorithm based on the implicit line equation $ax+by+c=0$.

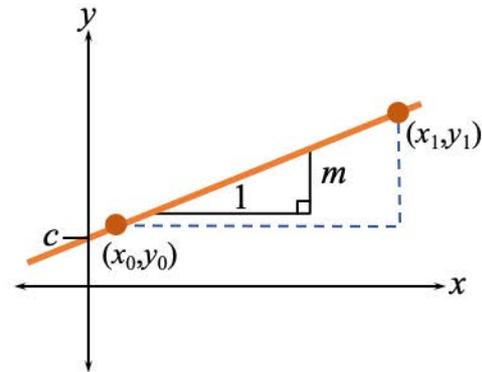
Line drawing is implemented on all graphics cards. Its main use there is as part of the polygon filling algorithm — more on that later.

Behind the scenes: how do you draw a line?

$$y = mx + c$$

For a line passing through (x_0, y_0) and (x_1, y_1) :

$$m = \frac{y_1 - y_0}{x_1 - x_0} \quad c = y_0 - mx_0$$



56

See the two similar triangles.

$y_1 - y_0$ is to $x_1 - x_0$ as m is to 1, which means $m = (y_1 - y_0) / (x_1 - x_0)$

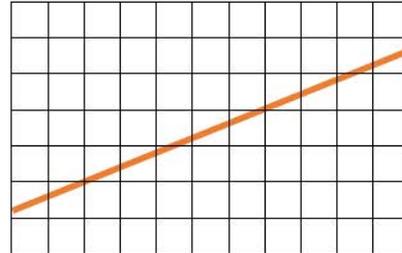
Once you have m you can work out c by substituting into the formula, with one of the given points: $y_0 = m x_0 + c$, so $c = y_0 - m x_0$

Behind the scenes: how do you draw a line?

- a straight line can be defined by:

$$y = mx + c$$

- a mathematical line is “length without breadth”
- a computer graphics line is a set of pixels
- which pixels do we need to turn on to draw a given line?



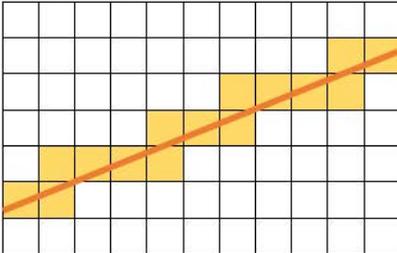
57

Without peeking at the next slide (which has the answer!), think about which pixels you would turn on to represent this line.

Which pixels do we use?

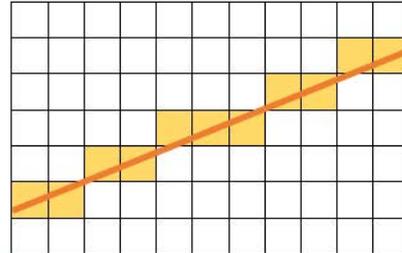
- there are two sensible alternatives:

every pixel through which the line passes



for lines of slope less than 45° ✗
either one or two pixels in each column

the "closest" pixel to the line in each column



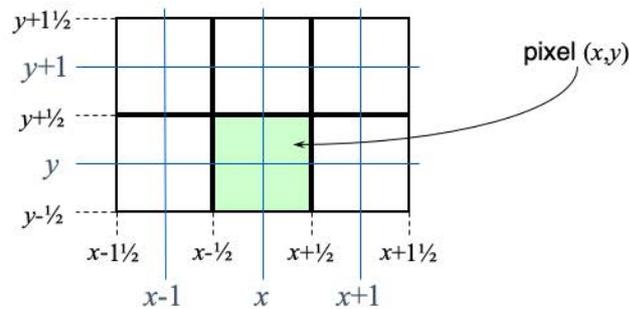
✓ for lines of slope less than 45°
always just one pixel in each column

58

The left hand version produces lines that look "lumpy". The right hand version produces lines that are the best that you can achieve with pixels, if you are only allowed to have them "on" or "off".

Line drawing algorithms — preparation I

- pixel (x,y) has its centre at real co-ordinate (x,y)
- it thus stretches from $(x-\frac{1}{2}, y-\frac{1}{2})$ to $(x+\frac{1}{2}, y+\frac{1}{2})$



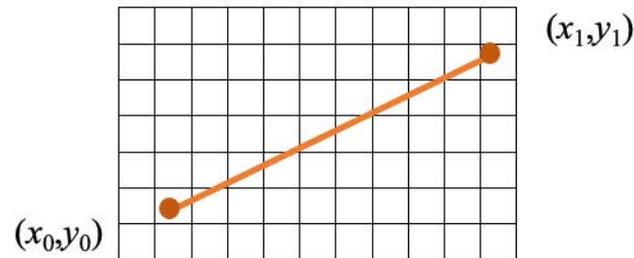
Beware: not every graphics system uses this convention. Some put real co-ordinate (x,y) at the bottom left hand corner of the pixel.

Two warnings:

1. I have used the convention that pixel (x, y) runs from $(x-\frac{1}{2}, y-\frac{1}{2})$ to $(x+\frac{1}{2}, y+\frac{1}{2})$, which is also the convention used in *Fundamentals of Computer Graphics*. Others may make the assumption that pixel (x, y) runs from (x, y) to $(x+1, y+1)$.
2. You will notice that, in the lecture notes and in *Fundamentals of Computer Graphics*, the y -axis runs upwards, while in *Processing* the y -axis runs downwards. You just have to get used to the inconsistency. All mathematics texts will have the y -axis pointing up. Some graphics systems have it pointing down.

Line drawing algorithms — preparation 2

- the line goes from (x_0, y_0) to (x_1, y_1)
- the line lies in the first octant: $(0 \leq m \leq 1)$ and $x_0 < x_1$



60

We can get $x_0 \leq x_1$ by exchanging the end points if necessary.

There are then four cases that need variations on the same algorithm:

- $m < -1$ one pixel in each row
- $-1 \leq m < 0$ one pixel in each column
- $0 \leq m \leq 1$ one pixel in each column
- $1 < m$ one pixel in each row

We'll develop an algorithm for the case $0 \leq m \leq 1$ where there is one pixel turned on in each column and where the line slopes upwards from left to right.

Naïve algorithm for integer end points

```

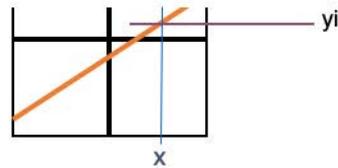
Initialisation
float m = (y1 - y0) / (x1 - x0)
float c = y0 - m * x0
int x = x0
int y = y0

Iteration
WHILE x ≤ x1 DO
    DRAW(x,y)
    x = x + 1
    yi = m * x + c

    y = ROUND(yi)

END WHILE

```



y_i is the floating point “y-intercept”: the y value at which the line we are drawing *intercepts* the vertical line at x.

61

y_i is a floating point variable
 y is an integer variable

For this naïve algorithm, we could dispense with y entirely, and just use the command `DRAW(x,ROUND(yi))`, but it is useful for the algorithm’s development in the next slides to include y at this stage.

`ROUND()` is a function that rounds a floating point number to the nearest integer

`DRAW()` is a function that turns on a single pixel. It is equivalent to Processing’s `point()` function.

DDA line algorithm for integer end points

Initialisation

```
float m = (y1 - y0) / (x1 - x0)
int x = x0
float yi = y0
int y = y0
```

Iteration

```
WHILE x ≤ x1 DO
  DRAW(x,y)
  x = x + 1
  yi = yi + m

  y = ROUND(yi)
END WHILE
```

DDA = "digital differential analyser" — it uses the difference (m) in y between one column and the next

62

We have replaced

$$y_i = m * x + c$$

with

$$y_i = y_i + m$$

This gives an immediate performance gain because, on each time round the loop, we have replaced a multiplication and an addition by a single addition.

Over a very long line, this might mean that we accumulate some small error because of doing thousands of additions, but no line in computer graphics is going to be more than a few thousand pixels long, so the errors will be irrelevant.

DDA line algorithm for floating point end points

Initialisation

```
float m = (y1 - y0) / (x1 - x0)
```

```
int x = ROUND(x0)
```

```
float yi = y0 + m * (x - x0)
```

Iteration

```
WHILE x ≤ ROUND(x1) DO
```

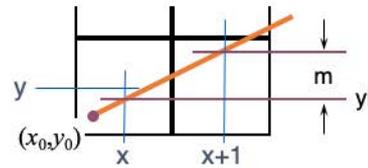
```
  DRAW(x,y)
```

```
  x = x + 1
```

```
  yi = yi + m
```

```
  y = ROUND(yi)
```

```
END WHILE
```



We need to calculate the initial y from the rounded off initial position of x_0 because we will not necessarily get the right answer by rounding x_0 and y_0 independently.

63

The only difference is in the initialization.

Today's machines have floating point multiplication that is (almost) as fast as floating point addition. It is still worth using the DDA algorithm because it replaces two floating point operations in the loop with one.

You could ask why we do not simply round the end points to the nearest integer and then use an integer version of the algorithm. Rounding end points is not good because it would mean (1) that animation would become jerky as the endpoints of the line move and (2) that dividing any line in two could lead to the two halves drawing (slightly) different sets of pixels to the un-split line.

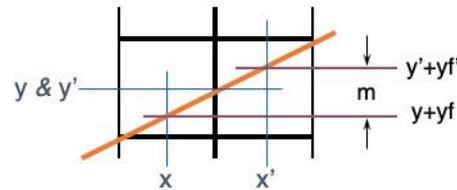
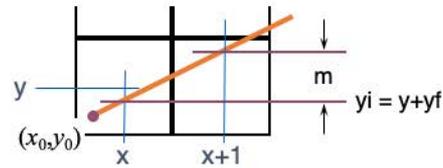
DDA algorithm without ROUNDing in the loop

```

float m = (y1 - y0) / (x1 - x0)
int x = ROUND(x0)
float yi = y0 + m * (x - x0)
int y = ROUND(yi)
float yf = yi - y
  
```

```

WHILE x ≤ ROUND(x1) DO
  DRAW(x,y)
  x = x + 1
  yf = yf + m
  IF ( yf > ½ ) THEN
    y = y + 1
    yf = yf - 1
  END IF
END WHILE
  
```



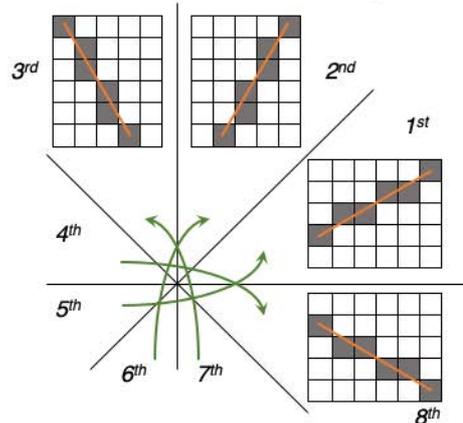
Splitting the y -coordinate into fractional (yf) and integer (y) parts avoids rounding on every cycle.

64

Replacing ROUND: Here we split y into integer and fractional parts and replace the ROUND operation with an IF statement.

Line drawing algorithm — more details

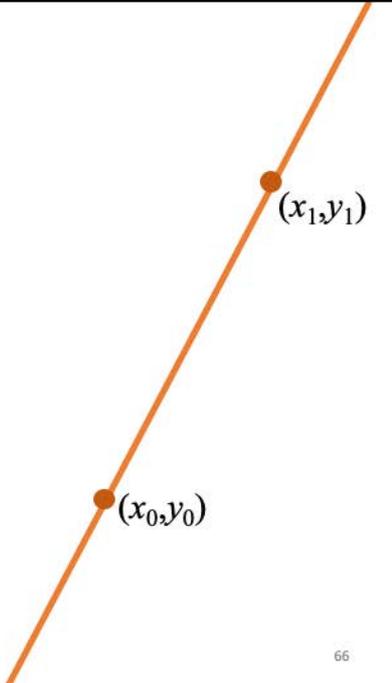
- we assumed that the line is in the first octant
 - can do fifth octant by swapping end points
- therefore need four versions of the algorithm



Exercise: work out what changes need to be made to the algorithm for it to work in each of the other three octants

Ways to represent a line

- Explicit
 $y = mx + c$
- Implicit
 $ax + by + c = 0$
- Parametric
 $x(t) = (1 - t)x_0 + tx_1$
 $y(t) = (1 - t)y_0 + ty_1$



66

Explicit and implicit representation: see *Fundamentals of Computer Graphics*, Section 2.5.2 starting halfway down page 33 (pages 33–35).

Parametric representation: see *Fundamentals of Computer Graphics*, Section 2.5.6 (pages 39–40).

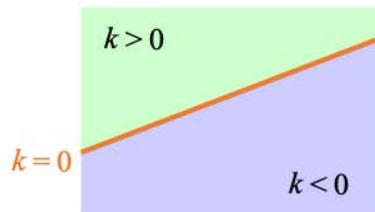
The midpoint line drawing algorithm I

- use an equation based on the implicit formula for a line:

$$k = ax + by + c$$

- this divides the plane into three regions:

- above the line $k > 0$
- below the line $k < 0$
- on the line $k = 0$



For a line segment from (x_0, y_0) to (x_1, y_1) , the line is defined by:

$$a = y_0 - y_1$$

$$b = x_1 - x_0$$

$$c = x_0 y_1 - x_1 y_0$$

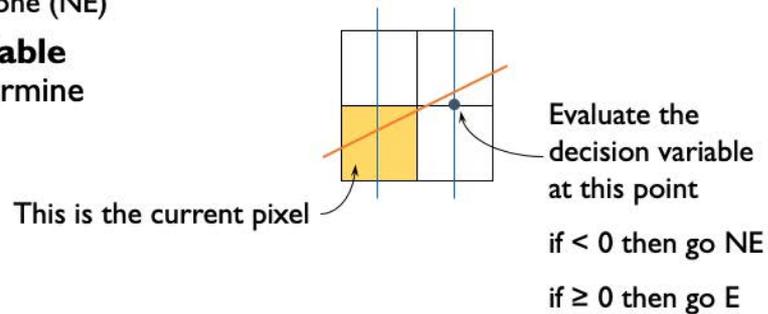
67

The values of a , b , and c are those in Equation 2.18 of *Fundamentals of Computer Graphics* (page 34).

Note that this version of c is different from the c in $y=mx+c$.

The midpoint line drawing algorithm 2

- first work out the iterative step
 - it is often easier to work out what should be done on each iteration and only later work out how to initialise and terminate the iteration
- given that a particular pixel is on the line, the next pixel **must** be either
 - immediately to the right (E) or
 - to the right and up one (NE)
- use a **decision variable** (based on k) to determine which way to go



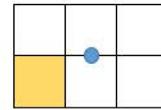
68

The midpoint algorithm is covered in *Fundamentals of Computer Graphics*, Section 8.1.1 (pages 163–165). It would help to read also the earlier parts of Chapter 8 (pages 161–163).

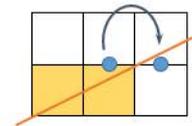
The code letters “E” and “NE” stand for “East” and “North-East”, as if the pixel grid was a map with North at the top.

The midpoint line drawing algorithm 3

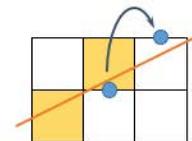
- decision variable needs to make a decision at point $(x_p+1, y_p+1/2)$ $k = a(x_p + 1) + b(y_p + 1/2) + c$



- if go E then the new decision variable is at $(x_p+2, y_p+1/2)$ $k' = a(x_p + 2) + b(y_p + 1/2) + c$
 $= k + a$



- if go NE then the new decision variable is at $(x_p+2, y_p+1 1/2)$ $k' = a(x_p + 2) + b(y_p + 3/2) + c$
 $= k + a + b$



69

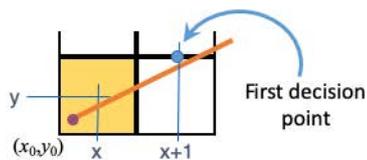
The decision variable is calculated from the formula $k=ax+by+c$ at the location $(x,y)=(x_p+1,y_p+1/2)$.

When you decide which way to move, the decision variable moves one unit to the right and either zero or one unit up. You can then calculate the new decision variable at the new point. The beauty of this method is that the new decision variable is easy to compute from the previous one and the computation does not require multiplication: it is just adding either a or $a+b$ to the current value of k .

The midpoint line drawing algorithm 4

Initialisation

```
float a = -(y1 - y0)
float b = (x1 - x0)
float c = x0*y1 - x1*y0
int x = ROUND(x0)
int y = ROUND((-a*x-c)/b)
float k = a * (x+1) + b * (y+1/2) + c
```



Iteration

```
WHILE x ≤ ROUND(x1) DO
  DRAW(x,y)
  IF k > 0 THEN
    k = k + a ← E case
  ELSE
    k = k + a + b
    y = y + 1 ← NE case
  END IF
  x = x + 1
END WHILE
```

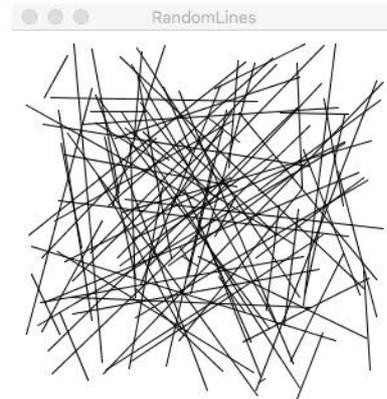
70

As with many graphics algorithms, the initialization is more complicated than the iterative loop.

Assignment 2 extension — line drawing

- Implement one of the line drawing algorithms
- Check it against Processing's own line drawing algorithm

```
RandomLines
1 size(300,300);
2 background(255);
3 stroke(0);
4 for( int i=0;i<100;i++){
5   line(random(10,290),random(10,290),
6         random(10,290),random(10,290));
7 }
```



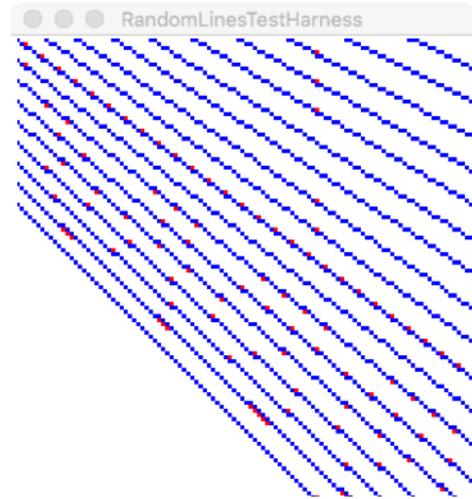
71

Assignment 2 extension — how?

- Write your own line drawing function

```
myLine( int x0, int y0,  
        int x1, int y1)
```

- Use `point(x,y)` to draw individual pixels
- Write a test harness:
 - A test function that draws your line in red then Processing's line in blue
 - A for loop that calls the test function with a carefully chosen set of lines (which might be a random set of lines)
 - Then swap the order of drawing in the test function and run again
 - One-pixel-away errors are acceptable



72

Assignment 2 extension — things to remember

```

RandomLinesTestHarness
1 void setup(){
2   size(300,300);
3   noLoop(); // call draw() only once
4   noSmooth(); // don't draw smooth lines
5 }
6
7 void testLines( int x0, int y0, int x1, int y1 ){
8   stroke(255,0,0); // red
9   myLine(x0,y0,x1,y1);
10  stroke(0,0,255); // blue
11  line(x0,y0,x1,y1);
12 }
13
14 void draw(){
15  background(255);

```

- If you define other functions you must also use `setup()` and `draw()`
- Use `noLoop()` to make Processing call `draw()` only once, rather than looping
- Use `noSmooth()` to make Processing draw pixelated lines

73

Processing has two modes, which it calls “active” and “static” modes.

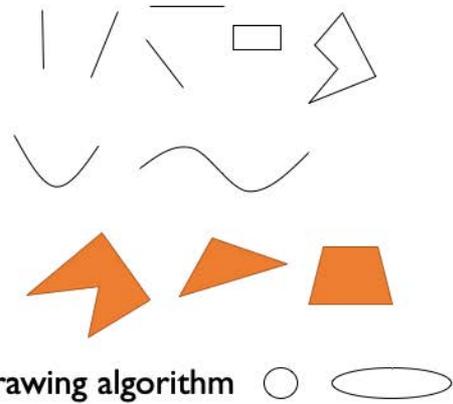
“static” mode has no functions at all, with all the code executed in order, once. It creates a static picture, because there is no update loop.

“active” mode uses `setup()` and `draw()` to make that update loop.

The challenge is that you cannot mix “active” and “static” modes. This means that you can define your own functions only in “active” mode. If you try to define your own functions in “static” mode you will get either a strange error message (such as “unexpected token”, which doesn’t make sense unless you realise that in active mode you cannot put statements outside functions) or the more helpful message ‘It looks like you’re mixing “active” and “static” modes.’ Remember: if you want to define any of your own functions, you must also use `setup()` and `draw()`. You cannot mix the two modes.

Uses of a line drawing algorithm

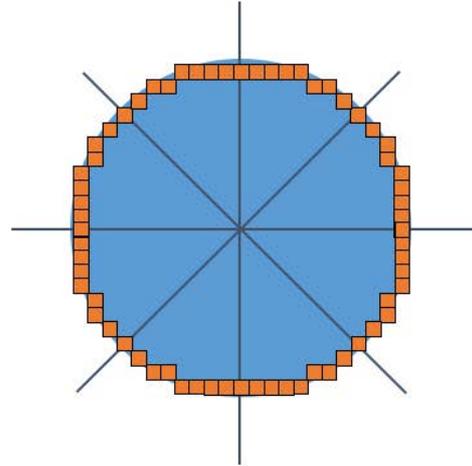
- to draw lines
- to draw curves as a sequence of lines
 - coming later in the course
- a variant is used in polygon filling algorithms
 - to run up each of the polygon's edges
- a similar structure can be used for a circle-drawing algorithm



74

Midpoint circle algorithm I

- implicit equation of a circle is $x^2 + y^2 = r^2$
 - centred at the origin
- decision variable can be $k = x^2 + y^2 - r^2$
 - $k = 0$ on the circle, $k > 0$ outside, $k < 0$ inside
- divide circle into eight octants
 - on the next slide we consider only the second octant, the others are similar

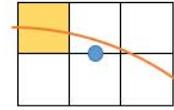


75

Midpoint circle algorithm 2

- decision variable needs to make a decision at point $(x_p+1, y_p-1/2)$

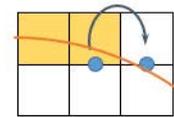
$$k = (x_p + 1)^2 + (y_p - 1/2)^2 - r^2$$



- if go E then the new decision variable is at $(x_p+2, y_p-1/2)$

$$k' = (x_p + 2)^2 + (y_p - 1/2)^2 - r^2$$

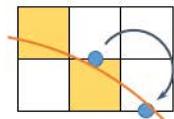
$$= k + 2x_p + 3$$



- if go SE then the new decision variable is at $(x_p+2, y_p-1 1/2)$

$$k' = (x_p + 2)^2 + (y_p - 3/2)^2 - r^2$$

$$= k + 2x_p + 3 - 2y_p + 2$$



76

This detail is for the *second* octant.

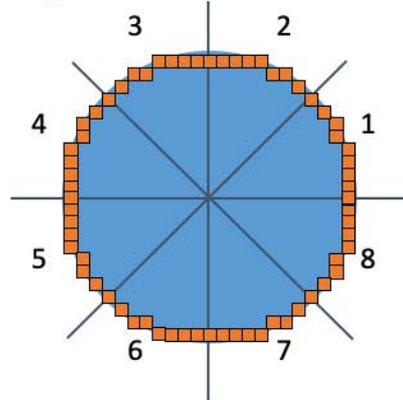
This slide only covers the *increment* part of the algorithm. It assumes that we already know the location of the previous pixel and the current value of k .

Homework: How would you initialise the algorithm? At which pixel would you start?

Midpoint circle algorithm 3

- Drawing an origin-centred circle in all eight octants

Call	Octant
Draw(x,y)	2
Draw(-x,y)	3
Draw(-x,-y)	6
Draw(x,-y)	7
Draw(y,x)	1
Draw(-y,x)	4
Draw(-y,-x)	5
Draw(y,-x)	8



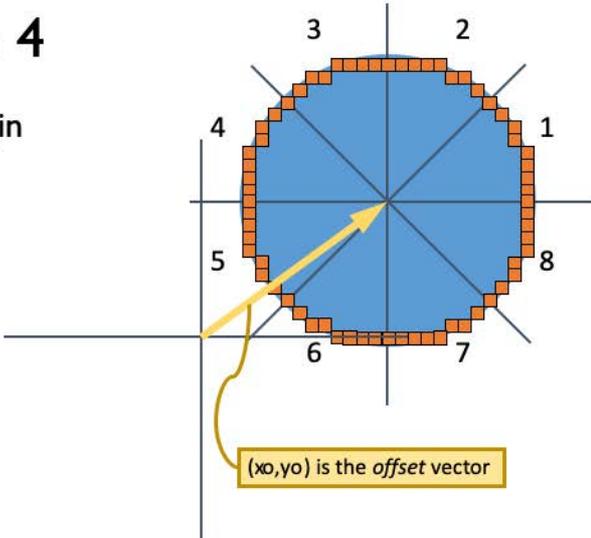
The second-octant algorithm thus allows you to draw the whole circle.

77

Midpoint circle algorithm 4

- Drawing an circle offset from the origin

Call	Octant
<code>Draw(x+xo,y+yo)</code>	2
<code>Draw(-x+xo,y+yo)</code>	3
<code>Draw(-x+xo,-y+yo)</code>	6
<code>Draw(x+xo,-y+yo)</code>	7
<code>Draw(y+xo,x+yo)</code>	1
<code>Draw(-y+xo,x+yo)</code>	4
<code>Draw(-y+xo,-x+yo)</code>	5
<code>Draw(y+xo,-x+yo)</code>	8



`(xo, yo)` is the *offset* vector

78

Homework: Implement this algorithm in Processing. Compare it against Processing's own circle drawing (using the `ellipse()` function with equal width and height parameters).

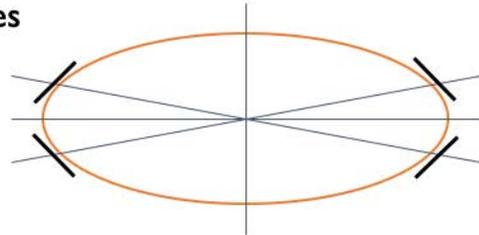
Where can we go from here?

- Can we derive similar algorithms for all curved shapes?

NO!

- A similar method *can* be derived for ellipses

- but: cannot naively use octants
 - use points of 45° slope to divide ellipse into eight sections
- and: ellipse must be axis-aligned
 - there is a more complex algorithm which can be used for non-axis aligned ellipses
- early drawing packages used just ellipses & segments of ellipses



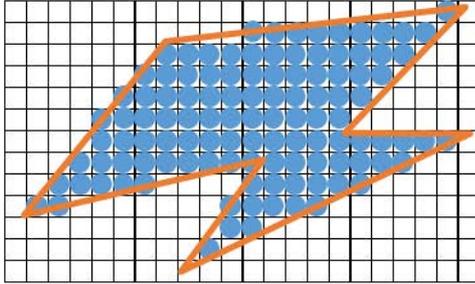
- But graphic design & CAD need something with more flexibility
 - spline curves, which need a totally different approach to drawing

79

This idea of drawing only one pixel in each column (or row) does not extend well as the shapes get more complicated. We need to consider a different way of drawing more complex curves. We will get to that later in the course.

Polygon filling

- which pixels do we turn on?



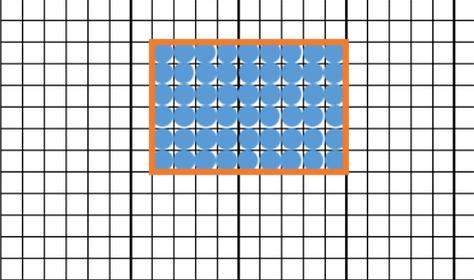
- those whose *centres* lie inside the polygon
 - this is a naïve assumption, but is sufficient for now

80

Why is it naïve? Because we would really want a pixel to be shaded in proportion to the fraction of the pixel that is covered by the polygon rather than just being a binary choice: fully on or fully off.

There is also a subtlety: what about pixels whose centres lies exactly on an edge. If two polygons share that edge, we don't want the pixel to belong to *both* polygons. In this case we can say, for example, that pixels on an edge at the left side of the polygon belong to that polygon, while pixels on edges at the right do not.

Axis-aligned rectangle — the easiest polygon to fill



- on each row from top to bottom:
- fill from left to right

```

int left=10;
int right=90;
int top=20;
int bottom=80;

stroke(255, 0, 0); // red
for ( int y=top; y<bottom; y++) {
  for (int x=left; x<right; x++) {
    point(x, y);
  }
}

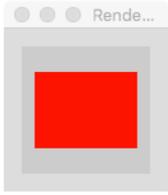
```

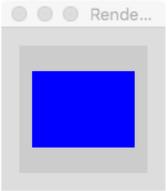
```

int left=10;
int right=90;
int top=20;
int bottom=80;

noStroke();
fill(0, 0, 255); // blue
rect(left, top, right-left, bottom-top);

```



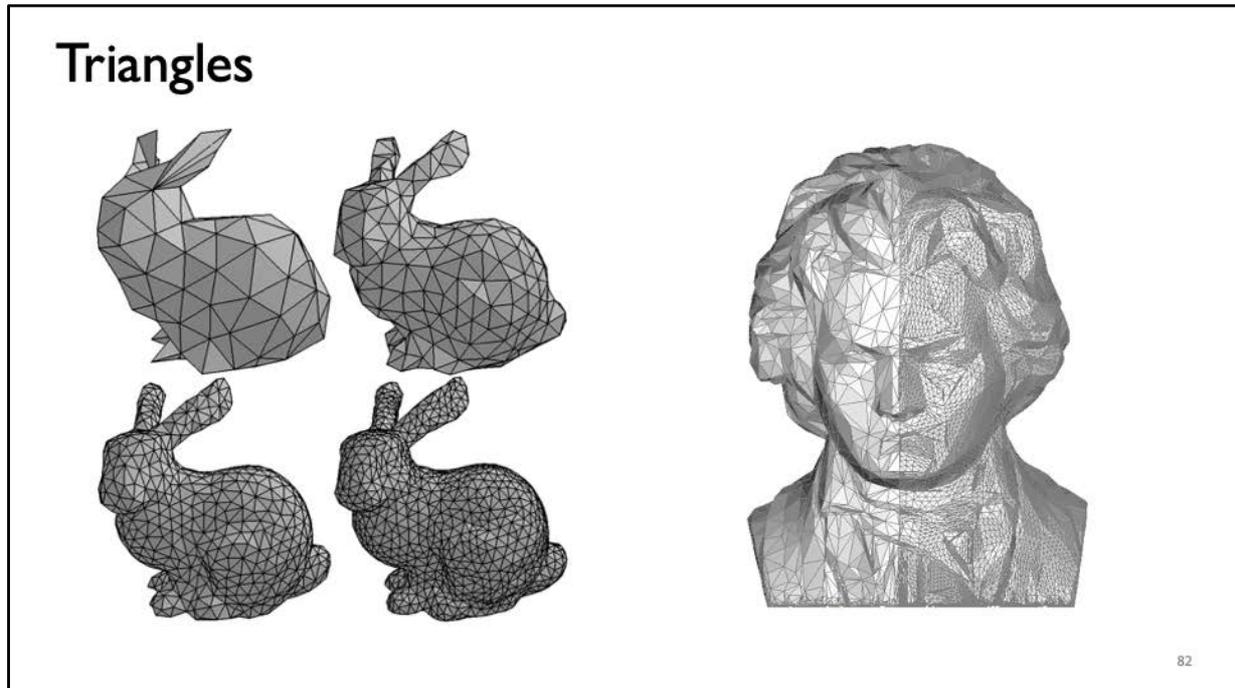


81

The top fragment of code is a loop to draw every pixel in the polygon.

The bottom fragment of code is the equivalent Processing function call of `rect()`, which internally does just what happens in the top fragment.

A note on APIs. Processing is a graphical API (Application Programming Interface) that has a whole bunch of function to do things so that you don't have to write them. In this course we are learning both about how to use that API and about what is going on behind the scenes. When writing computer graphics code, you will always be using APIs, some of which have many more features than Processing.

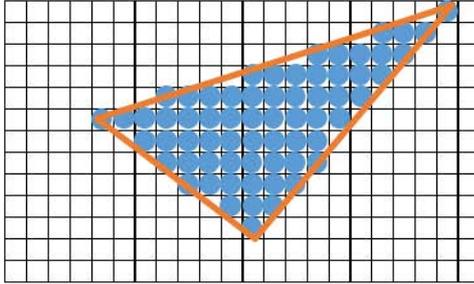


Triangles are the default primitive for graphics cards to draw because:

- Guaranteed to be convex (so no nasty cases to consider).
- Guaranteed to have only three vertices and three edges (so can explicitly have hardware optimised for the number, rather than having to cope with an arbitrary number of vertices and edges)
- In 3D, guaranteed to be planar.

We get to 3D drawing later in the course...

Triangle filling — what every graphics card does



```

work out top and bottom
for( int y=top; y<bottom; y++){
  work out left and right
  for( int x=left; x<right; x++){
    point( x, y );
  }
}

```

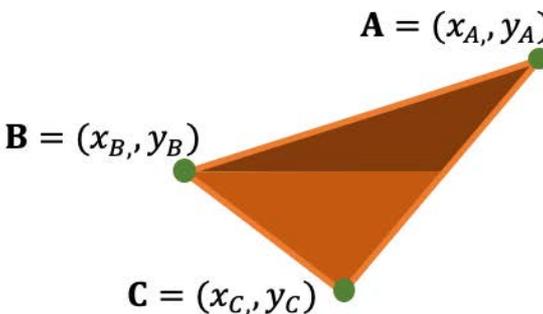
- on each row from top to bottom:
- fill from left to right between the triangle's edges

83

The triangle drawing algorithm presented in these notes is based on using a modified version of the line drawing algorithm to scan up the edges of the triangle, and to fill between those edges.

The textbook presents an alternative algorithm that uses barycentric coordinates. This can be found in of *Fundamentals of Computer Graphics* Section 8.1.2 (pages 166–169). You'll also need to read up on barycentric coordinates in Section 2.7 (pages 44–48).

Triangle filling — what every graphics card does



$A = (x_A, y_A)$

$B = (x_B, y_B)$

$C = (x_C, y_C)$

- fill rows from y_A to y_B
- then fill rows from y_B to y_C

```

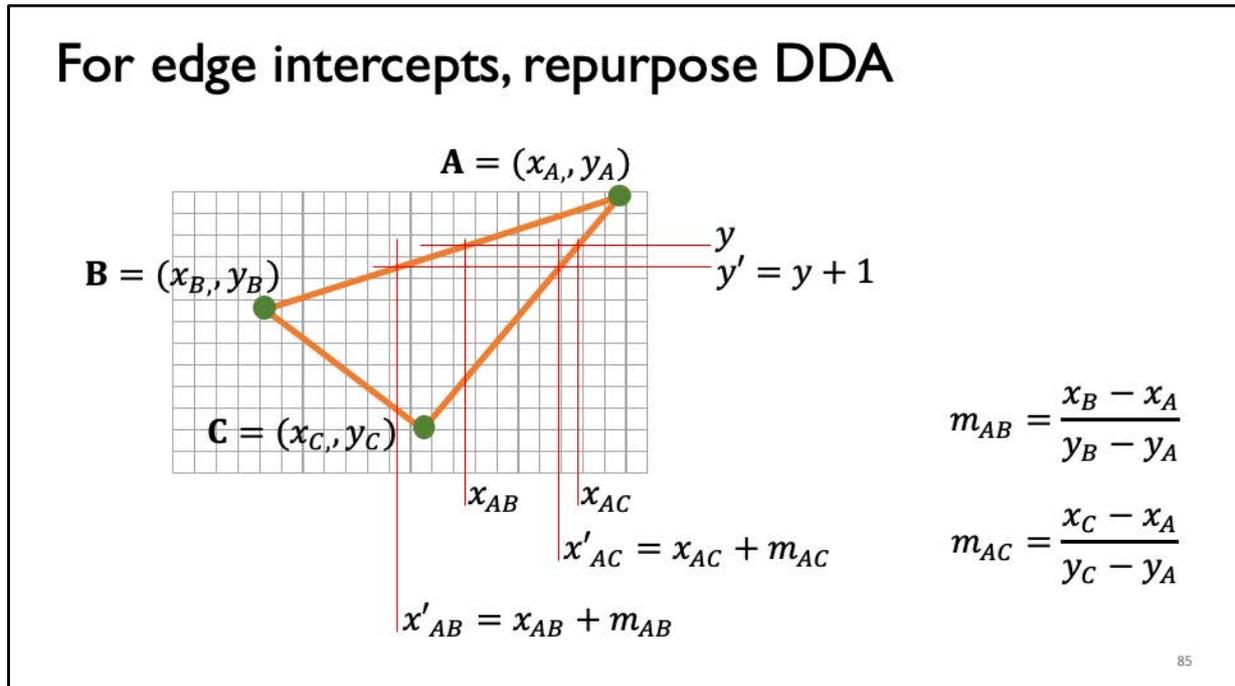
work out correct order for points, sorted on y value: A, B, C
for( int y=yA; y<yB; y++){
  work out left and right
  for( int x=left; x<right; x++){
    point( x, y ) ;
  }
}
for( int y=yB; y<yC; y++){
  work out left and right
  for( int x=left; x<right; x++){
    point( x, y ) ;
  }
}

```

84

Here we are using Processing's convention that the y -axis points downwards, so increasing y moves you down.

Notice that here we are assuming that the vertices of the triangle are at integer coordinates. What changes would be needed to allow floating point coordinates?

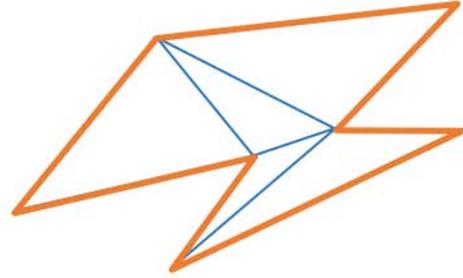
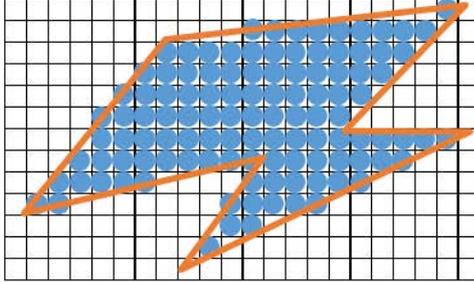


This is an extension of our line drawing algorithm.

With two edges, we have to keep track of two lines and therefore two intercepts, one for each edge.

In the diagram, the current horizontal scan line, y , has two intercepts, x_{AB} and x_{AC} . When we increment y to y' , we need to increment these two intercepts also, by the appropriate values, m_{AB} and m_{AC} .

Polygon filling — the generic case



- Graphics card solution: convert polygon to triangles
- Alternative: generalise triangle algorithm to arbitrary number of edges

86

Scanline polygon fill algorithm

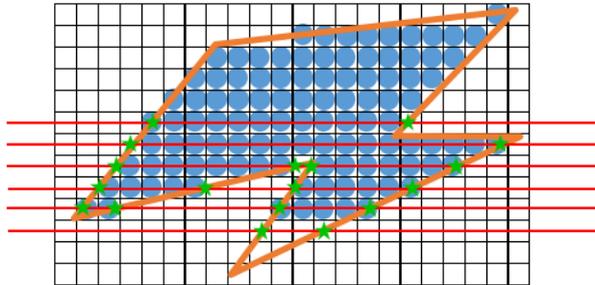
- Initialisation*
- 1) take all polygon edges and place in an *edge list (EL)*, sorted on lowest y value
 - 2) start with the first scanline that intersects the polygon, get all edges which intersect that scan line and move them to an *active edge list (AEL)*

- Iteration*
- 3) for each edge in the AEL: find the intersection point with the current scanline; sort these into ascending order on the x value
 - 4) fill between pairs of intersection points
 - 5) move to the next scanline (increment y); move new edges from EL to AEL if start point $\leq y$; remove edges from the AEL if endpoint $< y$; if any edges remain in the AEL go back to step (3)

87

This is the algorithm for an arbitrary number of vertices and edges.

Scanline polygon fill example

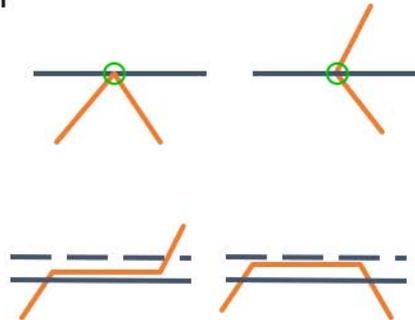


88

Homework: try this yourself. Get a piece of squared paper and draw an arbitrary polygon on it, with five or six edges. Then run the scanline polygon fill algorithm by hand, keeping track of which edges are in the edge list, which are in the active edge list, and which have been discarded.

Scanline polygon fill details

- how do we efficiently calculate the intersection points?
 - use a variant of the DDA algorithm to do incremental calculation
 - store current x value, increment value m, starting and ending y values
 - on increment do a single addition $x=x+m$
- what if endpoints exactly intersect scanlines?
 - need to ensure that the algorithm handles this properly
- what about horizontal edges?
 - can throw them out of the *edge list*, they contribute nothing



89

How do we handle edge points that exactly intersect?

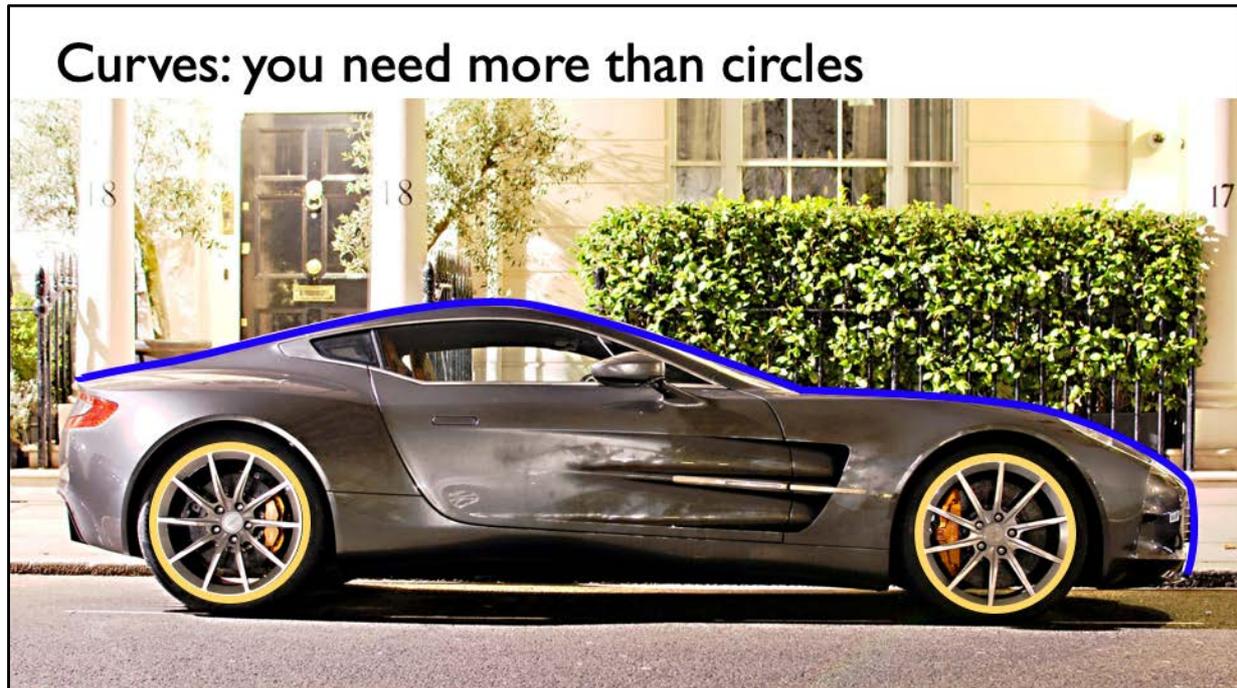
In the left hand case, the point needs to be included twice: once for each edge

In the right hand case, the point needs to be included once: once for one of the edges and not for the other

This requires that you are careful in how you code your conditions in the algorithm for when an edge is added to or removed from the AEL.

Do we really throw away horizontal edges?

Rather than explicitly testing for horizontal edges, it is neater and simpler to code so that a horizontal edge gets added to the AEL and then removed from the AEL. This requires that, on each scanline you check for new lines to add to the AEL before you check for lines to remove from the AEL and do both before you draw anything.



Curves are covered in *Fundamentals of Computer Graphics*, Chapter 15 (page 339–384). That chapter goes deeper into the material than is necessary for this course. In this course we will cover particularly Bézier curves (Section 15.6.1, pages 365–372). Understanding the material in Section 15.6.1 requires you to have at least read Sections 15.2 and 15.3, even if you are not totally confident of that earlier material.

Curves: you need splines



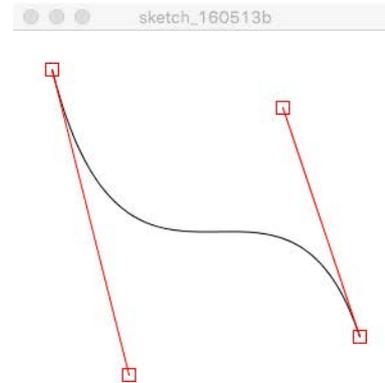
The spline here is created by a flexible piece of wood held in place by five lead “ducks”. The wood naturally bends to a shape that is piecewise cubic: that is, between each pair of ducks the curve is a cubic function.



Splines were originally used in boat building. Boats would be planned out in a process call *lofting*, where the designers would go up into the very large loft of the factory or boat house, lay out enormous pieces of paper, and draw the cross-sections of the boat at real-life scale, using the flexible wood and ducks to get the lines right. The lofting process dates back at least 200 years.

Bézier cubic splines in Processing

```
size(300,300);  
background(255);  
  
noFill();  
bezier(30,30,90,270,210,60,270,240);  
  
stroke(255,0,0);  
rectMode(CENTER);  
rect(30,30,10,10);  
rect(90,270,10,10);  
rect(210,60,10,10);  
rect(270,240,10,10);  
  
line(30,30,90,270);  
line(210,60,270,240);
```



93

The Bézier curve has four control points. In this case they are at (30,30), (90,270), (210,60), (270,240).

The Bézier curve starts at the first point, heading in the direction of the second.

It ends at the fourth point, coming in from the direction of the third.

The first and fourth points define the ends of the curve. The second and third points define the tangent vectors at the ends.

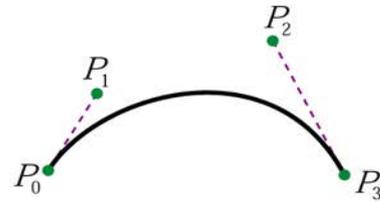
The Processing sketch, `BezierInteraction`, allows you to play with a Bézier curve.

Bézier cubic

- Bézier cubic is defined by two end points and two other control points

$$\begin{aligned}P(t) &= (1-t)^3 P_0 \\ &+ 3t(1-t)^2 P_1 \\ &+ 3t^2(1-t) P_2 \\ &+ t^3 P_3\end{aligned}$$

where: $P_i = (x_i, y_i)$
 $0 \leq t \leq 1$



Pierre Bézier worked for Renault in the 1960s

94

Bézier cubics: *Fundamentals of Computer Graphics* Section 15.6.1 (pages 365–372).

Work out what happens at $t=0$ and $t=1$.

Bézier properties

- Tangent vectors at end points are defined by the end point and its adjacent point

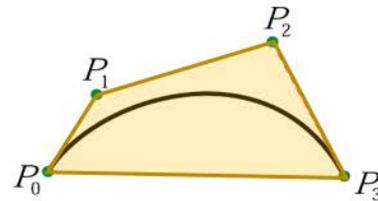
$$T_0 = 3(P_1 - P_0) \quad T_1 = 3(P_3 - P_2)$$

- Weighting functions are cubic Bernstein polynomials

$$(1-t)^3 \quad 3t(1-t)^2 \quad 3t^2(1-t) \quad t^3$$

- Weighting functions sum to one $\sum_{i=0}^3 b_i(t) = 1$

- Bézier curve lies within convex hull of its control points
 - because weights sum to 1 and all weights are positive



95

Tangent vectors are the direction that the curve leaves the start point and the direction that it arrives at the end point.

Look at the form of the Bernstein polynomials. For cubics they are:

$$b_m = k_m (1-t)^m t^{(3-m)}$$

where k_m is a constant. For cubics the constants are $\{1,3,3,1\}$, which is one of the rows of Pascal's triangle. There is a general form for these polynomials for arbitrary degree, n

$$b_m^n = k_m^n (1-t)^m t^{(n-m)}$$

For any degree, the weighting functions always sum to one. This is vital for ensuring that the cubic curve is invariant to translation. That is, if you move all the control points by the same distance in the same direction, the entire curve moves the same distance in the same direction.

The convex hull property is important when we are doing bounding box calculations (see later) because it means that we know that if the control points all lie on one side of any given line, the entire Bézier cubic curve does too.

Drawing a Bézier cubic – iterative method

- draw as a set of short line segments equispaced in parameter space, t

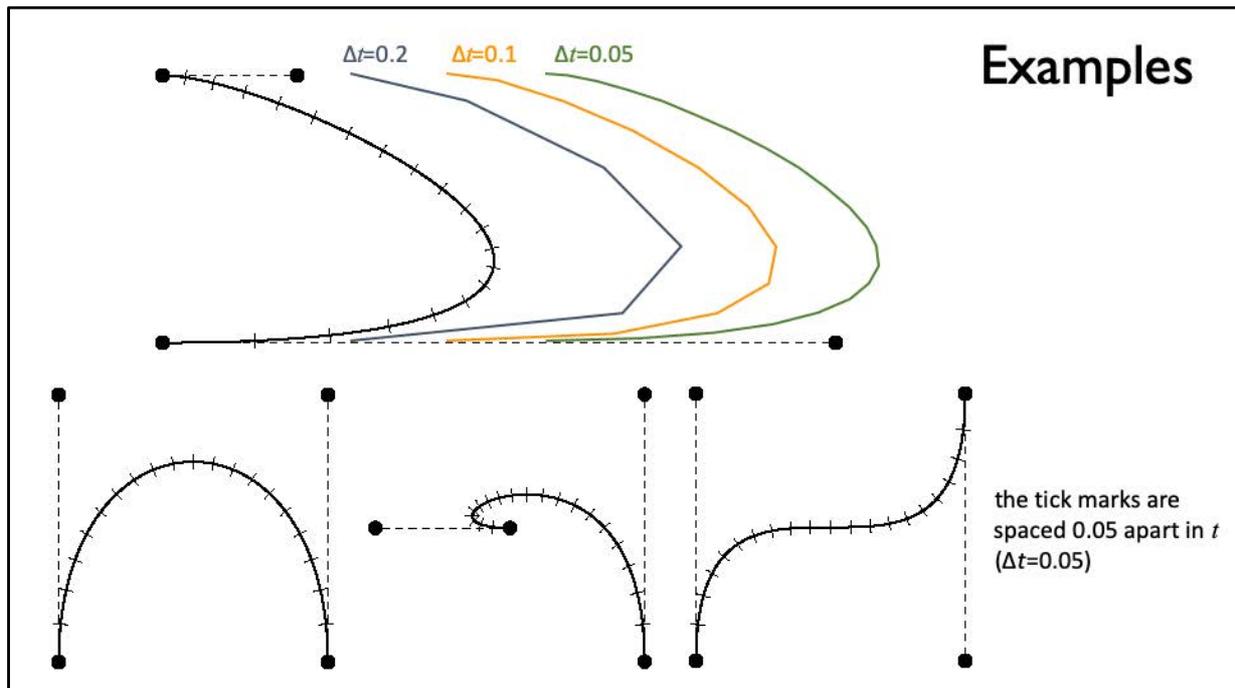
```
(x0,y0) = Bezier(0)
FOR t = 0.05 TO 1 STEP 0.05 DO
  (x1,y1) = Bezier(t)
  DrawLine( (x0,y0), (x1,y1) )
  (x0,y0) = (x1,y1)
END FOR
```

- problems:
 - cannot fix a number of segments that is appropriate for all possible Beziers: for some there will be either too many or too few segments
 - distance in real space, (x,y) , is not linearly related to distance in parameter space, t

96

The basic idea is that we know how to draw straight lines quickly, so it is straightforward to approximate the Bézier as a sequence of straight lines. In this example, we use 20 straight lines (it is 20 because the STEP is 0.05, t ranges from 0 to 1, and $20 \times 0.05 = 1$).

The iterative method fixes the number of line segments that should be used. The adaptive method (see two slides ahead) adapts the number of line segments to suit the Bézier curve and the pixel resolution of the display.



The top example shows, at left, tick marks spaced evenly in parameter space at a STEP spacing of 0.05, superimposed on the actual curve.

At right are three approximations drawn with different STEP values:

0.2 approximates the curve using 5 lines,

0.1 approximates it with 10 lines, and

0.05 approximates it with 20 lines.

Notice that the right hand example appears visually to be curved over most of its length but that you can make out the individual line segments at the place where the curve turns the tightest. This means that a STEP size of 0.05 is good enough for parts of this curve but not good enough for other parts.

At the bottom are three examples of actual curves with tick marks superimposed at a STEP size of 0.05. The tick marks are equally spaced in parameter space but notice how they are not equally spaced in real space.

Drawing a Bézier cubic – adaptive method

- adaptive subdivision
 - check if a straight line between P_0 and P_3 is an “adequate approximation” to the Bézier
 - if so: draw the straight line
 - if not: divide the Bézier into two halves, each a Bézier, and repeat for the two new Bezier
- need to specify some tolerance for when a straight line is an adequate approximation
 - when the Bézier lies within half a pixel width of the straight line along its entire length

Three different examples

98

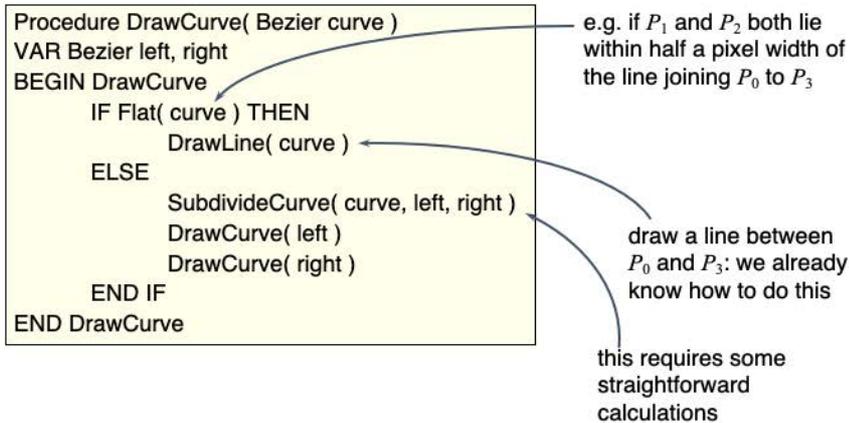
The illustrations on the right show you examples of three different Bézier curves and the straight line approximation to them.

The bottom curve is clearly almost a straight line already so you could just draw the straight line and you are done.

The middle curve is nearly a straight line, but it would depend on how big the pixels are as to whether it is close enough to a straight line for the straight line to be a good approximation.

The top curve is clearly not a straight line but, if the two end points were both in the same pixel, the straight line (i.e., one pixel!) would be a good approximation.

Drawing a Bézier cubic (continued)



99

This is a recursive function.

If the curve is sufficiently flat then we draw a line and we are finished.

Otherwise we split the curve into two parts and recurse on each of the two parts.

We already know how to draw a line.

We need to work out how to test for flatness and how to subdivide a Bézier curve into two new Bézier curves that, between them, exactly match the original Bézier curve.

Checking for flatness

Either solve this: $\overline{AB} \cdot \overline{CP(s)} = 0$

Or solve this: $s = \frac{\overline{AB} \cdot \overline{AC}}{|\overline{AB}|^2}$

Either solution gets you to this equation for s :

$$s = \frac{(x_B - x_A)(x_C - x_A) + (y_B - y_A)(y_C - y_A)}{(x_B - x_A)^2 + (y_B - y_A)^2}$$

we need to know this distance

so we need to know the location of this point

$$P(s) = (1 - s)A + sB$$

100

$P(s)$ is the closest point on the line to point C .

Either of the “solve this” equations relies on the dot product. See the Mathematics Workbook or *Fundamentals of Computer Graphics*, Sections 2.4.3 (pages 23–24).

The dot product of two vectors, $V=(x_1, y_1)$ and $W=(x_2, y_2)$ is calculated as

$$V \cdot W = x_1x_2 + y_1y_2$$

The dot product is also $V \cdot W = |V| |W| \cos\theta$ which, amongst other things, allows you a way to find $\cos\theta$ where θ is the angle between V and W .

For the first version, we say that the line connecting point $P(s)$ to C must be at right angles to line AB . Therefore $\cos\theta=0$.

For the second version, we know that $|AP| = |AC| \cos\theta$ from trigonometry.

We also know that the dot product, $AB \cdot AC = |AB||AC|\cos\theta$.

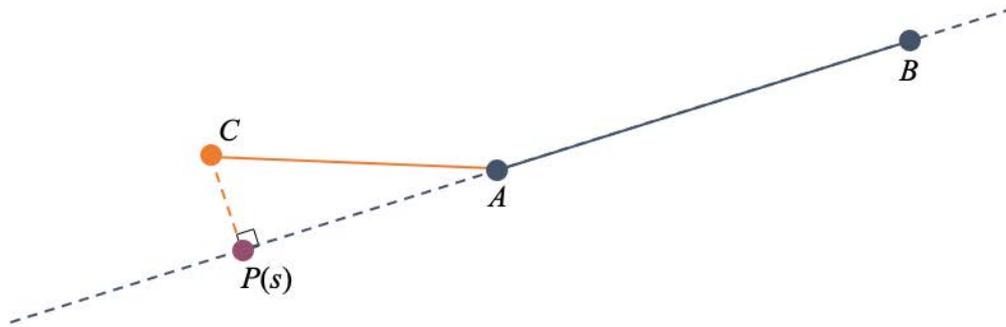
So to get $|AP|$ we need to divide $AB \cdot AC$ by $|AB|$.

Then s is the fraction along AB that P lies, so: $s = |AP| / |AB|$.

These two divisions by $|AB|$ appear in the denominator of the final equation as the square of the length of AB .

Special cases

- if $s < 0$ or $s > 1$ then the distance from point C to the line segment \overline{AB} is not the same as the distance from point C to the infinite line \overleftrightarrow{AB}
- in these cases the distance is $|AC|$ or $|BC|$ respectively



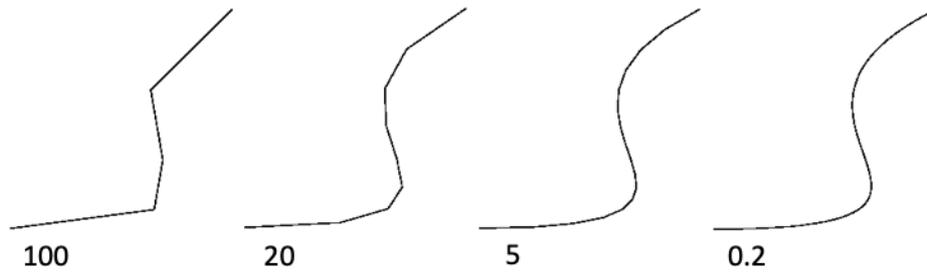
101

The subtlety here is that the maths on the previous slide related to the closest point, $P(s)$, to point C , that lies on the *infinite* line through A and B .

However, we actually want to know the closest point to the line segment that runs just between points A and B . This means that, if s lies between 0 and 1 then the maths on the previous slide is what we need, but if s is less than 0 or greater than 1 , then the closest point to C is one of the two end points: A if $s < 0$ and B if $s > 1$.

The effect of different tolerances

- this is the same Bezier curve drawn with four different tolerances



102

Subdividing a Bézier cubic

- a Bézier cubic can be easily subdivided into two smaller Bézier cubics

$$Q_0 = P_0$$

$$Q_1 = \frac{1}{2}P_0 + \frac{1}{2}P_1$$

$$Q_2 = \frac{1}{4}P_0 + \frac{1}{2}P_1 + \frac{1}{4}P_2$$

$$Q_3 = \frac{1}{8}P_0 + \frac{3}{8}P_1 + \frac{3}{8}P_2 + \frac{1}{8}P_3$$

$$R_0 = \frac{1}{8}P_0 + \frac{3}{8}P_1 + \frac{3}{8}P_2 + \frac{1}{8}P_3$$

$$R_1 = \frac{1}{4}P_1 + \frac{1}{2}P_2 + \frac{1}{4}P_3$$

$$R_2 = \frac{1}{2}P_2 + \frac{1}{2}P_3$$

$$R_3 = P_3$$

Exercise: prove that the Bézier cubic curves defined by Q_0, Q_1, Q_2, Q_3 and R_0, R_1, R_2, R_3 match the Bézier cubic curve defined by P_0, P_1, P_2, P_3 over the ranges $t \in [0, \frac{1}{2}]$ and $t \in [\frac{1}{2}, 1]$ respectively

There is a Processing program to demonstrate the algorithm: BezierSplitting.

Notice that the points can all be made by repeated averaging:

Define the following simple averages:

$$P_{01} = (P_0 + P_1) / 2$$

$$P_{12} = (P_1 + P_2) / 2$$

$$P_{23} = (P_2 + P_3) / 2$$

$$P_{012} = (P_{01} + P_{12}) / 2$$

$$P_{123} = (P_{12} + P_{23}) / 2$$

$$P_{0123} = (P_{012} + P_{123}) / 2$$

$$Q_0 = P_0 \quad R_0 = P_{0123}$$

$$Q_1 = P_{01} \quad R_1 = P_{123}$$

$$Q_2 = P_{012} \quad R_2 = P_{23}$$

$$Q_3 = P_{0123} \quad R_3 = P_3$$

You can do splits at arbitrary values of t by defining:

$$P_{01}(t) = (1-t)P_0 + (t)P_1$$

$$P_{012}(t) = (1-t)P_{01} + (t)P_{12}$$

etc.

Use the same t value for every split, then you get two smaller Beziers that together exactly match the bigger Bezier.

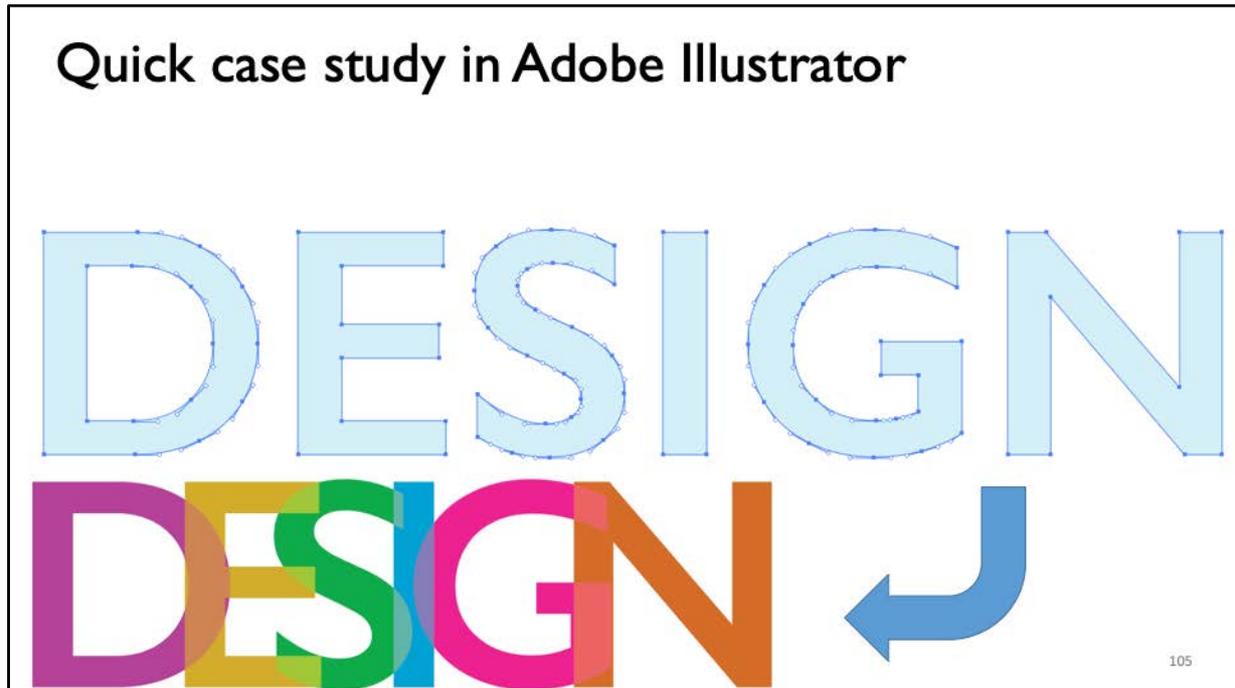
Cubic Béziers in typography

- Computer typefaces are defined using cubic Béziers



Filled boxes are end-points
Hollow circles are the other points
Straight lines have only end-points

104



In this case study, we have six letter defined by Bézier curves.

We want the visual effect to be that we have overlapped these curves and, where they overlap, they appear semi-transparent. A graphics designer will expect the software to allow them to do this. As software designers we need to work out how to achieve that visual effect.

How we actually achieve this, in this example, is to split the Bézier curves at the points where they overlap and to colour each individual section so that it achieves the desired look.

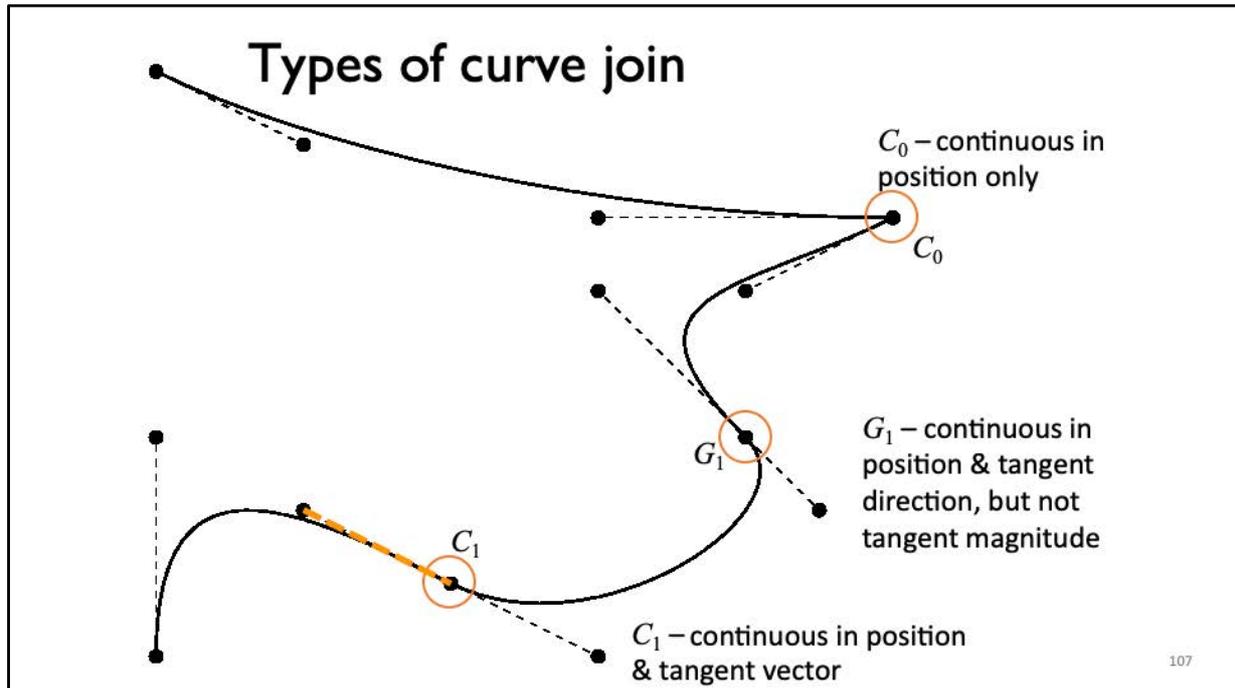
There are other ways to get the same look, which involve how we do colour shading with semi-transparent effects.



We split the shapes into overlapping and non-overlapping shapes. Each shape has a series of Bézier curves as its boundary. We colour each shape so that the pieces *appear* translucent.

On the left: the top row shows the letters overlapping. The middle row shows the overlapping regions coloured in black, after the shapes have been split. The bottom row shows the desired colouring.

On the right, notice that Adobe Illustrator has code that allows it to split the Bézier curves at arbitrary positions, creating new Bézier curves that exactly match the originals.



Types of continuity: *Fundamentals of Computer Graphics* Section 15.2.1 (pages 346–348).

Each Bézier curve is defined by its own four points. To make Bézier curves join up we make the end point of one curve match the start point of the next. To make them join up smoothly, we also need to worry about the positions of the control points adjacent to the shared end point.

Types of curve join

- each curve is smooth within itself
- joins at endpoints can be:
 - C_1 – continuous in both position and tangent vector
 - smooth join in a mathematical sense
 - G_1 – continuous in position, tangent vector in same direction
 - smooth join in a geometric sense
 - C_0 – continuous in position only
 - “corner”
 - discontinuous in position

C_n (mathematical continuity): continuous in all derivatives up to the n^{th} derivative

G_n (geometric continuity): each derivative up to the n^{th} has the same “direction” to its vector on either side of the join

$C_n \Rightarrow G_n$

108

What does this mean in practice?

For C_0 we make the end points in the same location.

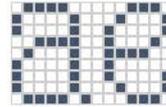
For G_1 the end point and the control points either side must all be in a straight line.

For C_1 the end point and the control points either side must all be in a straight line *and* the distance from the end point to the two control points must be the same.

Application of Bézier cubics: typography

- **typeface**: a family of letters designed to look good together
 - usually has upright (roman/regular), italic (oblique), bold and bold-italic members

abcd <i>efgh</i> ijkl <i>mnop</i> – Gill Sans	abcd <i>efgh</i> ijkl <i>mnop</i> – Times
abcd <i>efgh</i> ijkl <i>mnop</i> – Arial	abcd <i>efgh</i> ijkl <i>mnop</i> – Garamond
- **two forms of typeface used in computer graphics**
 - pre-rendered bitmaps
 - single resolution (don't scale well)
 - but fast to write to screen so used in the 1980s
 - outline definitions
 - defined by Bézier cubic curves (scale well)
 - need to render (fill) to write to screen
 - can include "hints" for how to render at low resolution

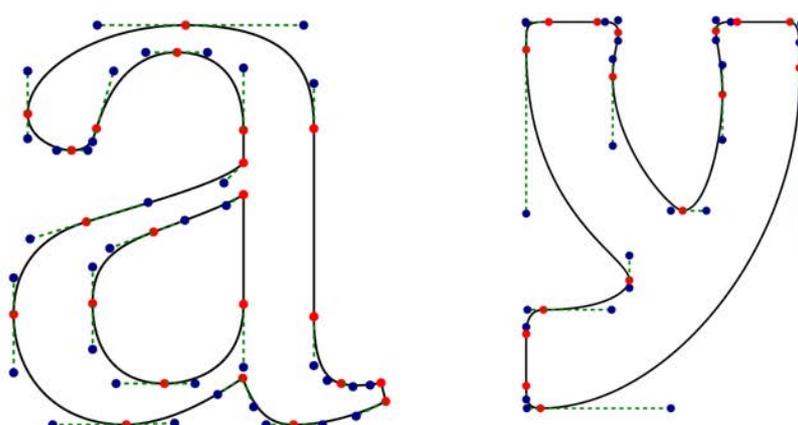


These notes are mainly set in Gill Sans, a lineale (sans-serif) typeface designed by Eric Gill for Monotype, 1928–30. The lowercase italic *p* is particularly interesting. The mathematics is mainly set in Times New Roman, a roman typeface commissioned by *The Times* newspaper in 1931, the design supervised by Stanley Morison.

109

Outline definitions are rendered to the screen by combining two algorithms that we already know: first the Bézier outline is converted to a series of straight lines (a polygon!) then we call the polygon filling algorithm on that polygonal outline.

Example typefaces that are Bézier-friendly

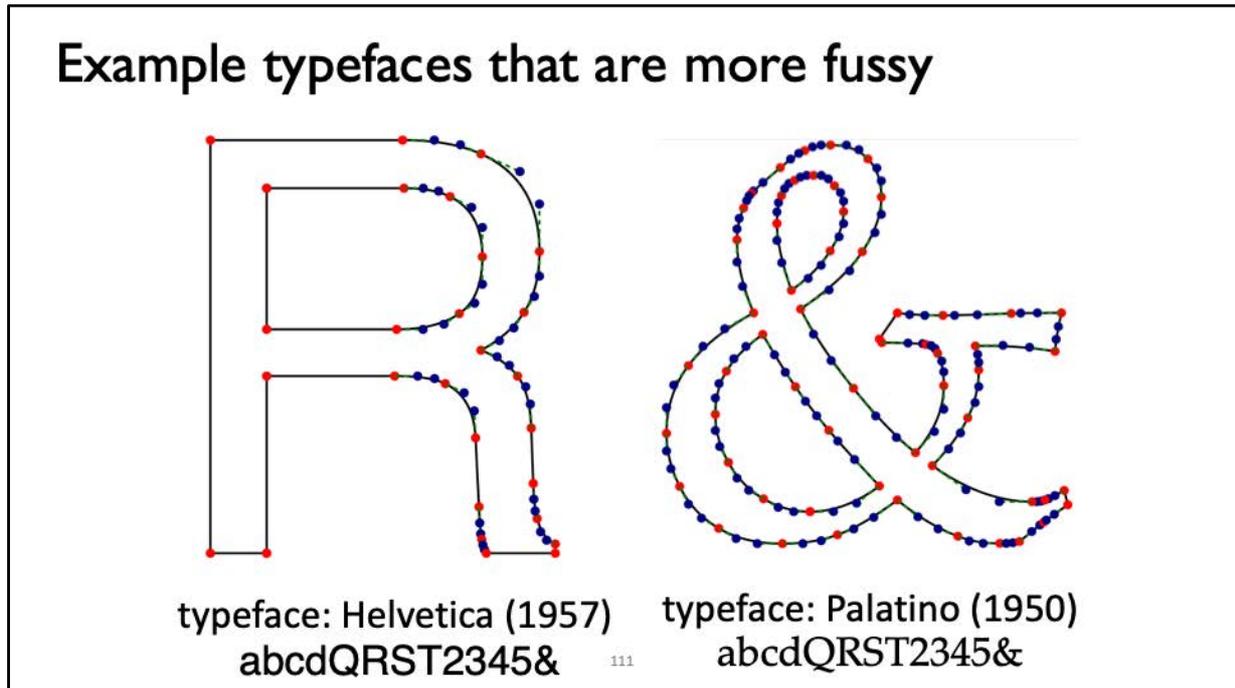


The image shows two examples of typefaces that are Bézier-friendly. On the left is the letter 'a' from the typeface Utopia (1989), designed as a Postscript typeface by Robert Slimbach at Adobe. On the right is the letter 'y' from the typeface Hobo (1910), which can be easily approximated by Bézier curves. Both letters are shown with their respective Bézier curves and control points, illustrating how they can be defined with a small number of control points.

typeface: Utopia (1989)
designed as a Postscript typeface by
Robert Slimbach at Adobe 110

typeface: Hobo (1910)
this typeface can be easily
approximated by Béziers

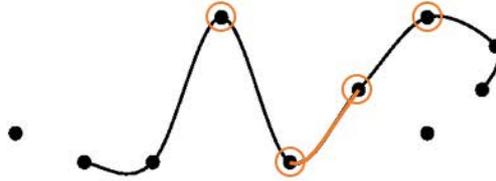
These typefaces need few control points to define their curves, because the designers are happy to use what a Bézier curve can deliver.



Compared to the previous slide, these letters need a lot more control points to get the shapes to exactly match the original, non-computerised, versions.

What if we have no tangent vectors?

- The Catmull-Rom cubic spline goes through all the data points.



- Each cubic piece depends on the four surrounding data points Why?
- At each data point the curve must depend solely on the three surrounding data points Why?
- the tangent at each point is defined as the direction from the preceding point to the succeeding point
 - tangent at P_1 is $\frac{1}{2}(P_2 - P_0)$, at P_2 is $\frac{1}{2}(P_3 - P_1)$

112

Now we come to the case where we want to have a curve that passes through all of the points.

A cubic spline curve is defined by four control points. Two of those points are the two ends of the curve. We need two further points and we use the next point in each direction.

As we the curve passes through a control point, the four points that define the curve change over so only three control points control what happens as the curve passes through that point: the point itself and the points either side. The curve position is determined by the point (that is, the curve goes through the point) and the tangent vector at that point is determined by the control points on either side.

Catmull-Rom cubic

- method for generating Bezier curves which match the Catmull-Rom model
 - calculate the appropriate Bezier control point locations from the given points
 - e.g. given points A, B, C, D , the Bezier control points are:

$$\begin{array}{ll} P_0=B & P_1=B+(C-A)/6 \\ P_3=C & P_2=C-(D-B)/6 \end{array}$$

- The Catmull-Rom cubic *interpolates* its controlling data points
 - good for control of movement in animation
 - not so good for industrial design because moving a single point modifies the surrounding four curve segments
 - compare with Bezier where moving a single point modifies just the two segments connected to that point

113

The formulas for P_0 and P_3 are obvious: the Bézier end points are the same as the Catmull-Rom end points.

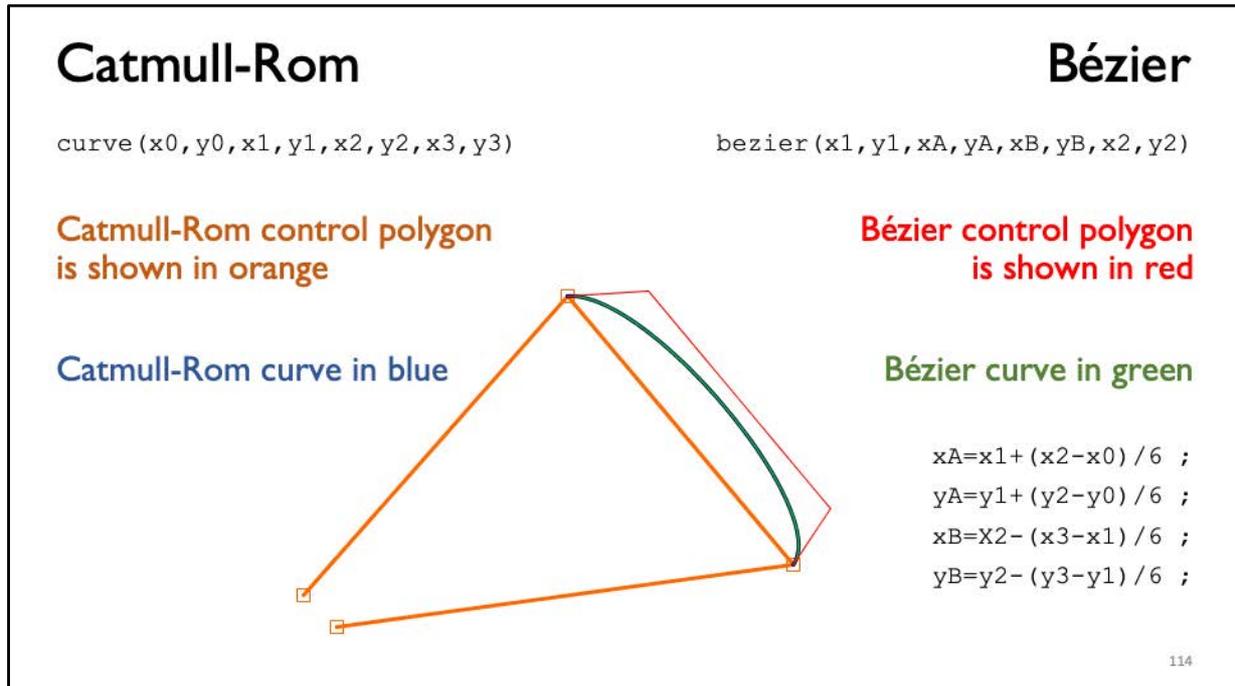
The formulas for P_1 and P_2 need a bit of explanation:

A Bézier curve's tangent vector at P_0 is: $3(P_1-P_0)$.

A Catmull-Rom's tangent vector at P_0 is: $(C-A)/2$ (see previous slide)

Remember that $P_0=B$, so:

$$\begin{aligned} 3(P_1-P_0) &= (C-A)/2 \\ \Rightarrow 3(P_1-B) &= (C-A)/2 \\ \Rightarrow P_1-B &= (C-A)/6 \\ \Rightarrow P_1 &= B+(C-A)/6 \end{aligned}$$

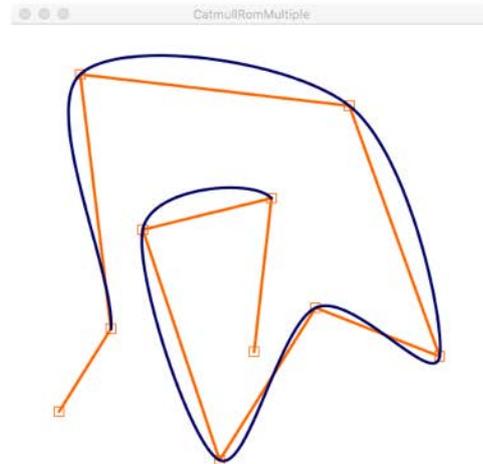


The Catmull-Rom curve and the Bézier curve in this example are identical. What this slide demonstrates is how to generate a Bézier curve equivalent to the Catmull-Rom curve. Notice how far away the outer Catmull-Rom control points are compared to how close to the curve the Bézier inner control points are.

Catmull-Rom cubics in Processing

```
strokeWeight( 3 ) ;  
stroke(0,0,102); // blue Catmull-Rom  
beginShape() ;  
for( int i=0 ; i<numPoints ; i++ ) {  
  curveVertex( pts[i].x, pts[i].y ) ;  
}  
endShape() ;
```

- Notice that the curve starts at the second point and ends at the second-to-last point



115

The image and code are from the example code `CatmullRomMultiple`. The orange boxes are the points that define the curve. The orange lines are connected in the order in which the points are used.

Béziers: Calculating position and tangent vector

- Position runs from $t=0$ to $t=1$, with

$$P(0)=P_0$$

$$P(1)=P_3$$

- Tangent vector found by taking the first derivative of $P(t)$

$$P'(0)=3(P_1-P_0)$$

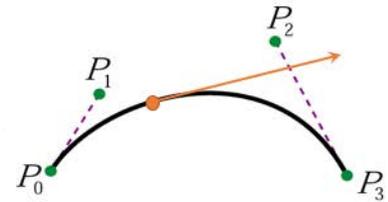
$$P'(1)=3(P_3-P_2)$$

$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1$$

$$+ 3(1-t)t^2 P_2 + t^3 P_3$$

$$P'(t) = -3(1-t)^2 P_0 + [3(1-t)^2 - 6(1-t)t] P_1$$

$$+ [6(1-t)t - 3t^2] P_2 + 3t^2 P_3$$



116

The tangent vector is a vector that shows the direction in which the curve is moving at that point. If you drove a car along the curve, the tangent vector would be the direction that the car is moving as it drives along the curve.

Reminder: what does P represent?

- Position runs from $t=0$ to $t=1$, with
 $P(0)=P_0$
 $P(1)=P_3$
- P is short-hand for the point (x,y)

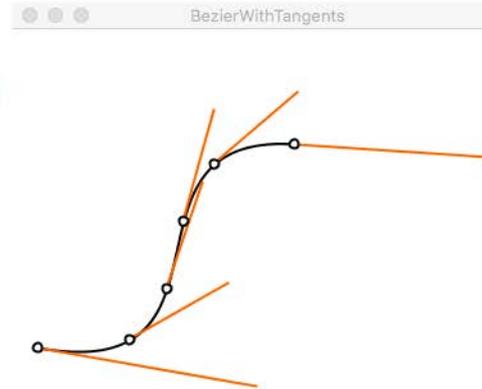
$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3 \quad \left\{ \begin{array}{l} x(t) = (1-t)^3 x_0 + 3(1-t)^2 t x_1 \\ \quad \quad \quad + 3(1-t)t^2 x_2 + t^3 x_3 \\ y(t) = (1-t)^3 y_0 + 3(1-t)^2 t y_1 \\ \quad \quad \quad + 3(1-t)t^2 y_2 + t^3 y_3 \end{array} \right.$$

117

Remember that we are using mathematical shorthand. The vector on the left encapsulates, in a single equation, the two equations on the right.

Computing position and tangent in Processing

```
size(400,300);
background(255);
strokeWeight(2);
noFill();
// draw the curve in black
bezier(20, 250, 190, 280, 70, 80, 220, 90);
fill(255);
for (float t = 0.0 ; t <= 1.0; t+=0.2) {
  // get the location of the point
  float x = bezierPoint(20,190,70,220, t);
  float y = bezierPoint(250,280,80,90, t);
  // get the tangent points
  float tx = bezierTangent(20,190,70,220, t);
  float ty = bezierTangent(250,280,80,90, t);
  // draw the tangent line in orange
  stroke(255, 102, 0);
  line(x, y, x+tx/3, y+ty/3);
  // draw the point as a circle
  stroke(0);
  ellipse(x, y, 7, 7);
}
```



118

The curve is shown in black.

We draw the tangent vector, in orange, at the locations of the little circles on the curve. We have positioned the circles at a STEP size of 0.2.

There are similar functions for Catmull-Rom points and tangent vectors, which are:

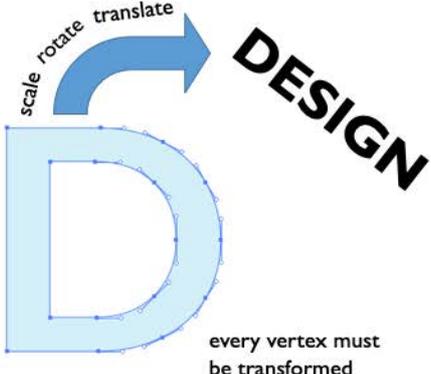
```
curvePoint()
curveTangent()
```

2D transformations

- scale 
- rotate 
- translate 
- (shear) 

• why?

- it is extremely useful to be able to transform predefined objects to an arbitrary location, orientation, and size



every vertex must be transformed

119

The top three translations are the ones that you will use regularly.

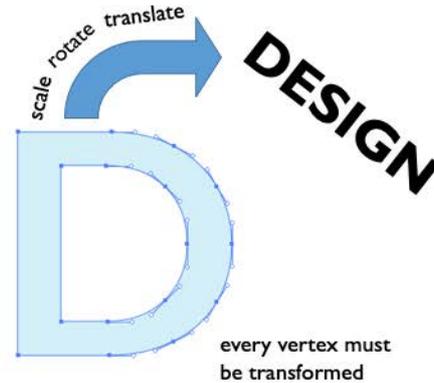
Rotate and translate do not change the size or aspect ratio of the object. They are therefore called rigid-body transformations because the object does not change, it just turns or moves.

Scale is not a rigid-body transformation because the object gets bigger. However, it does not change its aspect ratio or the internal angles of the object.

Shear is not a rigid-body transformation because the object is distorted: its aspect ratio changes and all the internal angles change. Shear is not used much in practice but, because it is easy to implement, you usually find it implemented alongside the three more useful transformations.

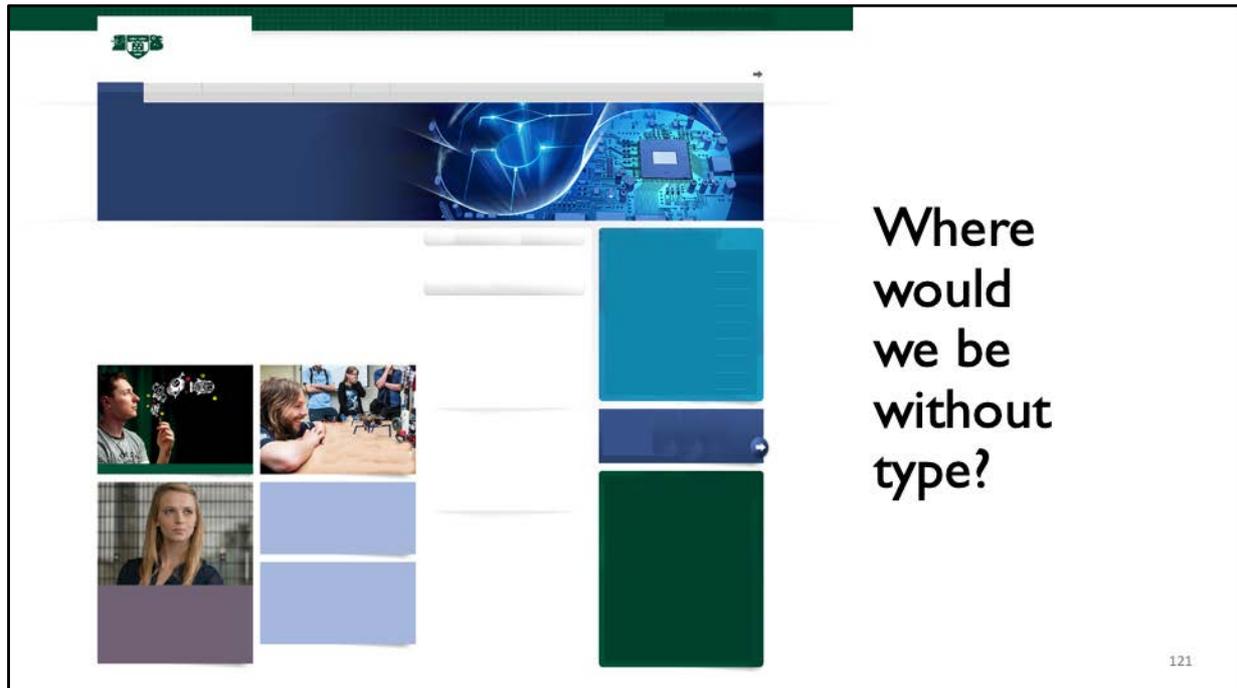
Transformations allow us to reuse objects

- it is extremely useful to be able to transform predefined objects to an arbitrary location, orientation, and size
- for example, all typefaces are defined with letters of a standard height in a standard position



120

We can specify an object once (such as the letter 'e' on this slide) and then scale, rotate and translate it to place it in all the positions where it is needed.



The slide shows a webpage without any text on it. It has some nice visuals but the text is what makes it useful.

Transformations in Processing

- `scale(factor)`



- `rotate(angle)`



- Think of the transformation as moving the entire universe

- `translate(tx, ty)`

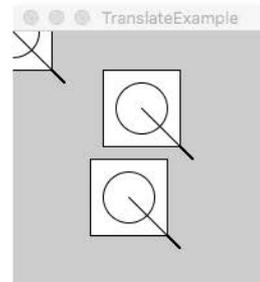


122

Getting Started with Processing Chapter 6 (pages 75–87) covers this in detail. It will give you an alternative explanation of transformations.

Translation

```
void setup(){
  size(200,200);
}
void drawThing(){
  rect(-30,-30,60,60);
  ellipse( 0,0,40,40);
  line(0,0,40,40);
}
void draw(){
  drawThing();
  translate(100,60);
  drawThing();
  translate(-10,70);
  drawThing();
}
```



123

Three versions of the Thing are drawn.

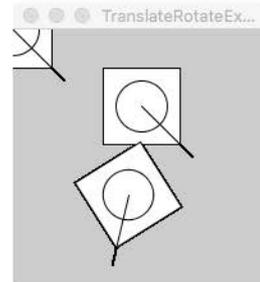
The first is drawn in the default position, with the origin, (0,0), at the top left corner of the screen.

The second is drawn with the origin translated to (100,60).

The third is drawn with the origin translated to $(100-10,60+70) = (90,130)$.

Rotation

```
void setup(){
  size(200,200);
}
void drawThing(){
  rect(-30,-30,60,60);
  ellipse( 0,0,40,40);
  line(0,0,40,40);
}
void draw(){
  drawThing();
  translate(100,60);
  drawThing();
  translate(-10,70);
  rotate( 1.0 ); // one radian
  drawThing();
}
```

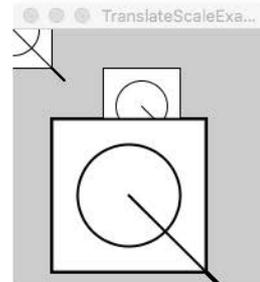


124

The third version is now also rotated by 1.0 radian (57.3°).

Scale

```
void setup(){
  size(200,200);
}
void drawThing(){
  rect(-30,-30,60,60);
  ellipse( 0,0,40,40);
  line(0,0,40,40);
}
void draw(){
  drawThing();
  translate(100,60);
  drawThing();
  translate(-10,70);
  scale( 2.0 ); // double size
  drawThing();
}
```

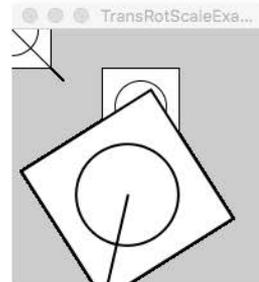


125

The third version is now scaled up by a factor of 2.0. Notice that the stroke thicknesses get scaled up along with everything else.

Transformations concatenate

```
void setup(){
  size(200,200);
}
void drawThing(){
  rect(-30,-30,60,60);
  ellipse( 0,0,40,40);
  line(0,0,40,40);
}
void draw(){
  drawThing();
  translate(100,60);
  drawThing();
  translate(-10,70);
  rotate( 1.0 ); // one radian
  scale( 2.0 ); // double size
  drawThing();
}
```



126

Finally, we combine all three types of transformation: translation, rotation and scaling.

Processing stores an internal *current transformation matrix*. Whenever you call a transformation function, it concatenates the new transformation onto the stored transformation. In this case it does two translations, one rotation, and one scaling, in that order.

Processing resets its internal *current transformation matrix* every time it starts the `draw()` function.

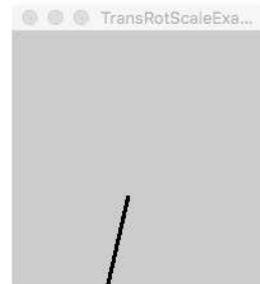
Let's look at this from a point's point of view

```
void setup(){
  size(200,200);
}
void drawThing(){

  line(0,0,40,40);
}
void draw(){

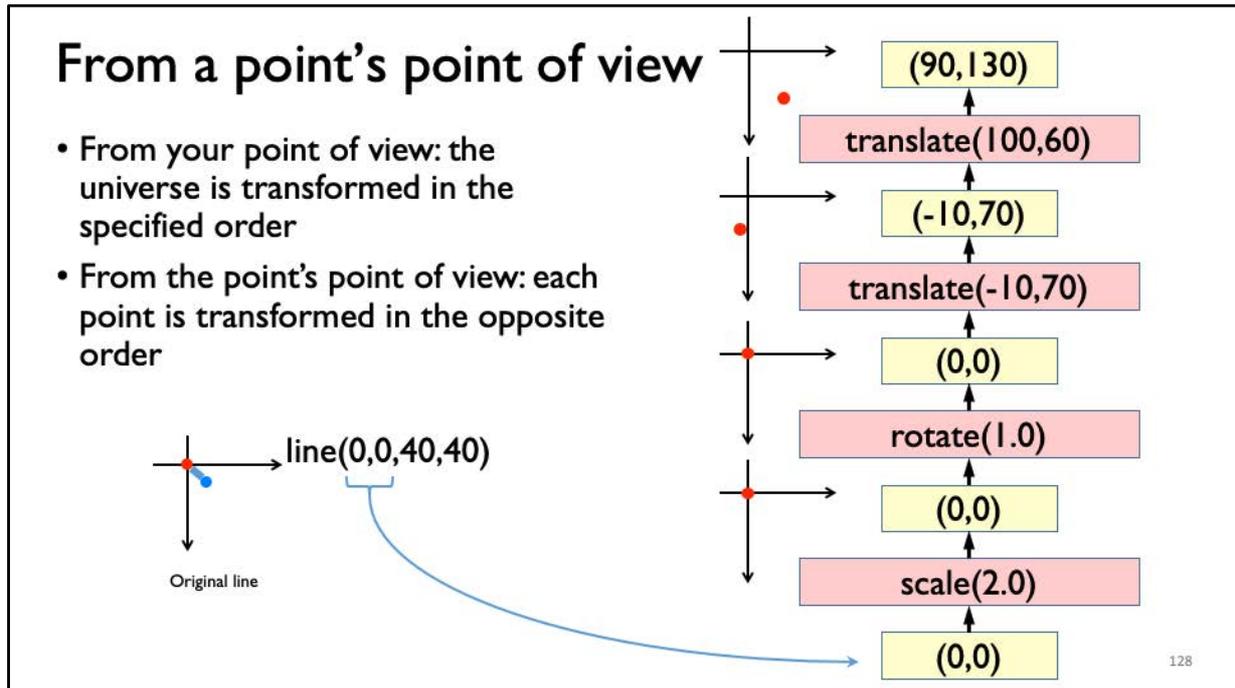
  translate(100,60);

  translate(-10,70);
  rotate( 1.0 ); // one radian
  scale( 2.0 ); // double size
  drawThing();
}
```



127

We have removed everything except the single line from the Thing. We have chosen to draw only one Thing (which was the third Thing on the previous slide).



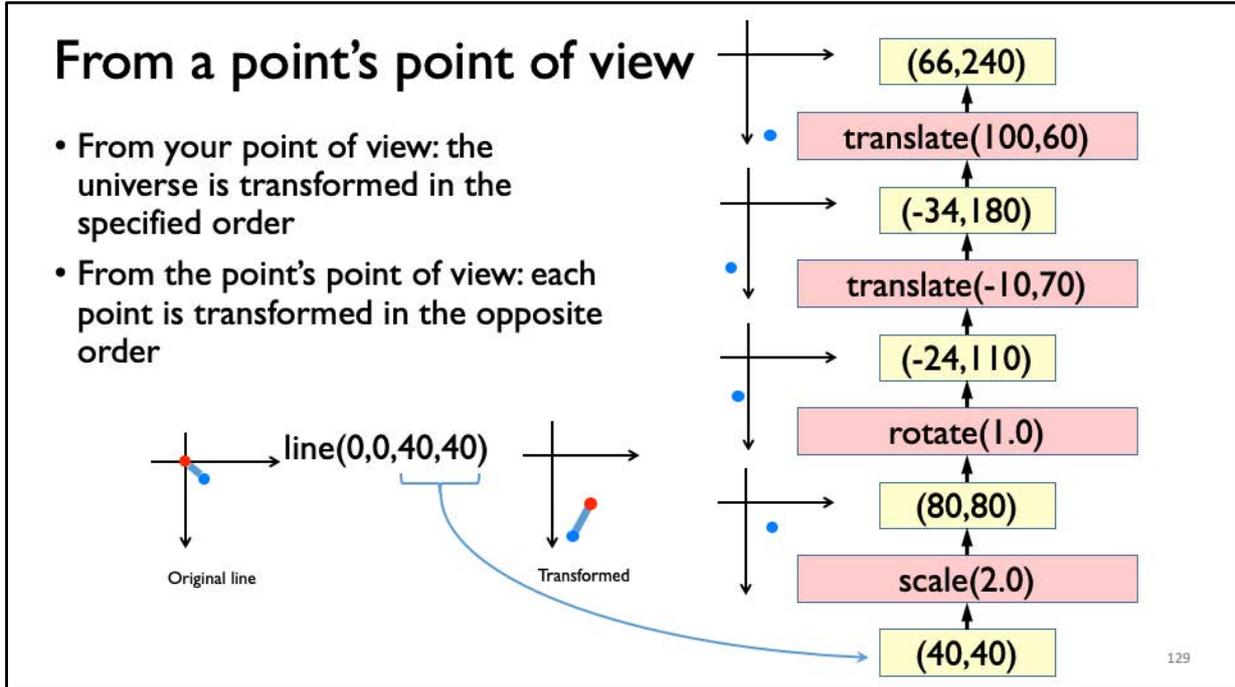
The point's point of view is important for understanding the matrix representation of transformations, coming up in a couple of slides.

From your (the programmer's) point of view, you think of the universe as being moved so that things get drawn where you want them.

But from Processing's point of view, the pixels in the window have fixed locations with (0,0) fixed at the top-left corner. It cannot move the universe. Its universe is fixed. Processing therefore has to move the points, by putting them through the transformations in the opposite order to get them to where you want them to be.

These are two complementary ways of looking at how transformations work.

In this case, the point (0,0) is not moved by either the scale or the rotation, because both of those transformations are done with the origin as a fixed point, so those two transformations leave (0,0) at (0,0).



In this second example, the point (40,40) is first scaled about the origin by a factor of 2, to (80,80), then rotated about the origin by 1.0 radians, to reach point (-24,110) [rounded to the nearest integer]. The two translations then move the point, so that it ends up at (66,240).

Basic 2D transformations

<ul style="list-style-type: none"> • scale <ul style="list-style-type: none"> • about origin • by factor m 	$x' = mx$ $y' = my$	
<ul style="list-style-type: none"> • rotate <ul style="list-style-type: none"> • about origin • by angle θ 	$x' = x \cos \theta - y \sin \theta$ $y' = x \sin \theta + y \cos \theta$	
<ul style="list-style-type: none"> • translate <ul style="list-style-type: none"> • along vector (x_o, y_o) 	$x' = x + x_o$ $y' = y + y_o$	
<ul style="list-style-type: none"> • shear <ul style="list-style-type: none"> • parallel to x axis • by factor a 	$x' = x + ay$ $y' = y$	

130

There are only four basic 2D transformations: scale, rotate, translate and shear. The fourth (shear) is not particularly useful in practice. However, the other three are needed for practically everything.

Matrix representation

Rotation...

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta\end{aligned}$$

• rotate

• about origin, angle θ

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The general case...

$$\begin{aligned}x' &= ax + by \\y' &= cx + dy\end{aligned}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

131

See *Fundamentals of Computer Graphics*, Chapter 6 (pages 111–140), which cover transformations in detail. At this stage, you can skip over Section 6.2 (3D transformations).

If you do not know, or cannot remember, how to handle matrix multiplication, see *Fundamentals of Computer Graphics*, Section 5.2 (pages 93–98) and the CGRA151 Mathematics Workbook.

Basic Matrix Operation

• Addition of Matrices

If A and B are both $m \times n$ matrices then the **sum** of A and B , denoted $A + B$, is a matrix obtained by adding **corresponding elements** of A and B .

$$A = \begin{bmatrix} 1 & -2 & 2 \\ 0 & -1 & 3 \end{bmatrix} \quad B = \begin{bmatrix} -3 & 0 & 4 \\ 2 & 1 & -4 \end{bmatrix}$$

$$A + B = \begin{bmatrix} -2 & -2 & 6 \\ 2 & 0 & -1 \end{bmatrix}$$

• Scalar Multiplication of Matrices

If A is an $m \times n$ matrix and s is a scalar, then we let kA denote the matrix obtained by multiplying every element of A by k . This procedure is called **scalar multiplication**.

$$A = \begin{bmatrix} 1 & -2 & 2 \\ 0 & -1 & 3 \end{bmatrix}$$

$$3A = \begin{bmatrix} 3(1) & 3(-2) & 3(2) \\ 3(0) & 3(-1) & 3(3) \end{bmatrix} = \begin{bmatrix} 3 & -6 & 6 \\ 0 & -3 & 9 \end{bmatrix}$$

132

The CGRA151 Mathematics Workbook covers this in detail.

There are links to useful online resources on the course website.

Basic Matrix Operation

• Multiplication of Matrices

The multiplication of matrices is easier shown than put into words. You multiply the rows of the first matrix with the columns of the second adding products

$$A = \begin{bmatrix} 3 & -2 & 1 \\ 0 & 4 & -1 \end{bmatrix} \quad B = \begin{bmatrix} 2 & 4 \\ -1 & 3 \\ -3 & 1 \end{bmatrix}$$

$$AB = \begin{bmatrix} 5 & 7 \\ -1 & 11 \end{bmatrix}$$

$$3(2) + (-2)(-1) + (1)(-3) = 5$$

Notice the sizes of A and B and the size of the product AB .

$$\begin{array}{c} m \times n \quad \quad n \times p \\ \uparrow \quad \quad \uparrow \\ \text{size of product} \end{array}$$

133

One way to think of this is to think of the row of matrix A jumping and diving into the column of matrix B . You multiply corresponding elements and add up the results.

Matrix representation of transformations

- scale

- about origin, factor m

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- do nothing

- identity

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- rotate

- about origin, angle θ

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- shear

- parallel to x axis, factor a

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

134

Compare with the equations four slides earlier to see how these matrices are constructed.

For example, for the shear, we had:

$$x' = x + ay$$

$$y' = y$$

Homogeneous 2D co-ordinates

- But translations cannot be represented using simple 2×2 matrix multiplication on 2D vectors

$$\begin{aligned}x' &= x + x_o \\ y' &= y + y_o\end{aligned}$$

- so we switch to homogeneous co-ordinates

$$(x, y, w) = \left(\frac{x}{w}, \frac{y}{w} \right)$$

- an infinite number of homogeneous co-ordinates map to every 2D point
- $w=0$ represents a point at infinity
- we take the inverse transform to be: $(x, y) \rightarrow (x, y, 1)$

135

With a 2×2 matrix, you know (from the previous slide) that $x' = ax + by$. You cannot pick two constants, a and b , that will allow you to say $x' = x + x_o$.

We fix this by moving to a 3×3 matrix, with a third dimension to our vectors, and we set that dimension equal to 1. We then have $x' = ax + by + e \times 1$ and we can set $a=1$, $b=0$, $e=x_o$ to get $x' = x + x_o$.

Matrices in homogeneous co-ordinates

- scale

- about origin, factor m

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- do nothing

- identity

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- rotate

- about origin, angle θ

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- shear

- parallel to x axis, factor a

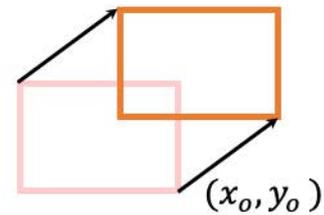
$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

136

These are the 2×2 matrices from three slides ago, with an extra row and column added. You can see that the extra rows and columns have the values 0,0,1.

Translation by matrix algebra

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$



In homogeneous coordinates

$$x' = x + wx_o \quad y' = y + wy_o \quad w' = w$$

In conventional coordinates

$$\frac{x'}{w'} = \frac{x}{w} + x_o \quad \frac{y'}{w'} = \frac{y}{w} + y_o$$

137

This demonstrates that the w coordinate divides out so that it does not matter what value w has, the translation is always by (x_o, y_o)

Concatenating transformations

- it is often necessary to perform multiple transformations on the same object
- concatenating transformations can be done by multiplying their matrices
- we can then apply the matrix product to the points, rather than the individual matrices

e.g. a shear followed by a scaling:

$$\begin{aligned}
 \begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} &= \begin{matrix} \text{scale} \\ \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{matrix} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} & \quad \begin{matrix} \text{shear} \\ \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{matrix} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} \\
 \begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} &= \begin{matrix} \text{scale} & \text{shear} \\ \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{matrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} &= \begin{matrix} \text{both} \\ \begin{bmatrix} m & ma & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{matrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}
 \end{aligned}$$

138

The great advantage of using matrices is that we can multiply together as many transformations as we like and the resulting single matrix applies all of those transformations to any point in a single matrix multiplication operation.

Matrix multiplication is covered in the CGRA151 Mathematics Workbook.

Concatenation is not commutative

- be careful of the order in which you concatenate transformations

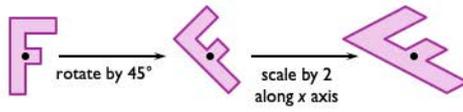
scale

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

rotate

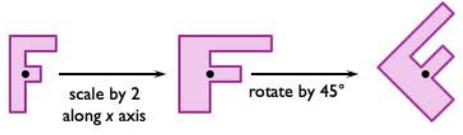
$$\begin{bmatrix} 0.71 & -0.71 & 0 \\ 0.71 & 0.71 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

rotate then scale



$$\begin{bmatrix} 1.42 & -1.42 & 0 \\ 0.71 & 0.71 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

scale then rotate



$$\begin{bmatrix} 1.42 & -0.71 & 0 \\ 1.42 & 0.71 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

139

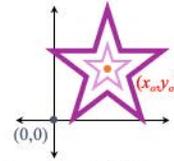
Matrices are not like normal numbers: order matters when doing multiplication of matrices.

The examples at left show you intuitively that the order matters. The matrices in the centre show the actual mathematical result of multiplying the two matrices in different orders, so that you can see that they are different.

Scaling about an arbitrary point

- scale by a factor m about point (x_o, y_o)

- 1 translate point (x_o, y_o) to the origin
- 2 scale by a factor m about the origin
- 3 translate the origin to (x_o, y_o)



$$\textcircled{1} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \quad \textcircled{2} \begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} \quad \textcircled{3} \begin{bmatrix} x''' \\ y''' \\ w''' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix}$$

$$\begin{bmatrix} x''' \\ y''' \\ w''' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$\textcircled{3}$
 $\textcircled{2}$
 $\textcircled{1}$

Exercise: show how to perform rotation about an arbitrary point

140

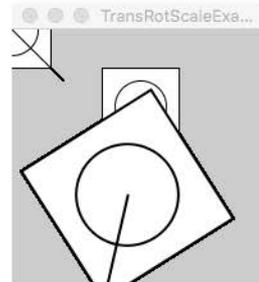
The star is defined by ten points. We multiply the three matrices together, to get a single matrix M . We then multiply M by each of the ten points, so using ten matrix multiplications to do the transformation, rather than the thirty we would need if we did each transformation separately.

So, what does the final matrix, M , look like? We don't particularly care, because the computer does all the manipulation internally and stores that single matrix. If you are interested you can multiply it out for yourself:

$$M = \begin{bmatrix} m & 0 & x_o(1-m) \\ 0 & m & y_o(1-m) \\ 0 & 0 & 1 \end{bmatrix}$$

Processing's *current transformation matrix*

```
void setup(){
  size(200,200);
}
void drawThing(){
  rect(-30,-30,60,60);
  ellipse( 0,0,40,40);
  line(0,0,40,40);
}
void draw(){
  drawThing();
  translate(100,60);
  drawThing();
  translate(-10,70);
  rotate( 1.0 ); // one radian
  scale( 2.0 ); // double size
  drawThing();
}
```



141

Processing's *current transformation matrix*

```
void setup(){
  size(200,200);
}
void drawThing(){
  rect(-30,-30,60,60);
  ellipse( 0,0,40,40);
  line(0,0,40,40);
}
void draw(){
  drawThing();
  translate(100,60);
  drawThing();
  translate(-10,70);
  rotate( 1.0 ); // one radian
  scale( 2.0 ); // double size
  drawThing();
}
```

- Processing internally stores a *current transformation matrix*
- Every point on every drawn object goes through the *current transformation matrix*
- When you call `translate()`, `scale()` or `rotate()`, the matrix for that transformation is multiplied into the *current transformation matrix*

142

Processing resets the *current transformation matrix* every time it starts the `draw()` function. So, when the `draw()` function starts, the initial value of *current transformation matrix* is an identity matrix.

Matrices for our example transformations

$$\text{translate}(100,60) \quad T_1 = \begin{bmatrix} 1 & 0 & 100 \\ 0 & 1 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(-10,70) \quad T_2 = \begin{bmatrix} 1 & 0 & -10 \\ 0 & 1 & 70 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{rotate}(1.0) \quad R = \begin{bmatrix} 0.54 & -0.84 & 0 \\ 0.84 & 0.54 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{scale}(2.0) \quad S = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

143

This slide is the start of a worked example showing you how each matrix is multiplied into the *current transformation matrix*.

Updating the *current transformation matrix*

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(100,60) \quad T_1 = \begin{bmatrix} 1 & 0 & 100 \\ 0 & 1 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M \leftarrow M \times T_1 = \begin{bmatrix} 1 & 0 & 100 \\ 0 & 1 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(-10,70) \quad T_2 = \begin{bmatrix} 1 & 0 & -10 \\ 0 & 1 & 70 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{rotate}(1.0) \quad R = \begin{bmatrix} 0.54 & -0.84 & 0 \\ 0.84 & 0.54 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{scale}(2.0) \quad S = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

144

Updating the *current transformation matrix*

$$M = \begin{bmatrix} 1 & 0 & 100 \\ 0 & 1 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(100,60) \quad T_1 = \begin{bmatrix} 1 & 0 & 100 \\ 0 & 1 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M \leftarrow M \times T_2 = \begin{bmatrix} 1 & 0 & 90 \\ 0 & 1 & 130 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(-10,70) \quad T_2 = \begin{bmatrix} 1 & 0 & -10 \\ 0 & 1 & 70 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{rotate}(1.0) \quad R = \begin{bmatrix} 0.54 & -0.84 & 0 \\ 0.84 & 0.54 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{scale}(2.0) \quad S = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

145

Updating the *current transformation matrix*

$$M = \begin{bmatrix} 1 & 0 & 90 \\ 0 & 1 & 130 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(100,60) \quad T_1 = \begin{bmatrix} 1 & 0 & 100 \\ 0 & 1 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M \leftarrow M \times R = \begin{bmatrix} 0.54 & -0.84 & 90 \\ 0.84 & 0.54 & 130 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(-10,70) \quad T_2 = \begin{bmatrix} 1 & 0 & -10 \\ 0 & 1 & 70 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{rotate}(1.0) \quad R = \begin{bmatrix} 0.54 & -0.84 & 0 \\ 0.84 & 0.54 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{scale}(2.0) \quad S = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

146

Updating the *current transformation matrix*

$$M = \begin{bmatrix} 0.54 & -0.84 & 90 \\ 0.84 & 0.54 & 130 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(100,60) \quad T_1 = \begin{bmatrix} 1 & 0 & 100 \\ 0 & 1 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M \leftarrow M \times S = \begin{bmatrix} 1.08 & -1.68 & 90 \\ 1.68 & 1.08 & 130 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(-10,70) \quad T_2 = \begin{bmatrix} 1 & 0 & -10 \\ 0 & 1 & 70 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{rotate}(1.0) \quad R = \begin{bmatrix} 0.54 & -0.84 & 0 \\ 0.84 & 0.54 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{scale}(2.0) \quad S = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

147

Updating the *current transformation matrix*

$$M = \begin{bmatrix} 1.08 & -1.68 & 90 \\ 1.68 & 1.08 & 130 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(100,60) \quad T_1 = \begin{bmatrix} 1 & 0 & 100 \\ 0 & 1 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{translate}(-10,70) \quad T_2 = \begin{bmatrix} 1 & 0 & -10 \\ 0 & 1 & 70 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{rotate}(1.0) \quad R = \begin{bmatrix} 0.54 & -0.84 & 0 \\ 0.84 & 0.54 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{scale}(2.0) \quad S = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

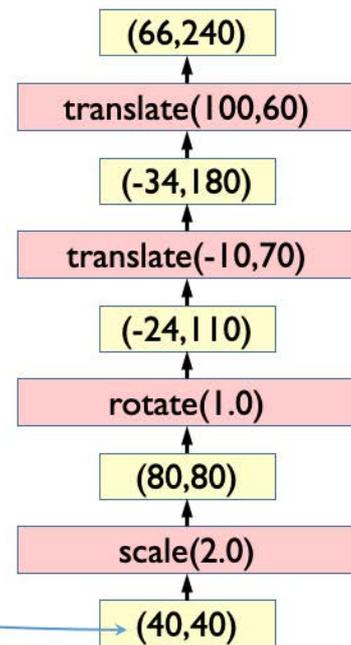
148

From a point's point of view

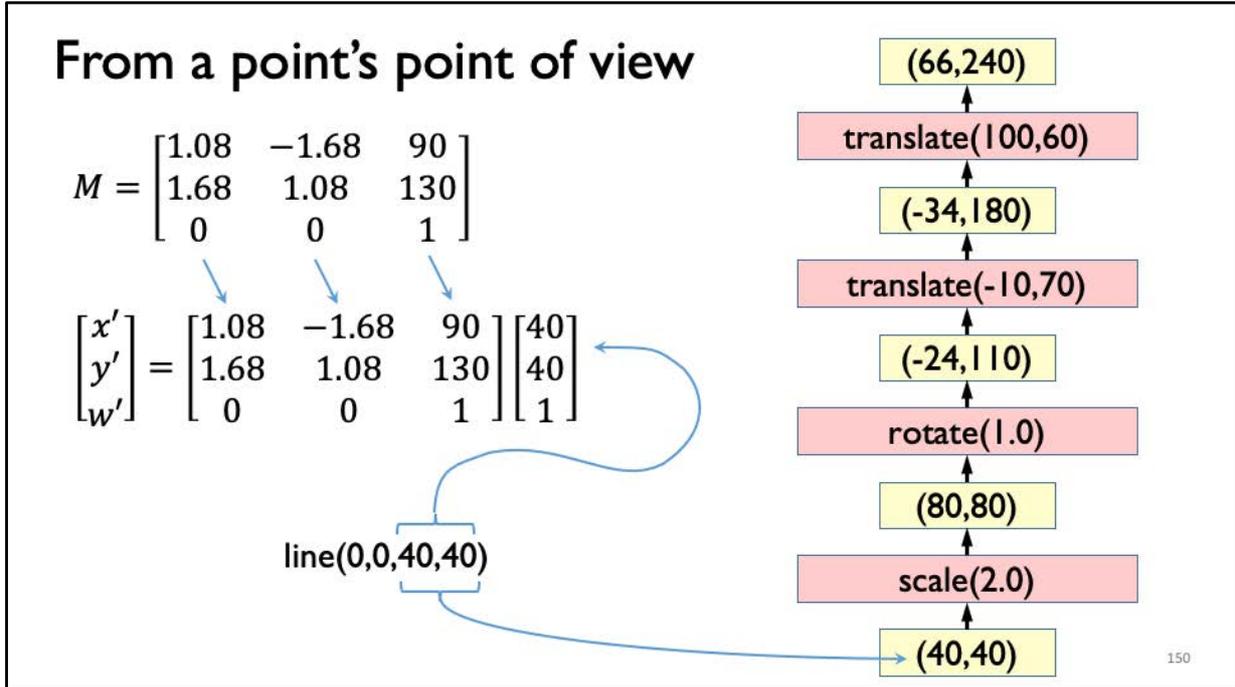
$$M = \begin{bmatrix} 1.08 & -1.68 & 90 \\ 1.68 & 1.08 & 130 \\ 0 & 0 & 1 \end{bmatrix}$$

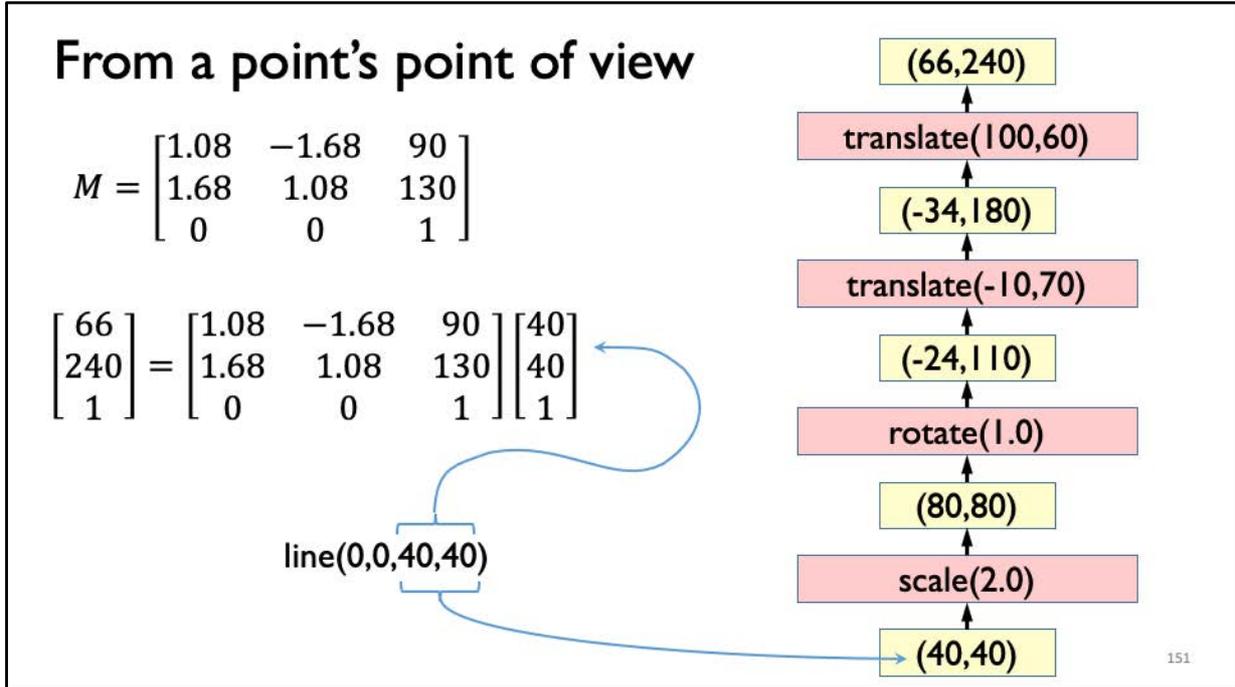
$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

line(0,0,40,40)



149



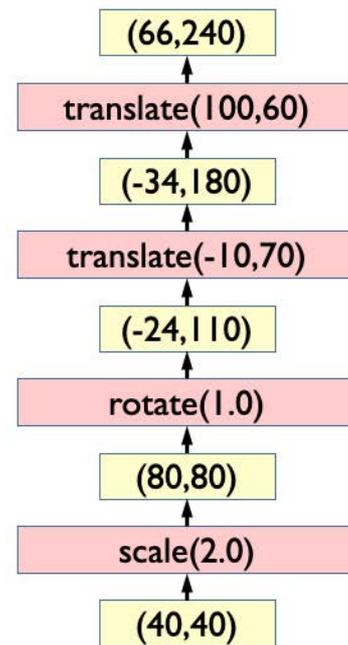


From a point's point of view

$$M = \begin{bmatrix} 1.08 & -1.68 & 90 \\ 1.68 & 1.08 & 130 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = T_1 T_2 R S \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



152

From a point's point of view

$$M = \begin{bmatrix} 1.08 & -1.68 & 90 \\ 1.68 & 1.08 & 130 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = T_1 T_2 R S \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The diagram illustrates a transformation process. At the bottom, a yellow box contains the coordinates (40,40). An upward-pointing arrow leads to a pink rectangular box labeled "current transformation matrix". From the top of this pink box, another upward-pointing arrow leads to a yellow box containing the coordinates (66,240). This visualizes the application of the transformation matrix to the initial point.

153

If you have not done so already, now would be a good time to skim through *Getting Started with Processing*, Chapter 6 (pages 75–87).

pushMatrix() and popMatrix()

```
1 float angle=0.0;
2 void setup() {
3   size(300, 200);
4 }
5 void draw() {
6   angle += mouseX/1000.0;
7   translate(100, 100);
8   pushMatrix();
9   rotate(angle);
10  rect(-20, -20, 40, 40);
11  popMatrix();
12  translate(100, 0);
13  pushMatrix();
14  rotate(angle);
15  rect(-20, -20, 40, 40);
16  popMatrix();
17 }
```

- Used to store the current matrix and retrieve it later

Stores the current matrix

Retrieves the stored matrix

154

`pushMatrix()` stores the current transformation matrix on a stack

`popMatrix()` removes the matrix that is at the top of the stack and makes it the current transformation matrix

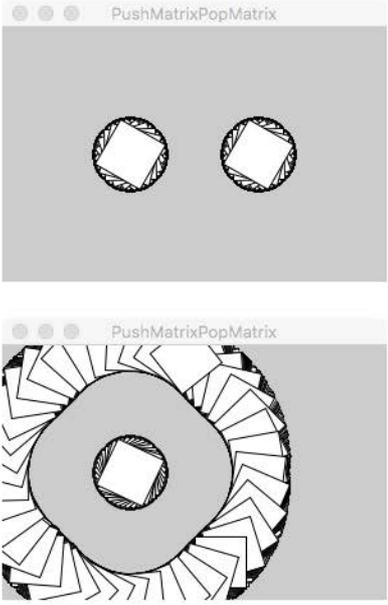
Think of a stack like a stack of plates on a spring-loaded platform. Push pushes a plate onto the top of the stack (pushing all the other plates down). Pop pops the top plate off the stack.

pushMatrix() and popMatrix()

```

PushMatrixPopMatrix
1 float angle=0.0;
2 void setup() {
3   size(300, 200);
4 }
5 void draw() {
6   angle += mouseX/1000.0;
7   translate(100, 100);
8
9   rotate(angle);
10  rect(-20, -20, 40, 40);
11
12  translate(100, 0);
13
14  rotate(angle);
15  rect(-20, -20, 40, 40);
16
17 }

```



The top screen-shot is of the program running with the `pushMatrix()` and `popMatrix()` commands.

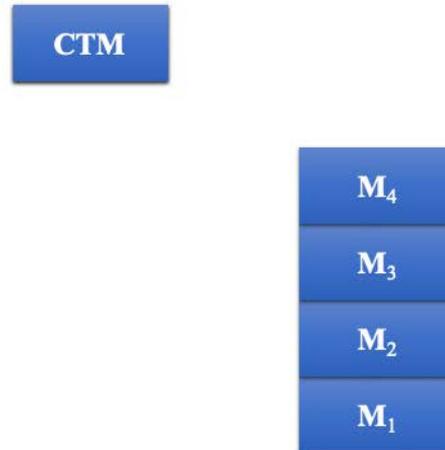
The bottom screen-shot is of the program running without them.

See how the `pushMatrix()/popMatrix()` isolates the first rotation command in the top screen-shot, so that the second rectangle is not affected by the first rotation.

In the bottom screen-shot the first rotation is also applied to the second rectangle, meaning that the second rectangle orbits around the first, rather than rotating independently of it.

Where are those matrices stored?

- `pushMatrix()` pushes the *current transformation matrix* on to the top of a **stack** of matrices
- `popMatrix()` pops the top matrix off that stack and makes it the new *current transformation matrix*



156

The **stack** allows you to store up multiple matrices, as necessary. Processing has a stack that can handle up to 32 matrices. Any more than that will cause an error, but anything close to that and you really need to think about whether you could write your program more efficiently.

You should *always* have a `popMatrix()` to match every `pushMatrix()`.

Errors that can happen:

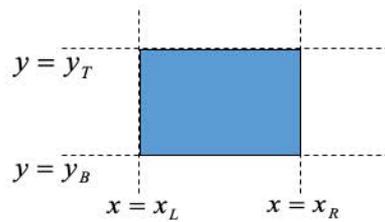
Calling `popMatrix()` when there are no matrices on the stack — Processing halts with the error message “missing a `pushMatrix()` to go with that `popMatrix()`”

Calling `pushMatrix()` when there are already 32 matrices on the stack — Processing halts with the error message “`pushMatrix()` cannot use push more than 32 times”

Unlike *current transformation matrix*, the stack does *not* get cleared every time you call `draw()`. This means that, if you forget to match every `pushMatrix()` with a `popMatrix()` you will quickly (in 32 cycles) reach the limit of what can be pushed on the stack.

Clipping

- what do we do about stuff that is drawn outside the window?
- we need to **clip** so that we only draw what is actually in the window
- clipping points against a rectangle



need to check against four edges:

$$x = x_L$$

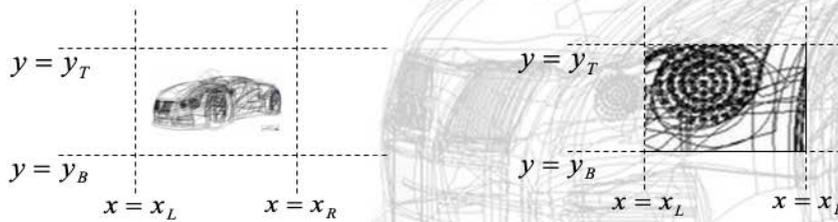
$$x = x_R$$

$$y = y_B$$

$$y = y_T$$

Clipping

- it is very expensive to check every pixel that is drawn against the window
- **Why? Consider the two commonest cases:**
 - Everything fits in the window: so clipping each pixel is an unnecessary waste of time
 - Almost everything is outside the window: so drawing every pixel is a waste of time

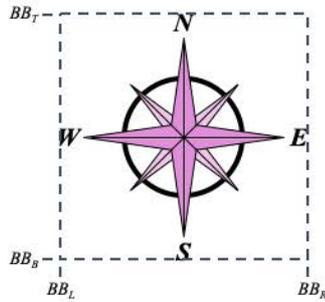


158

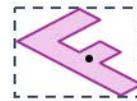
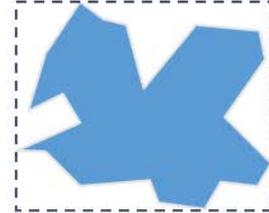
The most extreme case is that of a world map. Consider Google Maps, zoomed in to your street in your home town. Google Maps knows about the entire world, but it must only draw the tiny portion of the world that is visible in the window.

Bounding boxes

- bounding boxes can be used to speed up some operations by combining a whole bunch of objects into a single box
- the box is defined by the maximum and minimum x and y values



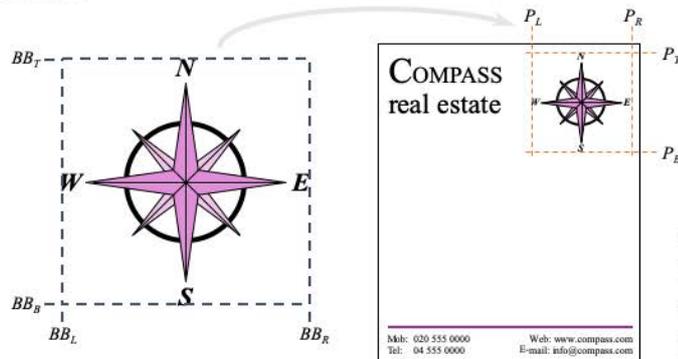
This is a block
of text to draw
on the screen



159

Object inclusion with bounding boxes

- including one object inside another can be easily done if bounding boxes are known and used

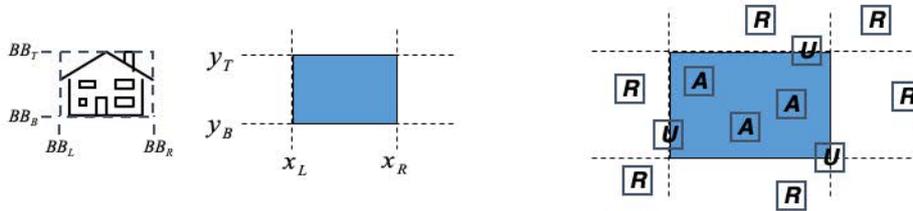


use the eight values to translate and scale the original to the appropriate position in the destination document

160

Clipping with bounding boxes

- do a quick *accept/reject/unsure* test to the bounding box then apply clipping to only the *unsure* objects



$$BB_L > x_R \vee BB_R < x_L \vee BB_T > y_B \vee BB_B < y_T \Rightarrow \text{REJECT}$$

$$BB_L \geq x_L \wedge BB_R \leq x_R \wedge BB_T \geq y_T \wedge BB_B \leq y_B \Rightarrow \text{ACCEPT}$$

otherwise \Rightarrow clip at next higher level of detail

161

Here we are assuming that the y -axis points down, that is, the bottom has a higher y -value than the top.

We have used the standard notation for AND (an \wedge shape) and OR (a \vee shape).

Clipping Bézier curves

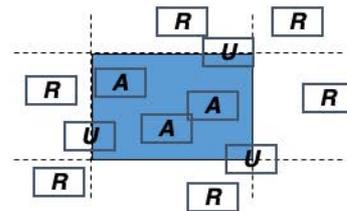
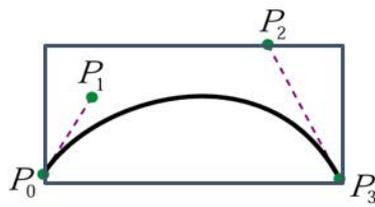
If flat \Rightarrow draw using a clipped line drawing algorithm

Else consider the Bézier's bounding box

accept \Rightarrow draw using normal (unclipped) Bézier algorithm

reject \Rightarrow do not draw at all

unsure \Rightarrow split into two Béziers, recurse



162

Clipping lines against a rectangle

- you can naïvely check every line against each of the four edges
 - this works but is inefficient
- adding a little cleverness improves efficiency enormously
 - Cohen-Sutherland clipping algorithm

163

The naïve check means that you check a line against all four edges. Then, if you find an intersection that lies between the two end points you need to make a decision as to which are the end points of the piece of the line that goes through the rectangle.

An intersection calculation needs three subtractions, one addition, a multiplication and a division: it is computationally expensive (see three slides later for details).

The diagram shows a range of examples that demonstrate why this is inefficient:

- No line has more than two intersections with the box, so doing four intersection calculations is wasteful.
- Many lines are entirely above, below, to the left or to the right of the bounding box so, for these lines, doing any intersection calculations is wasteful.
- Some lines are entirely inside the box so, for these lines, doing any intersection calculations is wasteful.

We instead can do some simple and fast tests to see whether we need to do the calculations and to determine which calculations to do.

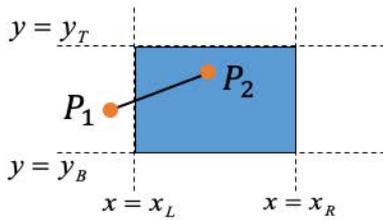
Clipping a line against a rectangle — naïvely

P_1 to $P_2 = (x_1, y_1)$ to (x_2, y_2)

$$P(t) = (1 - t)P_1 + tP_2$$

$$x(t) = (1 - t)x_1 + tx_2$$

$$y(t) = (1 - t)y_1 + ty_2$$



to intersect with $x = x_L$

if $(x_1 = x_2)$ then no intersection

else

$$t_L = \frac{x_L - x_1}{x_2 - x_1}$$

if $(0 \leq t_L \leq 1)$ then

line segment intersects $x = x_L$ at

$(x(t_L), y(t_L))$

else line segment does not intersect

Need to do this for each of the four edges.

It is naïve because a lot of unnecessary operations will be done for most lines.

164

Where does the equation for t_L come from?

From the equation for $x(t)$, which can be written for x_L as:

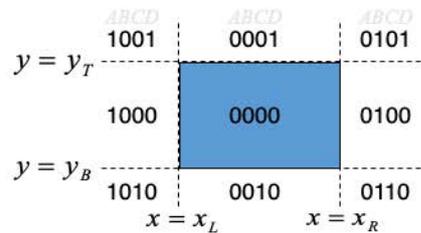
$$x_L = (1 - t_L)x_1 + t_L x_2$$

Try this algorithm for the left edge for all of the examples from the previous slide. Of those 12 lines, one has $x_1 = x_2$, so has no intersection. The other 11 all have intersections of the infinite line with the left edge. Of those, only 3 have an intersection that occurs between the two end points, that is, for $0 \leq t_L \leq 1$. So only 3 of the 12 lines needed these calculations to be done, and 2 of those lines are actually entirely outside the rectangle entirely, so there must be a more efficient way to do this.

Cohen-Sutherland clipper I

- make a four bit code, one bit for each inequality

$$A \equiv x < x_L \quad B \equiv x > x_R \quad C \equiv y < y_B \quad D \equiv y > y_T$$



- evaluate this for both endpoints of the line

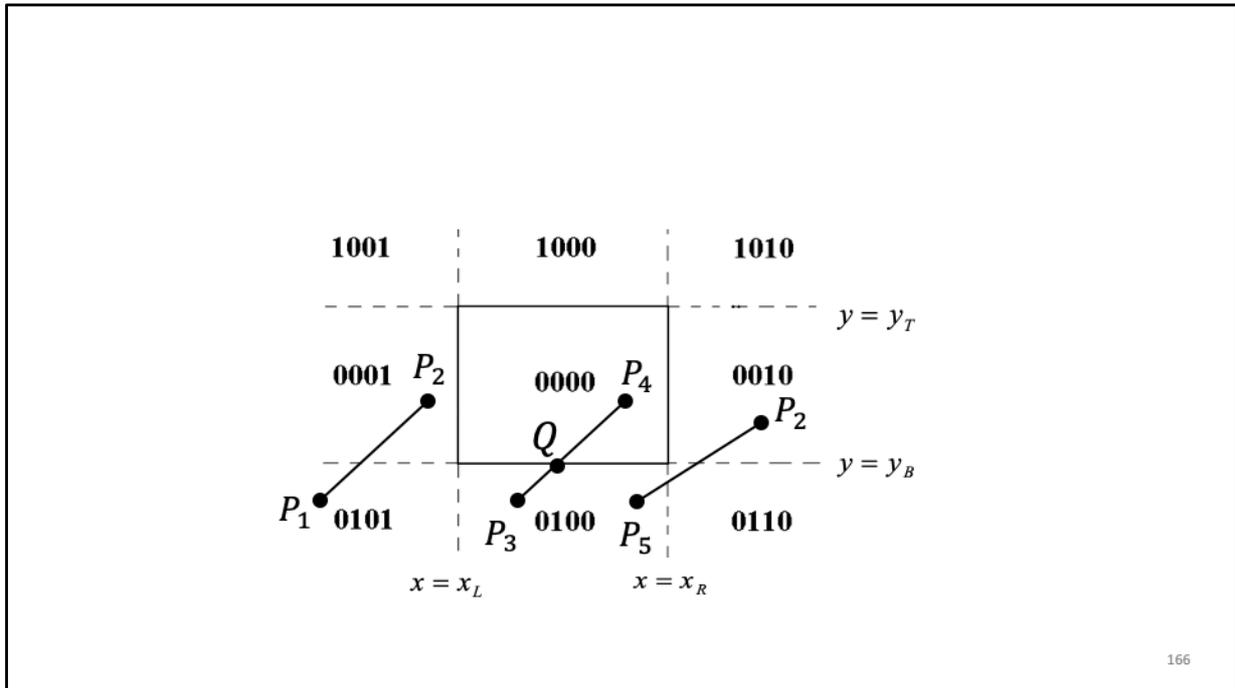
$$Q_1 = A_1 B_1 C_1 D_1 \quad Q_2 = A_2 B_2 C_2 D_2$$

Ivan Sutherland is one of the founders of Evans & Sutherland, manufacturers of flight simulator systems

A hardware version of this: "A clipping divider", R.F. Sproull & I.E. Sutherland, Fall Joint Computer Conference, 1968

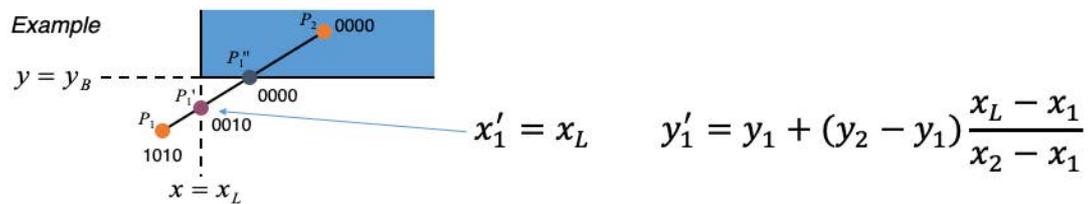
165

It is very fast to do an inequality test. An inequality test is essentially a single subtraction followed by a test for negative or positive.



Cohen-Sutherland clipper 2

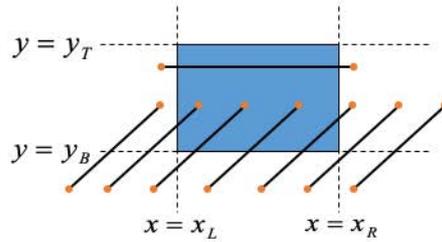
- $Q_1 = 0 \wedge Q_2 = 0$
 - both ends in rectangle *ACCEPT*
- $Q_1 \wedge Q_2 \neq 0$
 - both ends outside and in same half-plane *REJECT*
- otherwise
 - need to intersect line with one of the edges and start again
 - you must *always* re-evaluate Q and recheck the above tests after doing a single clip
 - the 1 bits tell you which edge to clip against



167

Cohen-Sutherland clipper 3

- if code has more than a single 1 then you cannot tell which is the best: simply select one and loop again Why not?
- horizontal and vertical lines are not a problem Why?
- need a line drawing algorithm that can cope with floating-point endpoint co-ordinates Why?

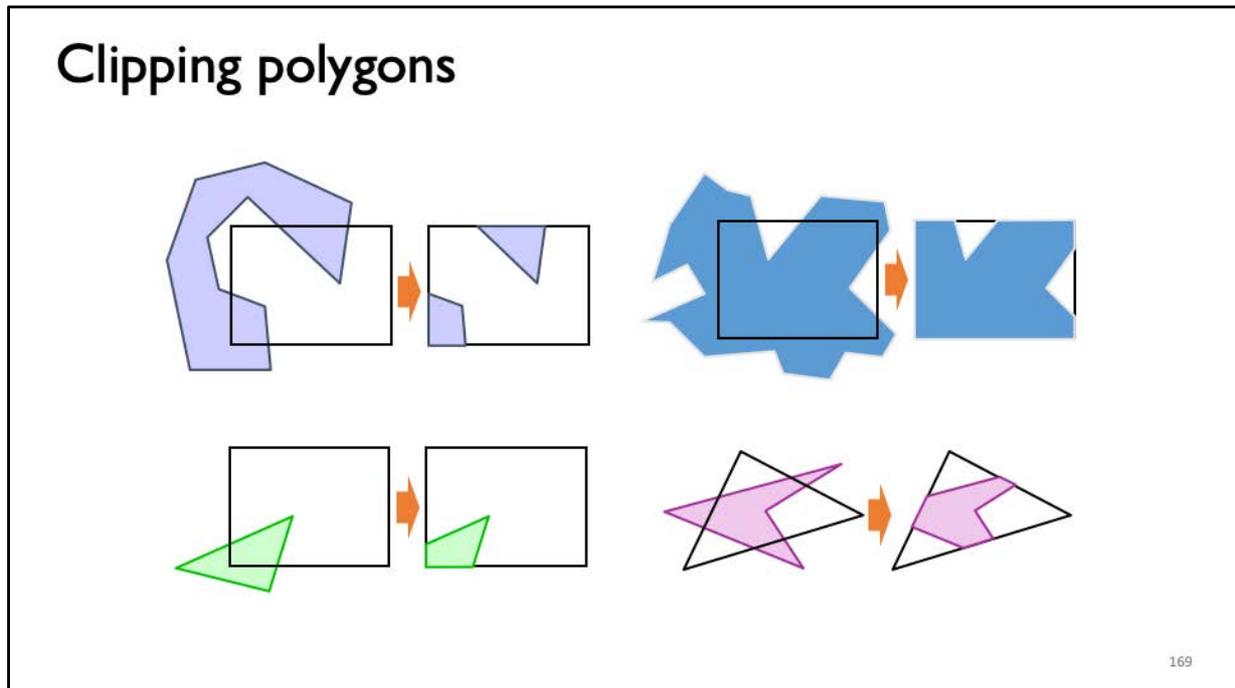


Exercise: what happens in each of the cases at left?

[Assume that, where there is a choice, the algorithm always tries to intersect with x_L or x_R before y_B or y_T .]

Try some other cases of your own devising.

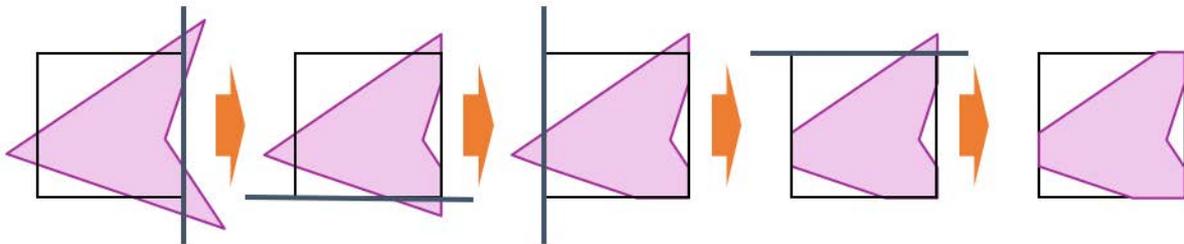
168



This is no longer about just line clipping. If we clip just the lines, using the algorithm that we just learnt, then we end up with a shape that is not closed and therefore we cannot fill it. We somehow need to retain a closed polygon, which means that the clipped polygon may end up having bits of the enclosing rectangle as its edges.

Sutherland-Hodgman polygon clipping I

- clips an arbitrary polygon against an arbitrary *convex* polygon
 - basic algorithm clips an arbitrary polygon against a single infinite clip edge
 - so we reduce a complex algorithm to a simpler one which we call recursively
 - the polygon is clipped against one edge at a time, passing the result on to the next stage



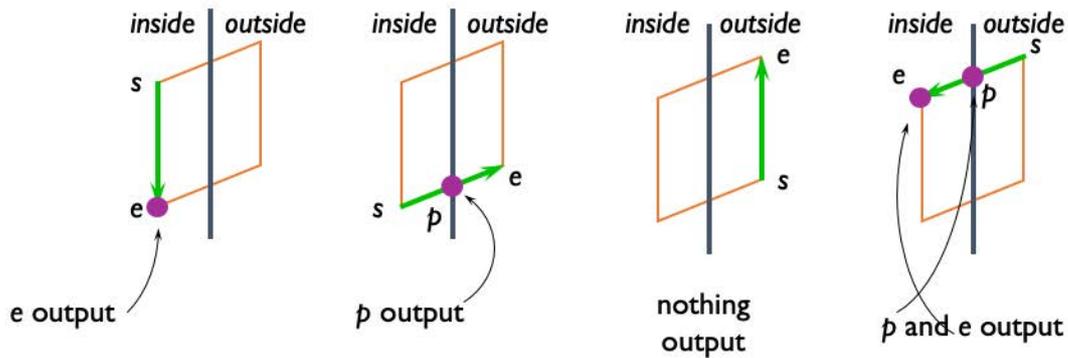
Sutherland & Hodgman, "Reentrant Polygon Clipping," *Comm. ACM*, 17(1), 1974

170

Clipping against an infinitely long edge is relatively easy (see next slides).

Sutherland-Hodgman polygon clipping 2

- the algorithm progresses around the polygon checking if each edge crosses the clipping line and outputting the appropriate points

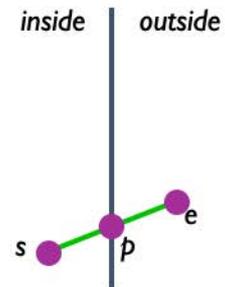


Exercise: the Sutherland-Hodgman algorithm may introduce new edges along the edge of the clipping polygon — when does this happen and why?

Sutherland-Hodgman polygon clipping 3

- line segment defined by (x_s, y_s) and (x_e, y_e)
- line segment is: $p(t) = (1-t)s + te, 0 \leq t \leq 1$
- clipping edge defined by $ax + by + c = 0$

- test to see which side of edge s and e are on:
 - $k = ax + by + c$
 - k negative: inside, k positive: outside, $k=0$: on edge



- if k_s and k_e differ in sign then intersection point can be found by solving:

$$a[(1-t)x_s + tx_e] + b[(1-t)y_s + ty_e] + c = 0$$

$$\Rightarrow t = \frac{ax_s + by_s + c}{a(x_s - x_e) + b(y_s - y_e)}$$

172